Non-Photorealistic Rendering Techniques for Real-Time Character Animation

Diplomarbeit im Studiengang Informatik von

Jérôme Thoma
Matr.-Nr. 208 795

Referent:
Prof. Dr. R. Westermann
Lehr- und Forschungsgebiet Visualisierung
Rheinisch-Westfälische Technische Hochschule Aachen

Koreferent:
Prof. G. Trogemann
Laboratory for Mixed Realities
Kunsthochschule für Medien Köln

10th December 2002

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Aachen, den 10. Dezember 2002

Contents

1 Introduction **1**

2 Fundamentals **3**
2.1 Non-Photorealistic Rendering 3
2.1.1 Introduction 3
2.1.2 Psychological Aspects of Visual Perception 6
2.1.3 Findings from Art Theory 13
2.1.4 Computer Graphics and Art History 19
2.1.5 Categorisation 22
2.1.6 Applications 24
2.2 Graphics Programming 29
2.2.1 OpenGL 29
2.2.2 OpenGL and Realtime NPR 34
2.3 Character Animation 36
2.3.1 Character Animation Techniques 37
2.3.2 Rendering Issues 38
2.4 Trick17 40
2.4.1 Overview 40
2.4.2 Integrating new Rendering Modules 42
2.4.3 Workflow 43

3 Shading **45**
3.1 Cartoon Shading 46
3.1.1 Lighting Model 46
3.1.2 Implementation 48
3.1.3 Results 51
3.1.4 Future Work 53
3.2 Textured Shading 55
3.2.1 Textured Shading 55
3.2.2 Implementation 56
3.2.3 Results and Discussion 59

3.3 Half-Toning . . . 59
3.3.1 Half-Toning . . . 60
3.3.2 Shading Model . . . 60
3.3.3 Implementation . . . 64
3.3.4 Examples . . . 65
3.3.5 Results . . . 66
3.4 Conclusions . . . 68

4 Outlines **69**
4.1 Introduction . . . 69
4.1.1 Definitions . . . 69
4.1.2 Overview . . . 70
4.1.3 Character Animation Issues . . . 72
4.2 Wireframe Outlines . . . 73
4.2.1 Method . . . 73
4.2.2 Implementation . . . 74
4.2.3 Examples and Discussion . . . 75
4.3 Per-Pixel Outlines . . . 77
4.3.1 Method . . . 77
4.3.2 Implementation . . . 78
4.3.3 Discussion . . . 79
4.4 Geometric Strokes . . . 81
4.4.1 Method . . . 81
4.4.2 Preprocessing . . . 82
4.4.3 Silhouette Extraction . . . 83
4.4.4 Visibility . . . 86
4.4.5 Simplification . . . 89
4.4.6 Stroke Building . . . 89
4.4.7 Rendering . . . 90
4.4.8 Examples and Discussion . . . 95
4.5 Discussion . . . 98

5 Discussion **101**
5.1 Critical Assessment . . . 101
5.2 Future Work . . . 102
5.2.1 NPR Techniques for Character Animation . . . 102
5.2.2 The Future of Computer Depiction . . . 103
5.3 Summary . . . 104

A Vertex Programs **107**
A.1 Per-Pixel Silhouettes 107
A.2 Geometric Strokes 109

B Artistic Examples **113**

Chapter 1

Introduction

In this thesis, we present the results of an investigation into the use of non-photorealistic rendering techniques for real-time character animation. So far, most non-photorealistic rendering methods have only been applied to simple, static scenes. In the context of real-time character animation, a number of special technical issues arise that put much higher requirements on the rendering algorithms. Based on established techniques, we examine and extend a number of different non-photorealistic rendering techniques and integrate them into a real-time animation system. We analyse the results with respect to a number of different criteria and develop ideas for extending and refining the presented techniques. Furthermore, we motivate our work by putting it into the context of computer depiction and related fields such as fine arts and psychology.

This thesis is structured as follows:

- In **Chapter 2**, we introduce the field of non-photorealistic rendering from different viewpoints and provide the necessary background information on graphics programming and character animation for understanding the subsequent chapters.

- In **Chapter 3**, we investigate three different non-photorealistic shading models. They are based on established shading models that we adjusted and extended for our purposes. We complement each model with an efficient OpenGL implementation and an analysis of its properties in the context of this thesis.

- In **Chapter 4**, we compare three different approaches to outline drawings. We put a special focus on geometric silhouette extraction and rendering methods and we present novel techniques for determining

the visibility of line segments using the graphics hardware and for rendering stylised strokes. All presented techniques are complemented with efficient OpenGL implementations and an in-depth analysis of the results.

- In **Chapter 5**, we assess the work that was done during this project and present ideas for future research.
- Finally two **Appendices** contain the assembler code of the vertex programs presented in the previous chapters and some artwork that was created during this project.

Chapter 2

Fundamentals

In this chapter we are going to introduce the basic concepts and tools that the subsequent chapters will be based on. We start with an introduction to non-photorealistic rendering (NPR) in which we will attempt to explain not only what NPR is, but also why it is useful. Most of the material is not new, but we try to integrate concepts from many different fields that are often considered as non-related. We then proceed to a short introduction to the graphics API in which the rendering algorithms are implemented with a strong focus on those aspects that are important for understanding the rendering modules developed in the next chapters. In the next section, we present the field of character animation from the point of view of rendering. Our contribution consists in a discussion of the special issues that arise in the context of NPR. We close with an overview of the software system in which our rendering modules were integrated and evaluated.

2.1 Non-Photorealistic Rendering

2.1.1 Introduction

Although the field of non-photorealistic rendering has existed for more than a decade, it has for a long time not been taken seriously by large parts of the research community. This is partly due to the lack of a coherent theory or conceptual framework that ties all the different efforts in NPR together. Only recently efforts have been undertaken to develop such a framework and to analyse the concept of NPR in the context of computer graphics, art history and theory, and cognitive psychology. In this section we will first present a definition of NPR and then discuss some fundamental concepts that will be treated in further detail in the subsequent sections.

What is NPR?

In order to explain what is meant by non-photorealistic rendering, it is useful to take a look at the history of 3D computer graphics. For the (roughly) first twenty years, developments in 3D computer graphics have revolved around the idea of simulating a film set or a photography studio (which is also reflected by its vocabulary). A scene consisting of 3D objects is illuminated by a number of virtual light sources and images are generated by a virtual camera that is placed in the scene. The idea is to generate 2D images of the scene by emitting light from the light sources into the scene, computing the interaction of the light with the surface of the 3D models, and capturing that portion of the light that reaches the camera on a virtual film plate. A vast number of different models have been explored that approximate these physical processes with the aim of generating images that are indistinguishable from photographic images. Therefore this field of computer graphics is also called *Photorealistic Rendering*. For a long time, this approach was considered to be the holy grail of computer graphics, but as we will see in the rest of this thesis, it constitutes only one of many possible approaches. As researchers started to explore alternative rendering techniques, they needed to differentiate themselves from the rest of the computer graphics community and therefore called their field of research *Non-Photorealistic Rendering* for lack of a better term. Although this name is not very expressive, it is still widely used because no better term was found that covers all efforts in this area. For instance the term *Artistic Rendering* is more appropriate for a subset of NPR techniques, but not all efforts in this field can claim to be of artistic value. Maybe *Stylised Rendering* captures the idea best, but as it is even less common than the original denomination, we will stick to the term non-photorealistic rendering for the rest of this thesis. So what is NPR? Lacking a better definition we will simply say that a rendering technique is called non-photorealistic if its goal is different from achieving the illusion of photorealism. This is not very satisfying, but hopefully the idea will become clearer as we proceed.

Why is NPR useful?

The first question that arises (and that researchers in this field are often confronted with) is: What is it useful for, anyway? For now, we will only hint at the answer because the main arguments in favor of NPR will be developed in the next few sections.

The following quote from the reference work on computer graphics by Foley et al. illustrates the prevailing idea about the aims of computer graphics

a few years ago (and still often today)[20]:

> *The creation of realistic pictures is an important goal in fields such as simulation, design, entertainment and advertising, research and education, and command and control.*
> *Foley et al.*

Foley et al. argue that realistic images can support designers in evaluating new products (e.g. car designs) by replacing model building with computer-generated images. Furthermore they claim that highly realistic graphics will turn computer games into a more enjoyable experience. Similar arguments are also provided for the other areas mentioned. This is certainly true in some cases, as for instance for special effects in movies, where the aim is to seamlessly fuse computer-generated elements with filmed material, but in general it only considers part of the imagery traditionally used in the cited areas. For instance it may be useful for designers to be able to generate photorealistic images of the finished product, but during the design process they prefer to work with sketches and conceptual drawings that are better suited for explaining the basic concept of a new product or showing its inner structure. Furthermore there are many research areas that can benefit a lot from automatically generated images based on purely abstract data, but how can one create photorealistic images of data that has no counterpart in the visual world? Foley et al. also mention this as they talk about visualising molecular structures using *stylised* ball-and-stick models, but they fail to realise the contradiction in their arguments. Or one could consider educational course books where most of the pictures are not photographs, but rather diagrams and illustrations that are better able to communicate the important aspects of a topic. Finally in the world of entertainment, realistic imagery often destroys the so-called suspension of disbelief by being not perfectly realistic (we will treat this in more detail in 2.1.3). Therefore one could also say that

> *a picture is worth a thousand words,*
> *but an image is worth a thousand pictures*
> *Gooch*

In this quote by Gooch[27], a picture stands for a photorealistic image or a photograph, whereas an image can be anything from a drawing or a diagram to a painting as long as it *helps to communicate the intended idea* (note that the distinction between the terms image and picture is not well-defined, Durand [18] for instance defines them the other way round). This does not imply that photographs are always inferior to stylised images, but just as no

user manual contains only photographs, should computer-generated imagery not be restricted to photorealistic renderings.

There is also a practical problem inherent in the concept of accurately simulating physical processes which is the *problem of complexity*. As we progress from very simple physical models to more sophisticated ones, the complexity involved in evaluating the models grows exponentially. Even though advances in computer hardware allow us to push back the limits of practicability, we will always reach a point where we need to introduce simplifications. Additionally, because the world has such a complex structure, a truly photorealistic image needs to be generated from extremely detailed object descriptions that are animated in a very accurate way thus requiring a great modelling and animation effort. If we consider how long it takes to create a typical animated movie from the modelling phase to the final rendered frames (which is of the order of years involving hundreds of artists and large 'rendering farms'), it becomes clear that this constitutes a real-world problem and not just a theoretical reasoning.

2.1.2 Psychological Aspects of Visual Perception

We are now going to give an introduction to visual perception from the point of view of Psychology. This is not meant to be comprehensive, but only aims at raising awareness that visual perception is not a passive process and that what we see is influenced by a large number of factors, including context, expectations and previous knowledge. We begin by presenting some examples of perceptual phenomena and then show how different psychological models can account for them. Note that there does not exist a universal psychological theory that can explain all perceptual phenomena and that each model focuses only on some aspects of perception.

Phenomena

Consider the following situation: You look at a book (or any other object) that is lying on a table and is illuminated by a lamp. Now what do you see? Well, probably exactly what we just described, i.e. a book of a certain colour and size at a fixed location in space. But if you consider the image that is continually projected onto your retina by the scene, you will notice that most of these properties are not contained in the retinal image. First of all the retinal image changes constantly because of small movements of your body and eyes. Therefore if our eyes passively recorded the world, everything around us would be in constant movement (This can be experienced if you watch for instance a holiday video that was filmed without a tripod. Because

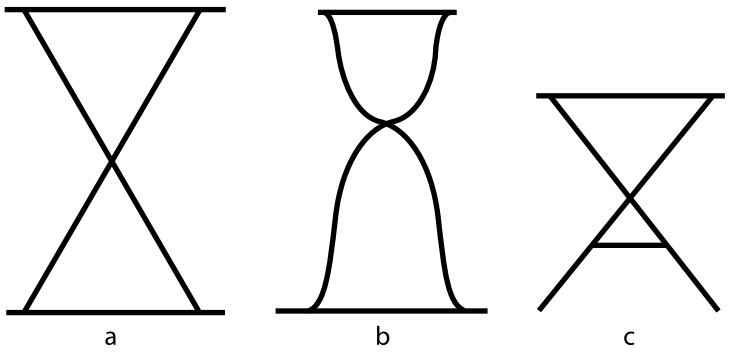


Figure 2.1: The power of expectation a) The original stimulus b) Drawing by a person expecting an hourglass c) Drawing by a person expecting a table

of all the small involuntary movements of the person handling the camera, the image is always shaky). Or take the size of the book. If you step back a bit, the retinal image of the book becomes smaller, but the perceived book remains of constant size. This kind of phenomenon is called *Constancy Effect* and is one of the most important perceptual effects in daily life, because without it we would be unable to interact with our environment. Constancy effects exist also for colour and form and can be illustrated best in circumstances where they break down. If we for instance look at a group of people that are far away we perceive them as being of normal height, but if we look down from the roof of a tall building, size constancy breaks down and people suddenly look like tiny ants. Another constancy effect that is very important for depiction is colour constancy. If you look at any object around you that is of a constant colour, you will notice that no matter how it is illuminated, you will always perceive the 'real' colour of it, even though the retinal image contains many different colours. We are somehow able to account for the lighting situation and to separate the base colour and the shading of an object.

Expectation can have an even more dramatic effect on perception as can be demonstrated by an experiment described in [10]. Subjects were shown the image in Figure 2.1b for a very short time on a projection screen and were then asked to draw what they had just seen. If the subjects had been previously told that they were going to see an image of an hourglass, the image they drew looked similar to Figure 2.1b. If on the other hand they were told to expect the image of a table, the result looked very different (Figure 2.1c). Such *priming* experiments are a common tool of psychologists to explore the role of expectation in mental processes and have been applied successfully to many other phenomena [24]. The same effect can also be

Figure 2.2: Example of a typical ink-blot used as an ambiguous stimulus in the Rorschach-Test

noticed in daily life. For instance we tend to interpret every odd-looking plant as a dangerous creature when walking alone in a wood at night. Although we know exactly that this is absurd, we cannot help it. These examples show that instead of passively recording the world, we actively form it. In fact the urge of *interpretation* is very strong and difficult to switch off. When looking at an abstract painting, it is hard for us to accept that it does not represent anything but itself and we always try to see something in it. The example of the famous ink-blots (see Figure 2.2), used by psychologists in the Rorschach-test to evaluate the mental state of a patient, shows that interpretation is a process that varies among individuals and is strongly influenced by the current situation. A depressed person might see a demon in Figure 2.2, whereas someone who is in a more positive mood might identify it as a butterfly.

Finally perception is based on *exploration* and guided by *attention*. If a subject is told to look at a moderately complex scene for some time and is subsequently shown a photograph of the same scene, they will always be surprised by all the things they did not see while they visually explored the scene. Although all these things were comprised in the retinal image at some time, they were discarded somewhere along the way from sensation to perception. This filter causes us to see only those things that we pay attention to [55].

On a high level, these phenomena can be explained by the *struggle for survival*. Attention for instance can be considered as a mechanism to separate important sensations from unimportant ones. If we always saw everything around us, we would be so overwhelmed by the sheer complexity of the world that we would be unable to process it all. In this case we would not be able to survive long, because we might for instance perceive the piece of paper lying on the road, but fail to see the truck that is going to run us over. On

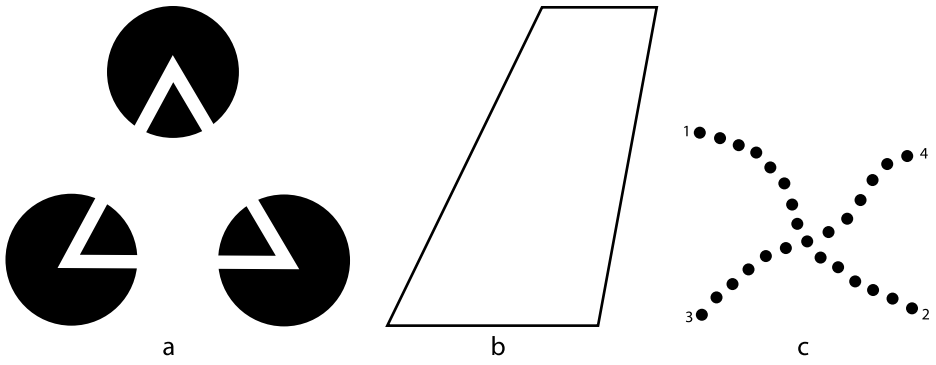


Figure 2.3: Gestalt Theory a) The white triangle that is perceived is not included in the image b) The principle of simplicity causes the shape to be perceived as a rectangle in space rather than as a more complex planar shape c) Because of the principle of direction, two lines (1,2) and (3,4) are perceived as all other combinations would introduce sharp angles

the level of perceptual processes, the phenomena we described are much more difficult to explain. Although some models of visual perception have been developed, they can only account for a subset of the effects we described. In the rest of this section, we will introduce two such models from the field of psychology.

Gestalt Theory

In the 1920s, a number of psychologists started investigating the relationship between visual stimuli and perceived shapes. Their main thesis was that *the whole is more than the sum of its components*, meaning that our perceptual system organises and classifies visual stimuli into higher-order constructs that are not necessarily contained in the retinal image. Figure 2.3a for instance shows three black circles and a number of white line segments, but we perceive a white triangle that seems to connect these segments. A number of similar experiments lead to the conclusion that the perception of individual elements depends on the whole configuration. These findings are summarised in the *Principles of Gestalt Theory* that we will briefly present now (from [55]):

- *Simplicity:* A visual pattern is always perceived in the way that leads to the simplest possible structure. For instance the form in Figure 2.3b is normally perceived as a rectangle in 3D space, because a rectangle is simpler (due to its regularity) than a form with unequal lengths and angles.

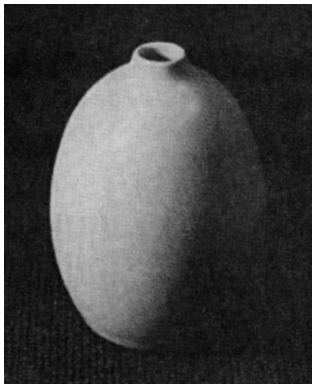


Figure 2.4: In the lower right corner, the contours of the vase are almost not discernible, but nevertheless we are able to perceive its shape.

- *Proximity:* Visual patterns that lie close together are grouped together.
- *Similarity:* Visual patterns that share some properties are grouped together. These properties can for instance include form, colour, orientation and size.
- *Direction:* Elements that are arranged along a continuous, smooth direction tend to be seen together (Figure 2.3c).
- *Common Fate:* Elements from a larger group that behave in a similar manner tend to be grouped (e.g. dancers moving in the same direction)
- *Objective Set:* An organisation that is seen in one setup tends to be propagated to a similar setup if it is presented immediately afterwards.

Critics of the Gestalt theory reproach it for only being able to *explain* visual phenomena, but not to *predict* them, because in complex situations more than one principle could be applied to the same setup, leading to different results. In computer graphics, these principles could be useful for deriving better depiction methods for applications where it is important to emphasise the relationships between different elements.

Cognitive Psychology

In recent years cognitive psychology has approached the problem of how we perceive the world from a different direction, i.e. from the point of view of data-processing. This is also motivated by the practical use of such theories in fields of computer science such as computer vision. Current models from

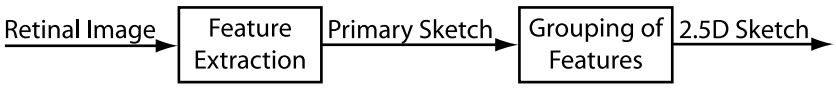


Figure 2.5: The visual pipeline according to Marr

cognitive sciences integrate two older models into a coherent framework, the first being that perception is a 'bottom-up' process that starts with the retinal image, extracts simple features such as outlines and organises the features into more and more complex objects. The second theory states that perception is guided by previous knowledge and that it consists mainly in matching mental representations of the world with the retinal image ('top-down' approach). Experimental data suggests that these two strategies are pursued simultaneously and that they influence each other.

In 1982, Marr presented the *computational approach* to bottom-up visual processing [24]. This model divided object perception into a number of stages that are performed sequentially (see Figure 2.5). The first stage consists in feature extraction from the retinal image. These features include edges and corners that are represented by changes in luminance in the retinal image. Although feature extraction seems to be a simple task, Figure 2.4 shows that often the outlines of an object are not clearly discernible in the retinal image, because of shading and shadows. In this case it is only possible to correctly determine features if we manage to abstract from influence of the lighting situation and take advantage of contextual knowledge (in the case of Figure 2.4 we know what shape to expect once we have identified that we are supposed to see a vase and can therefore fill in the missing information from what we know about typical vases). During the second stage the low-level features extracted from both retinal images are integrated into a 2.5D sketch of the object. We will not go into further details about this stage, because it is not yet known exactly how it works, but already the first stage leads to interesting implications for depiction. As this stage tries to determine edges and corners in retinal images, we can assist it by including object outlines in rendered images and thereby guarantee that the features are detected correctly. Furthermore it can be useful to include outlines that are hidden by a closer object as long as we make them discernible from visible ones by using a different drawing style (e.g. stippled lines). Of course this observation is not new and has been used by illustrators for a long time, but the psychological model can be considered as a justification for the intuitive observation. We will therefore present algorithms for outline drawings in Chapter 4.

One of the most important cues for determining the depth and shape of

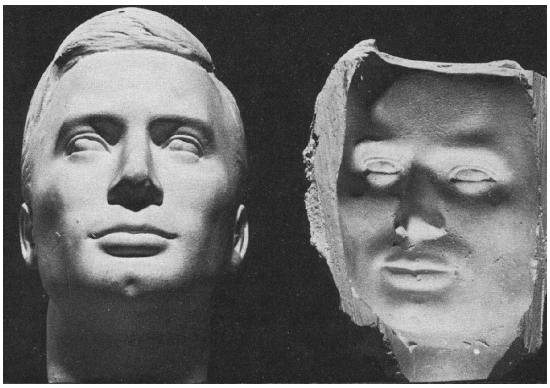


Figure 2.6: Example of the influence of high-level knowledge on shape perception. Left: Sculpture of a Head. Right: A Mould of the same head; although it is concave it is perceived as convex.

objects is given by shading [49]. The fact that many animals have evolved to compensate shading effects with special fur colouring suggests that it is one of the most primitive mechanism for extracting depth information from retinal images. 'Shape from shading' has only been explored by psychologists in recent years and our knowledge of the processes that are involved is still rudimentary. Experiments have shown that depth information is extracted at an early stage of visual perception. If no further cues about the current lighting situation are supplied, it is assumed that a scene is illuminated by a single light source that is located above the viewer (which can be explained by the fact that the sun is the most important light source in nature). Furthermore perception seems to assume that the lighting situation is constant over a scene. In the context of computer graphics this means that all objects in a scene should be illuminated by the same main light sources (this does not apply to rim lights that are only used to separate figures from the background). To be correctly interpreted, shading must be combined with information about the delineation of objects. As we have seen above, edge information is implicitly given by the contrast between overlapping objects, but experiments have also shown that explicit outlines can strengthen the impression of depth. Maybe the most important finding in the context of depiction is that shading does not need to be physically correct to convey a sense of shape as long as the relationship between light and dark areas is respected. This can be explained by the fact that all perceptual processes are based on relative values (relationships) rather than absolute values (e.g. brightness values). Because of this, artists have been able to create images that convey a strong sense of depth using a very limited palette. Finally it

should be noted that low-level shape information can be overridden by high-level knowledge at a later processing stage which can be illustrated by the fact that it is virtually impossible to see the face in Figure 2.6 as the concave mask that it is in reality. Because of our knowledge of faces, it 'must' be convex and can therefore not be perceived differently if no stereoscopic clues are provided.

At this point it should have become clear that if we create images that are meant to be viewed by humans, we have to account for the way in which we perceive them and that no single depiction method can possibly be optimal for all intended communicative uses. We will now further develop this idea by taking a closer look at art theory.

2.1.3 Findings from Art Theory

Considering the wealth of knowledge that has been gained from thousands of years of exploration of the possibilities and limits of visual expression by fine artists, it is surprising that computer graphics research has shown so little interest in this field. This is mainly due to the fact that fine arts are still believed to be of a purely 'metaphysical' nature and that there is no underlying theoretical knowledge worth examining. While it is certainly true that every creative act is partly guided by intangible 'forces' and 'feelings', many artists use their respective medium (e.g. painting or sculpture) to explore the inner and outer world and our relation to them (it has for instance been reported that Matisse considered his paintings as a way of thinking [11]). In the rest of this section we will present some of the theories that artists and art researchers have developed and show their relation to issues of computer graphics. Although these theories are often developed on a very abstract level, they still have important implications on a practical level.

Concepts of Realism

We are now going to discuss the notion of realism in the context of computer graphics. Before we get started it should be noted that in fine arts, the term Realism is used in a very different sense than in image theory, on which our discussion will be based. In fine arts, the *Naturalist* ideology corresponds to what we will call *Photorealism* in this Section, i.e. a 'faithful' rendering of the material world based on a number of depiction principles (see Section 2.1.4 for more detail). *Realism* on the other hand is not concerned with methods of depiction, but with contents. It designates a short period in art history (the Realist movement, 19th century) and the general concept of showing the world and life as it is, including all those subjects that were

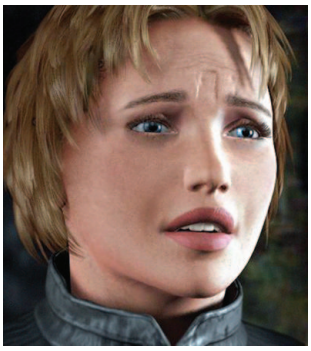


Figure 2.7: Can this image be considered as photorealistic?

usually considered as too profane to be captured on canvas (e.g. workers, old and ill people). It arose as an antipode to the prevailing *Idealist* ideology that claimed that the subject of painting should be the beauty of creation in its ideal, perfect form. As these aspects are not important in our discussion, we refer the interested reader to Herding [29] who offers a comprehensive overview of this subject.

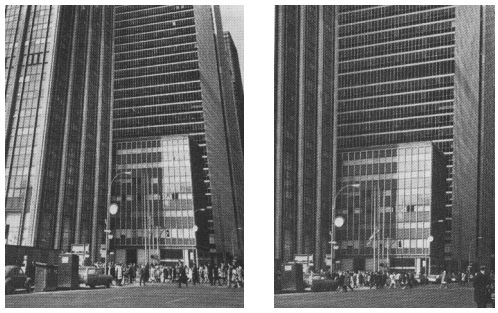
As the above example shows, one must be very careful when speaking of 'realism'. Although everyone has an intuitive idea of what 'realistic' means, a formal definition depends strongly on the context in which it is used. It would be rather tempting to say that an image is realistic if it shows the world *as it is*, i.e. accurately representing the properties of the world. Unfortunately this does not help much. Even if we ignore philosophical questions about the limitations of our knowledge of the world, we still need to define how we measure the properties of things in order to compare them to the properties of the image. As we are unable to directly experience the world around us, we can only measure it indirectly using a medium such as physics or our senses. Because the results will differ considerably depending on which reference system we choose, it is mandatory to include the choice into the definition of realism.

If we now consider the example of *Photorealism* in the context of computer graphics, it is obvious that its definition must be based on physical properties because the rendering techniques that are used to generate photorealistic images (by definition) simulate physical processes. But there is an additional question that needs to be raised for 3D computer graphics that is not necessary in the case of photography (to which photorealistic rendering is often compared): What is the reality that we use to evaluate the degree of visual realism of a rendered image? Two different answers are possible, it is either the reality of things in the world around us or that of the 3D object

descriptions that the image is derived from. If we assume the second answer, then every rendering technique must be considered as photorealistic, because it is always possible to describe the 3D models in such a way that the properties of the rendered image correspond to the properties of the 3D model. As this implies that there is no such thing as a non-photorealistic image, it does not help us to understand the relationship between non-photorealistic and photorealistic rendering methods and can therefore be discarded.

That leaves us with the alternative that a computer-generated image is photorealistic if its properties accurately represent properties of the real world. This seems to be appropriate for scenes that have been derived from measurements of the physical world (e.g. with the use of a laser scanner and photographs), but what about scenes that have no counterpart in the world? In some cases it would be sufficient to change the definition to include objects that *can be created* in the real world (such as an architectural building or a car), but that still only covers a portion of images that are usually labelled as photorealistic. Consider for instance the case of the model of a human character such as shown in Figure 2.7. Why are we tempted to say that this image is photorealistic, even though there does (probably) not exist a living human being that has exactly the same properties? This problem can be compared to issues raised in art theory about the reality of paintings (see [11] for a discussion of the famous Mona Lisa by Leonardo Da Vinci). Again one could extend the definition of realism to include *things that could potentially exist*, but it is impossible to judge where the limit between possible and impossible should be drawn (should for instance bigfoots or unicorns be considered as potentially existing?). We do not claim to know the answer to these questions and we can only safely state that the notion of realism in computer-generated imagery is even more ambiguous than in the case of photography as it involves two projections, first from the real world to a 3D scene and then from the 3D scene to a 2D image. Boehme [11] for instance conjectures that the main reason why we use the term realism so easily with regard to computer-generated images may be linked to the fact that much of our daily life revolves around images or, expressed differently, that we have become visual beings.

This leads immediately to a different notion of realism, because so far we have ignored the role of the beholder in viewing images and reality. As we have seen earlier, our perceptual system transforms the world around us into percepts in a variety of different ways. Notably the way in which we perceive the world and images can differ considerably. Therefore an image that may be considered as realistic in the above sense of a mathematical relationship between physical properties can appear less realistic to a viewer than one that is not physically correct. This phenomenon can be illustrated with an



a b

Figure 2.8: Photographs of a building a) perspectively correct b) perceptually correct

example from photography: When they are confronted with the task of creating a photograph of a very tall building from the ground level, photographers often use optical tricks to counteract the effect of foreshortening that causes the walls of the building to bend towards the center and to converge towards a distant point (see Figure 2.8) [22]. This correction is necessary because we tend to perceive parallel lines as parallel even if the retinal image does not have this property [25]. Despite this example, it is in general not possible to decide once and for all whether an image is realistic in the perceptual sense, because we have seen above that what we perceive depends on our expectations and on the situational context. Furthermore the concept of realism differs slightly from person to person, especially if they have different cultural backgrounds.

The Power of Abstraction

> *Photorealism is like pornography*
> *It leaves nothing to the imagination*
> *Cassidy Curtis*

We will now discuss an inherent problem of photorealism (in the conventional sense) that especially affects animated characters. The issue is that the more realistic a virtual character is, the more it becomes disturbing if small details are flawed. This principle is nicely illustrated by the computer-generated movie Final Fantasy [19]. To date, it is considered as the most ambitious effort to create a motion picture without any real-world elements. It is set in a futuristic world with human main characters. Although great care was taken to model and animate these characters in a very realistic fashion (each character had their own team of modellers, animators, make-up artists etc.),

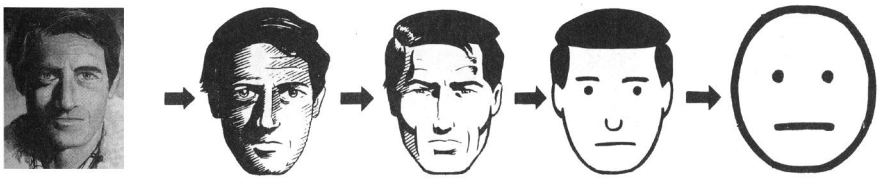


Figure 2.9: Progression from concrete to abstract depiction of a face. The leftmost face represents a single person, whereas the rightmost face can represent (almost) any person.

they just do not look right once they take off their space suits and their face and body becomes visible. Because we have a life-long experience of closely watching the body talk and facial language of people we interact with, it is disturbing to watch a character that seems to be real, but lacks all the subtle clues that help us to determine what a person is really thinking or meaning by a statement. If you closely watch any living person, you will notice the small and often involuntary movements that we do all the time. We are normally not aware of them, but as soon as they are missing, we feel that something is not right. Therefore if a virtual character is not animated perfectly (which is probably impossible), we will never accept it as truly human. Now what happens if we take the opposite approach and move towards more abstract characters? As anyone who has ever watched a cartoon or read a comic story will agree, we suddenly become much more forgiving. Nobody will criticise that there is something wrong with Bugs Bunny, because real hares walk on four legs and are unable to talk. The main reason for this is that nothing in a Bugs Bunny cartoon suggests that he represents a real hare and we therefore do not compare him to one. It is not even clear whether we perceive him as an animal with human behaviour or a human with animal traits.

In his comic book on the theory of comics [40], McCloud presents some of the underlying principles of the phenomena we described above. The first of these is the principle of *amplification through simplification*. By leaving out all irrelevant detail, it is possible to focus the attention of the beholder on those features that are important. As our imagination can fill in all the details that are missing, it is better to leave the task to our mind instead of including them in a wrong way. This is also part of the principle of *universality* that is illustrated in Figure 2.9. The leftmost face can only represent a single person because it determines every little detail of its appearance, whereas the next face is already less explicit. It only fixes some of the character's properties, such as its gender and approximate age. Finally the rightmost image only

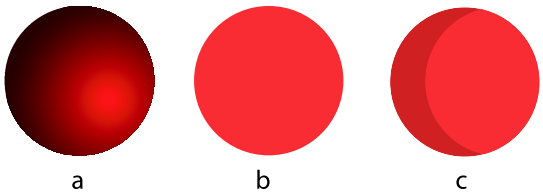


Figure 2.10: Difference between extrinsic and intrinsic properties (surface colour) a) Extrinsic (shading) b) Intrinsic (object colour) c) Hybrid (Hard Shading)

tells us that we are looking at a human face that could either be male or female, old or young. The level of abstraction has an important consequence for our reaction towards it. As a self-centered species we tend to react in a more critical way when we are confronted with other individuals. Therefore comic artists often draw villains in a very detailed style. The hero on the other hand is depicted in a more abstract way, because that gives us the opportunity to identify ourselves with the character and share his problems and triumphs. This greatly supports the *suspension of disbelief* which is a very important concept in any form of entertainment. In order to prevent the audience from wondering whether their experience makes sense, it is best to fix as little detail as possible and leave the task of filling the gaps to the audience, because every beholder develops a slightly different internal image of the world of a movie or game. As a consequence, more abstract depiction methods could be of great interest to character animation, because it is often desirable to create an emotional link between a virtual character and the audience, either for storytelling purposes or even to more easily convince people during a virtual shopping trip that is guided by an avatar.

Extrinsic vs. Intrinsic Properties

> *My business is to paint not what I know, but what I see*
> *William Turner*
>
> *I do not paint what I see, but what I know*
> *Pablo Picasso*

Producing figurative pictures (as opposed to purely abstract pictures) can also be considered as a mapping from scene or object properties to picture properties. The choice of which properties to include and how to represent them is guided by the intent of the image. Object properties can be divided into two groups [18]. *Intrinsic properties* are invariant, which means that

they are independent of the current lighting situation and view angle (e.g. object colour). *Extrinsic properties* change as soon as the camera or lighting setup is modified (e.g. shading or absolute size). In the case of photorealistic rendering, image generation is based only on extrinsic properties. But as we have seen in 2.1.2, constancy effects cause us to ignore these properties and to perceive the invariant qualities of objects around us. If on the other hand we look at pictures, most of the processes that are responsible for the constancy effects do not work. For instance, a white wall will always look white independently of the lighting situation because of chromatic adaptation, but a photograph of the same wall will look either yellow or blue depending on the characteristics of the light source and the type of film that was used. Therefore physical correctness does not guarantee that an image will be perceived as realistic. Furthermore the fact that we need to learn how to interpret shading and perspective distortion in images (see [25]) proves that there might exist more intuitive ways of representing the world. At the other extreme, one could choose to use a depiction method that is only based on intrinsic properties. Although it tends to be more easily understandable, it does not correspond to the way we perceive the world, because even if we do not see it, we are still always aware that distant objects are smaller than nearer ones or that some parts of an object are shadowed. Therefore most painters use a hybrid of extrinsic and intrinsic properties. For instance they often add shading cues in such a way that the intrinsic colour of an object remains dominant. Similarly Cartoon Shading can be considered as a mix of intrinsic and extrinsic properties.

2.1.4 Computer Graphics and Art History

If we look at the history of fine arts from the early middle-ages to today, one can see a number of interesting parallels to the development of computer graphics that can help us to foresee in which direction future research should head.

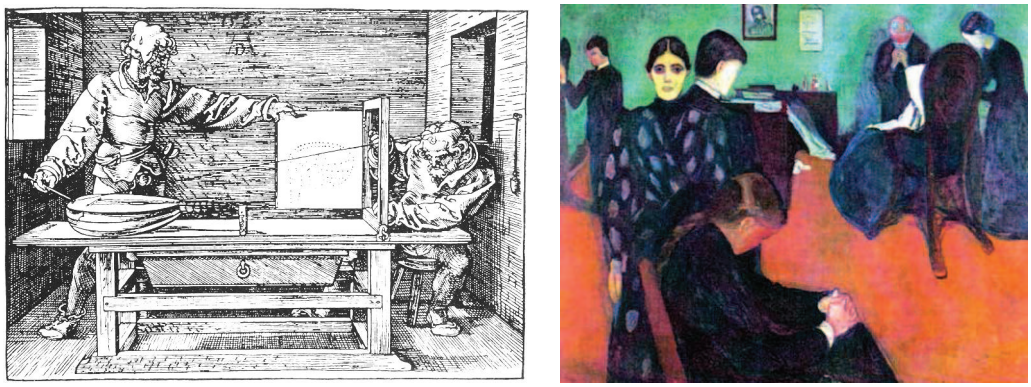
During the *early middle-ages* (up to the end of the thirteenth century) artists mainly chose subjects that belonged to the world of the invisible and supernatural (e.g. angels) [54]. Because these have no counterpart in the visible world, they could only be depicted by using symbols and metaphors and there was no need for depiction techniques that allowed to represent spatial extension (e.g. with perspective projection), volume (e.g. by using light and shadow) or surface materials (see Figure 2.11).

During the *Renaissance* (14th and 15th century) a paradigm shift occurred in all fields of science. Instead of concentrating on the irrational, intangible netherworld, a new interest in the rational and visible world arose.

Figure 2.11: *Abraham's hospitality.* Detail from a 12th century tapestry (Halberstadt Cathedral Museum).

This development also influenced fine arts and was characterised by the rediscovery of the six elements of *Naturalism* (i.e. the three principles of detail, anatomical correctness and correct colour rendition, and the three illusions of space, volume and texture [53]), knowledge that was first developed during the Antique, but which was lost during the following 'dark ages'. This had a great impact on painting techniques. The size of a picture element was for instance no longer solely determined by its importance, but foremost by its location in space, because the illusion of space requires that objects or persons that are further away from the viewer should be drawn smaller than those that are closer. Similarly rendering colours correctly meant that the interplay of materials and light sources had to be respected. During this period a number of tools (see Figure 2.12a) were invented in order to support the artist in creating naturalistic paintings, because it was already known at that time that one cannot always trust their eyes and that in order to represent the world 'as it is', it is necessary to use objective measurements [25].

Exactly the same observation lead to another paradigm shift at the beginning of the sixteenth century. Artists began to explore the way in which we perceive the world and their means of experimentation was painting. By and by, the principles of naturalism were discarded and replaced by new concepts with a stronger focus on inner than outer realities. In general one can say that the link between the physical world and pictoral elements was broken and artists became therefore free to explore new associations. The expressionist movement for instance experimented with ways of expressing moods and emotions (see Figure 2.12b). One means of achieving this was through a novel use of colour that was motivated by the feelings we attach to different hues and tones. Dark colours for instance can express loneliness or depression whereas bright colours suggest ease and happiness. The main point is that



a b

Figure 2.12: a) Dürer, from *Unterweisung der Messung*, 1525 b) Munch, *Death in the Sickroom*, 1893

we do not need to learn this association, we can feel it and that therefore, by exploring these connections, we can learn something about ourselves. Apart from expressionism, there were many other movements with different goals (e.g. impressionism, cubism or surrealism) which we cannot treat in this context, but the important lesson is that once the strict rules of naturalism were overcome, a whole new world opened itself to artists. Although many people still consider naturalistic paintings as the only real form of art, at the latest the advent of photography has rendered the quest for naturalistic depiction uninteresting for most artists.

If we now consider the development of computer graphics from its humble beginnings in the early 1960s to today, the development looks strangely familiar. Early computer games for instance used purely symbolic means of representing the game world, although not because this was the optimal solution, but mainly because of the limititations of the graphics hardware (still the addictiveness of games like Space Invaders proves that more abstract depiction methods can strengthen immersion). With the advent of more powerful hardware, began the era of 3D graphics that, until a decade ago, focused exclusively on photorealism. This development was driven by the desire to create images and virtual worlds that are indistinguishable from reality, but similar to the naturalist movement it failed to acknowledge the influence of mental processes on perception. Finally in the earlier 1990s, the non-photorealistic movement emerged as a reaction to photorealism. So far it has mainly been concerned with simulating traditional drawing and painting techniques, although some projects have also aimed at improving visualisation based on findings from cognitive psychology [26]. This intermediary step

is probably necessary, because the field is still young and somewhat chaotic (this can be illustrated by the fact that until recently there existed no uniting theoretical framework for NPR techniques [18]), the ultimate goal of computer graphics should neither be the simulation of reality nor the replication of depiction methods from other media, but rather the development of rendering methods that draw on the advantages of digital processing. Just as any other medium, computer depiction is limited, but it also offers opportunities that are unique to this medium. For instance one property that is unique to digital processing is the *reversibility of operations*. In all other known media, every action leaves a trace that can not be reversed completely (in painting for example, covering an area with a new layer of paint results in an increase in thickness in this area). On the other hand, changing the surface colour of an object in a 3D graphics package effectively erases all traces of the previous colour (at least in the rendered image). Furthermore the capacity of modern storage devices makes it possible to record all operations that were performed on a scene which allows users to revise earlier decisions and results in a 'delinearisation' of the creative process. The success of such *history functions* proves that it is worthwhile to explore the unique possibilities of the digital medium (see Section 5.2.2 for a discussion of this topic).

So far we have shown that there are many reasons for exploring non-photorealistic rendering techniques. The rest of this section is devoted to an overview of the large body of work that already exists in this area.

2.1.5 Categorisation

In the last decade a wealth of new NPR techniques have been developed using a variety of different approaches. In order to give an overview of the field, we will present a classification of NPR methods that is based on the taxonomy developed by Teece [57]. Note that the classification is entirely based on the technology that is used and does not include the intent of the resulting images (which is in most cases only vaguely defined).

- **Input data:** The kind of input data that a stylisation method operates on determines what kind of effects are possible. One can roughly distinguish between 2D, 2.5D and 3D algorithms. Historically *2D algorithms* were the first to be explored, long before the term non-photorealistic rendering was coined. As they operate only on 2D images, the range of effects that can be achieved is rather limited. 2D methods include 'artistitic filters' that are offered by many image-processing software packages and painting tools that simulate natural media. 2.5D methods represent an extension to the simple image-based approach and

work on the basis of augmented images that include, apart from the colour value, depth information or IDs that identify different areas of an image. This extra information can be used to stylise images automatically or to support the artist with more 'intelligent' tools (see the Example of Piranesi below). The most powerful NPR techniques work directly on 3D models. As they have complete knowledge of a 3D scene including lighting conditions, camera setup and the object topology, there are many properties that the stylisation process can be based on.

- **Performance:** Maybe the most important property of a rendering technique is the time it takes to render a single frame (at least in the context of this work). In a real-time application, one must make sure that all computations necessary to produce an output image can be performed in a very short time (depending on the kind of application rendering should take no more than 10-40ms). Due to these requirements, real-time implementations of rendering methods often make extensive use of the graphics hardware in order to unburden the CPU and speed up computations. Despite all optimisation efforts, it is normally necessary to make concessions regarding image quality in real-time applications. Offline rendering methods on the other hand are less affected by performance requirements. Ideally they should be able to render a draft-quality image at interactive rates, but as they focus more on high-quality output, computation times for production-quality renderings can be of the order of minutes or hours for highly complex scenes. Unfortunately the term 'real-time' is used in a very loose sense in computer graphics literature and can mean anything from a few frames per seconds to a few seconds per frame. Therefore many NPR techniques that claim to run at interactive rates fail the 'real-world' test if they are used in time-critical applications such as computer games.

- **Interactivity:** NPR techniques can also be distinguished by the amount of user interaction needed to produce an output image. Note that all rendering techniques rely on a certain amount of user input to specify surface properties, but this only needs to be performed once for each model as a preprocessing step. Given these properties and the viewing parameters, an automatic rendering method can produce an output image without any further user intervention. Interactive methods on the other hand usually only support the user in creating an image by offering 'intelligent' tools or by automating part of the workflow, but they leave all important artistic decisions to the user.

- **Output:** The final discriminating feature of NPR techniques is the

kind of output that is created. Here we usually distinguish between still images and animations. Although one might argue that still images are only a special case of animations, we will see later that their respective requirements can be very different. For instance temporal coherence plays a very important role in animation (see 2.3 for details).

Although the different axes are orthogonal and theoretically any combination of properties is valid, practical reasons limits the number of useful combinations. The rendering techniques we are going to present in the next chapters all fall into the category of automatic real-time 3D techniques for animation. We will now give some examples of applications that use NPR techniques and that belong to different categories.

2.1.6 Applications

To show that non-photorealistic rendering methods are not only of interest for research purposes and have already found their way into the 'practical' world, we will now present three applications from different domains that have adopted NPR techniques to create a distinct visual style.

Piranesi

Piranesi is a 3D painting tool developed by Informatix [7] that aims at producing high-quality still images for architectural presentations. It combines photorealistic effects with artistic drawing methods, thus generating images that are more expressive than the usual radiosity images, especially during early production stages, where many details have not been fixed yet.

Piranesi offers a highly interactive workflow, supporting the user whenever possible, but leaving all aesthetic choices to the artist. The general workflow can be broken down into the following stages:

- **Modelling:** 3D models of a scene are created using standard 3D software for architectural modelling such as 3D Studio Viz or MicroStation complete with camera definitions, surface materials and lighting information.

- **Augmented 2D image:** The next step consists in generating augmented 2D images from the 3D model. These augmented images are similar to the reference images described in 2.2.2 and encode additional 3D information such as per-pixel depth values or material IDs. This information is used in the subsequent stages to support the artist in painting over the raw image.

a b c

Figure 2.13: Examples of images created with Piranesi. a) Halftoning b) Watercolour c) Charcoal

- **Texturing and Cut-Outs:** The artist can now paint over the augmented image using a variety of 2D tools such as paint brushes or a fill tool. The main difference to pure 2D painting packages such as Photoshop is that the additional 3D data generated in the previous stage is used to support painting operations as described below. Furthermore typical props such as trees or people can be placed in the scene.
- **Stylisation:** Finally a wide range of stylisation methods can be applied to the augmented image ranging from simple outlines to watercolour or halftoning effects.

Figure 2.13 illustrates the variety of different styles that can be produced with Piranesi. We will now explain in more detail some of the most interesting aspects of the system.

If the painting tools are used on their own, they work in exactly the same way as in conventional painting software. But a number of special drawing modes make it much easier to colourise surfaces. *Locking* can be used to restrain painting to either a single object, all surfaces of a given orientation or all surfaces that share the same material ID. In combination with the *perspective texturing* tool that fills surfaces with a chosen texture and scales it locally to account for perspective foreshortening, one can for instance quickly add a brick texture to the outer walls of a building. Conceptually, one can compare this approach to adjusting and exchanging the surface texture of an object on-the-fly without the need to recalculate the whole scene thus enabling the artist to try out different textures with instant feedback.

Cut-outs can be compared to billboards in that they are flat images that always face the viewer. Using cut-outs, typical architectural props such as trees, lamp-posts and people can be easily added to the 2D image. Because

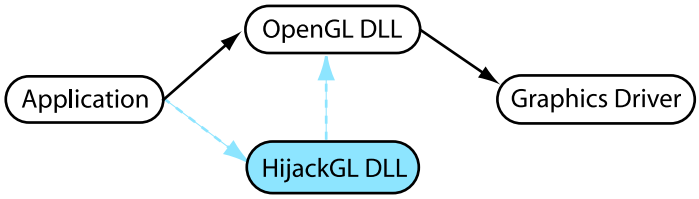


Figure 2.14: Normally an application sends graphics commands directly to the OpenGL DLL (black arrows). HijackGL intercepts these commands, reinterprets them and sends the new commands to the OpenGL DLL (blue arrows)

Piranesi still has access to depth information of the image, it can automatically scale cut-outs to match the depth at which they are placed and clip them against nearer image parts. Furthermore cut-outs can even create visually correct reflections and shadows on the surrounding environment, thereby fusing seamlessly with the scene.

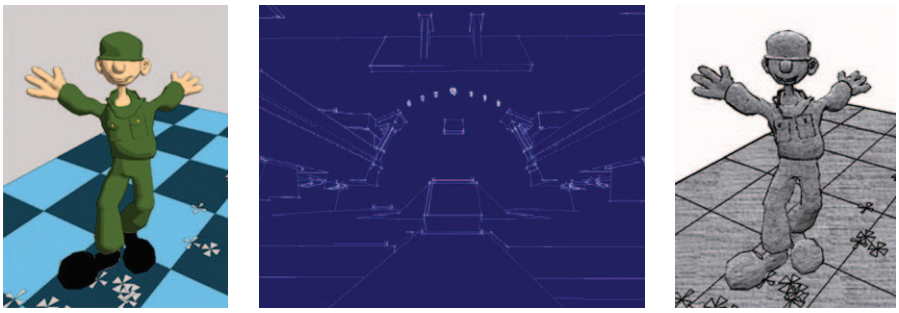
After the image has been textured and populated with cut-outs, *stylisation brushes* can be used to apply a wide range of drawing styles to either the whole image or regions of it. These brushes also make use of the partial 3D information that is still available to adjust stroke width or extract outlines. The palette of styles includes painterly techniques such as watercolour, oil and pointillism and drawing techniques such as hatching and pencil sketches. If they are applied wisely, these stylisation tools can create unique-looking images with little extra effort as can be seen in Figure 2.13.

The example of Piranesi shows that even very simple NPR techniques can be used to produce very effective images and that by combining automation with user interaction one can create intuitive tools that support the creative process.

HijackGL

HijackGL is a project developed by Alex Mohr and Andre Gleicher that aims at changing the visual style of existing 3D applications in a non-invasive way [43]. This is accomplished by intercepting the OpenGL command stream and changing it so as to produce a different output image using custom non-photorealistic rendering modules. The main components of the system are:

Intercepting OpenGL: As can be seen in Figure 2.14, the main interface between an application that generates 3D images and the graphics driver is located in the OpenGL dynamic load library (short DLL). By replacing the original DLL with a custom library that replicates the OpenGL interface it becomes possible to capture the command stream that an application



a b c

Figure 2.15: Examples of different HijackGL Rendering modules a) Cartoon Shading b) Blueprint Style c) Pencil

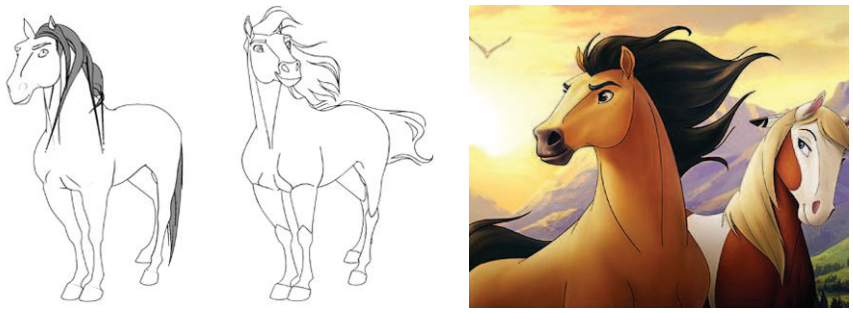
sends to the graphics hardware. This command stream can then be processed and sent to the original OpenGL library to generate an output image. As the custom library offers exactly the same interface to the application, no source-code modifications are necessary to make it work with HijackGL.

Reconstructing Geometry: Like most low-level graphics APIs, OpenGL does not explicitly represent geometric objects. The command stream only tells the graphics hardware what it needs to do in order to draw the desired object (programmatic or imperative approach), but it does not contain any high-level information such as connectivity data or object boundaries. Although some NPR methods can be implemented by simply remapping individual OpenGL commands, most rely on a declarative representation of the 3D scene. Therefore HijackGL first captures the raw command stream and then tries to reconstruct the high-level description of objects based on the declarative information inherent in the stream and heuristics. This results in a complete geometric model for each object that includes connectivity and silhouette information.

Rendering: HijackGL uses a plug-in architecture that allows new rendering modules to be easily added. Although it was mainly developed to quickly try out new rendering methods using existing applications, its possibilities are much more widespread:

- Wireframe Renderer: This rendering module simply draws all objects in wireframe mode with depth-cueing. Although its artistic value is rather doubtful, it can be very useful for analysing geometric algorithms such as subdivision schemes.

- Cartoon Renderer: A cartoon shader with outlines can produce results similar to those we will present in Section 3.1 (see Figure 2.15a).



a b c

Figure 2.16: a) CG Reference Image b) 2D Hand-drawn animation frame c) Still from the final movie

- Blueprint Renderer: This module produces images reminiscent of architectural blueprints by drawing white silhouettes and sharp edges on a blue background and extending the lines a bit beyond their original endpoints (see Figure 2.15b). Furthermore some lines are tagged with pseudo-measurements and dimension lines.

- Non-Graphical Modules: Of course an output module is not forced to redirect its output to the original OpenGL library. Alternative uses include writing the command stream to a text file for performance analysis and debugging, distributing the output to a cluster of machines that control different displays (e.g. for a tiled-display CAVE) or capturing the 3D models in a file that can be processed by software tools like 3D Studio Max.

Spirit: Stallion of the Cimarron

The example of Dreamworks' animated film *Spirit* illustrates nicely how NPR techniques can be incorporated into the traditional cel animation process to cut down costs and produce high-quality movies [15]. The story of Spirit revolves around a wild mustang stallion roaming the untamed American frontier and its first encounter with mankind. Set in the 'Old West', it features many scenes composed of large herds of horses that are time-consuming to create by hand, as well as close-up shots where precise control over every detail is required. We will now briefly describe how 3D models and toon-shaded imagery were used in various stages of production to create a compelling experience.

During preproduction, CGI (computer-generated imagery) was used extensively to plan camera movement and explore the effect of different shoot

lengths and angles. Once the desired shots were established, the CGI keyframes were printed on animation paper as outline drawings and served as a scale and perspective reference to the animators (see Figure 2.16ab).

All close-ups of the main character were completely drawn by hand to achieve the highest possible image quality (for instance hand-drawn shading is often guided more by aesthetic considerations than physical correctness) and to retain more control over facial expression which is still more tedious to animate in 3D than in 2D. Medium to long shots and complex camera moves, where the overall composition is more important than individual elements, were completely created in 3D and rendered to film using cartoon shading and outlining algorithms. This dual approach was even pushed as far as switching back and forth between 2D and 3D animation during a single shot, which required much tweaking to hide the transitions by adjusting the width of CG outlines and matching the hand-drawn subjective shading to the physically correct CG cartoon shading. Furthermore most of the landscape was first generated in 3D and then only retouched to match the hand-drawn style of the rest of the scene.

2.2 Graphics Programming

In this section we are first going to give a brief introduction to the OpenGL 3D graphics API and afterwards discuss special issues that arise when OpenGL is used to implement real-time non-photorealistic rendering techniques on consumer graphics hardware.

2.2.1 OpenGL

OpenGL is a 3D graphics API that was first introduced by SGI in 1992. Since then it has been refined and extended by a committee composed of researchers and representatives from major 3D graphics companies (called the Architecture Revision Board or ARB [4]). OpenGL implementations are available for all major (and even some obscure) operating systems. For this reason OpenGL is very popular with the open-source and scientific community whereas its main competitor, the DirectX API developed by Microsoft [3] dominates the gaming market.

OpenGL is a pure 3D graphics API, which means that it does not include an interface manager (for windows, dialogues, menus etc.). Furthermore it offers only the minimal set of features that are necessary to draw 3D primitives to a screen region and therefore lacks support for high-level primitives (e.g. spheres, cylinders etc.) and scene management. There exist a num-

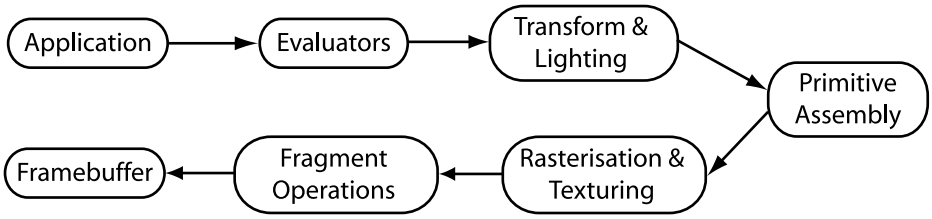


Figure 2.17: The OpenGL Rendering Pipeline

ber of toolkits based on OpenGL that add these missing features (e.g. GLU [5] for high-level primitives and GLUT [6] for platform-independent window management).

The basic concept of OpenGL is that of a state machine with many state variables [61]. Each state variable controls one aspect of the rendering process (e.g. surface colour or light position) and can be manipulated using accessor functions. Conceptually rendering an object can be split into three stages. First the scene is prepared by setting the camera, light sources and general properties. Then the material properties and transformation matrices for the object are defined. Now that the state machine has been configured, primitives that describe the object surface (e.g. triangles with normal vectors) are sent through the rendering pipeline (that is controlled by the state machine) and are transformed to pixels on the screen. This rendering pipeline consists of a number of different stages that can be considered as black boxes taking certain inputs and providing well-defined output values (see Figure 2.17). In standard OpenGL, the behaviour of these stages can only be influenced by the state machine, but it cannot be completely replaced, therefore it is also called a fixed-function pipeline. We will now first give an overview of the stages of the standard OpenGL pipeline and then present some extensions to the standard that allow a programmer to completely replace some stages with a custom implementation.

The OpenGL rendering pipeline

Conceptually the rendering pipeline can be broken down into the following stages:

- **Evaluators:** As the subsequent stages rely on vertex data, parametric surfaces and curves (which are described by control points and polynomial basis functions) first need to be converted to vertices with the help of evaluators.

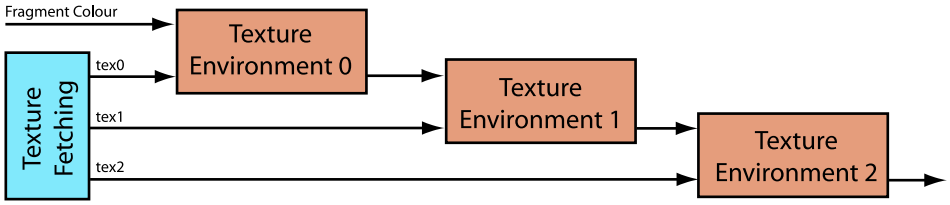


Figure 2.18: Standard OpenGL texture combination using texture environments

- **Transform & Lighting:** In this stage, per-vertex calculations are performed. These include transforming the vertex data from the local object space to a number of different coordinate spaces (e.g. to eye-space for lighting and clip-space for rasterisation). If hardware lighting is enabled, the Gouraud shading model is evaluated at each vertex, taking into account the light sources and material properties. The vertex data is then interpolated in the subsequent stages.

- **Primitive Assembly:** The next rendering step consists in clipping the line or triangle primitives against the view volume, culling faces that are not visible at all and applying perspective division that is necessary to correctly render foreshortening effects (only if a perspective camera is used).

- **Rasterisation & Texturing:** After the viewport positions of the vertices have been calculated, rasterisation determines colour values for all fragments that lie between them (i.e. for inner points in the case of triangles). This includes looking up colour values in texture maps and combining them with the interpolated values from the per-vertex lighting calculations.

- **Fragment Operations:** Finally a number of filters can be applied to fragments. The most important of these are the depth test that discards fragments based on their depth value and the alpha test that is based on the alpha value of the fragment colour.

In the next paragraphs we will describe some extensions to the OpenGL standard that have proved useful for efficiently implementing NPR methods.

Multitexturing

Although multitexturing has been included in the OpenGL 1.2 standard, it is still often considered as an extension because many OpenGL implementations

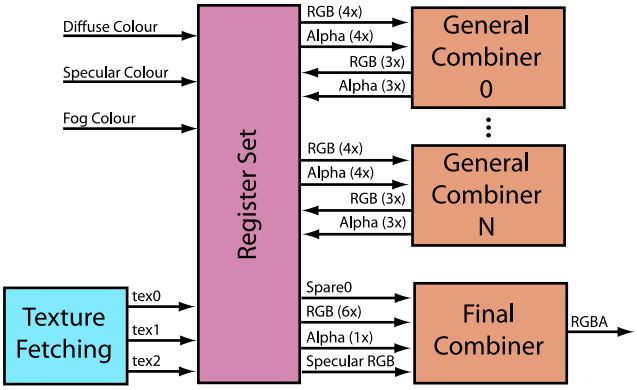


Figure 2.19: Overview of the Register Combiner Architecture

are still based on the 1.0 specification. Multitexturing allows more than one texture map to be applied to primitives in a single rendering pass by providing a number of texture units [61]. Each unit is bound to one texture map and has its own transformation matrix, mapping properties and texture coordinates. If no further extensions are used, the colour values from the different units are combined sequentially as shown in Figure 2.18. In this model, the programmer can only define which blending function is applied to the results of two subsequent stages. Although this is sufficient for many applications, some cases require a more flexible model such as the register combiner model we present next.

Register Combiner

The register combiner extension was introduced by Nvidia when it became apparent that the standard texture combination model was very inefficient for many advanced rendering techniques that need to access the same texture map (e.g. a normal map) multiple times in a single pass [56]. They therefore replaced the fixed input mapping for each combination stage with a flexible model where a number of so-called general combiners with custom input and output mappings perform configurable calculations on the input values and provide the result to the next combiner (see Figure 2.19). A special final combiner with a different internal setup puts together the results from the general combiners. As can be seen in Figure 2.20, a general combiner first remaps the input values, combines them in different ways, applies a scale and bias operation to the results and writes them to the chosen output registers. Furthermore the `RGB` and `Alpha` portions are treated separately. Although register combiners are very powerful, they are not intuitive to use. It is of-

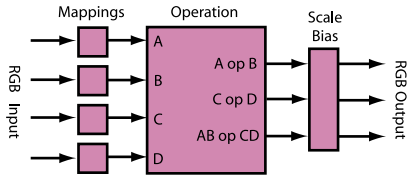


Figure 2.20: Internal structure of a general combiner (RGB portion only)

ten difficult to derive a combiner setup from a given shading model as the combiners cannot be freely programmed and the developer can only choose from a number of fixed configurations. This is mainly due to the fact that the register combiner model was initially tailored to suit some special applications such as per-pixel bump mapping. Lately, hardware manufacturers have become conscious of these problems and are now developing general fragment processors that work similar to the vertex programs we will treat next.

Vertex Programs

As we have seen in Figure 2.17, the standard OpenGL rendering pipeline contains a stage that transforms vertices and computes per-vertex lighting information. This module is highly optimised for the Gouraud shading model, and one has only a very limited influence on calculations by setting the transformation matrices and light and material properties, thus making if very difficult to implement alternative shading models on the graphics hardware. Nvidia remedied this by introducing the Vertex Program Extension [63] that completely replaces the standard Transform and Lighting stage by a user-defined assembler program that specifies how vertices are transformed and lit. Vertex Programs are based on a custom SIMD assembler instruction set that operates on 4-component vectors and has been designed to cover vector transformations and lighting calculations. Currently the assembler language is still rather limited as it does not support conditional branches or loops (although this is going to change in the near future). A vertex program takes a number of input vectors (e.g. vertex position, normal vector, transformation matrices etc.) and outputs the clip-space vertex position. Optionally it can also provide a shading colour and texture coordinates to the subsequent stages of the rendering pipeline. Custom input values can be provided to the vertex program through the OpenGL API on a per-vertex or per-object basis. Note that a vertex program always operates on a single vertex (so it does not have access to connectivity information) and cannot create or

delete vertices. Some applications for vertex programs include preprocessing for hardware bump-mapping [62], mesh skinning and custom texture coordinate generation. In Appendix A you find the source code of vertex programs that are used by the non-photorealistic rendering algorithms we will present in the next chapters.

Occlusion Culling

One of the main drawbacks of OpenGL is that the rendering system does not provide much feedback to the application. For instance it is not possible to query the graphics hardware whether an object will be visible under the current viewing conditions. At least for visibility determination, the problem is remedied by the Occlusion Culling Extension [50]. It basically allows an application to send primitives down the rendering pipeline and afterwards read back how many fragments were drawn (or would be drawn if writing is disabled) into the framebuffer. One obvious use for this extension consists in first drawing the bounding box of a complex object without writing to the framebuffer and with the occlusion culling test enabled. If no fragments of the bounding box have reached the end of the rendering pipeline then the complex object does not need to be drawn because it is not visible, thus saving a lot of rendering time. Unfortunately the first version of the extension (developed by HP) suffered from performance issues. As the hardware could one handle one test at a time, one had to submit a test and wait for the result before proceeding to drawing the next object. In the new version, Nvidia switched to an asynchronous model where a new query can be submitted without having to wait for the previous query to be finished. One would therefore first submit queries for all the bounding boxes in a scene and then read back the results in the order in which they were submitted. This approach only adds the overhead caused by processing the bounding boxes and thus saves a lot of rendering time for scenes composed of highly complex objects. As we will see in Section 4.4.4, the occlusion culling extensions can also be used to efficiently solve the visibility problem for outlines.

2.2.2 OpenGL and Realtime NPR

We will now present some of the basic techniques that many NPR algorithms rely on and investigate how well they are supported by the OpenGL architecture. This list is not meant to be comprehensive, but serves only to provide a feeling of what is currently possible using consumer graphics hardware. Before we get started, one general issue that is specific to OpenGL is worth mentioning. The fact that all changes to the OpenGL architecture

need to be approved by the ARB, has spawned a large number of unofficial vendor-specific extensions to the standard that are only supported by a limited number of graphics boards. In this situation, the decision whether to rely on certain extensions or to use only standard OpenGL features is very delicate. As we will see later, non-photorealistic rendering techniques rely heavily on extensions as the standard OpenGL feature set has been designed with photorealistic and more specifically Gouraud Shading in mind. Unfortunately this limits the number of platforms on which a given algorithm will run properly.

Non-Linear Shading

Many NPR shading techniques work by replacing the linear diffuse lighting term in the Phong lighting model by a non-linear term, i.e. by $f(NL)$ where $f(x)$ is a non-linear function. The general approach consists of supplying the NL term as texture coordinates and storing the non-linear function in a texture map. This works because $f(x)$ usually only needs to be defined for a fixed interval. In the simple case of a lighting function that takes only one argument, the function values can be stored in a 1D texture map, but the approach can also be extended to more complex functions with two or three arguments. How the texture coordinates are calculated depends on the kind of expression that needs to be evaluated at each vertex. In some very simple cases, setting a special texture transformation matrix does the job (see Section 3.1.2), but more often it is necessary to perform the calculations in a vertex program. In conclusion it can be said that non-linear shading can be implemented very efficiently, relying only on a few OpenGL extensions. It comes therefore as no surprise that many real-time NPR algorithms rely in some way on non-linear shading functions. We will present some examples of such algorithms in Chapter 3.

Image-Space Techniques

The general idea of image-space techniques consists in rendering special images of a scene, where the `RGBA` colour values can encode any kind of information, ranging from IDs to 3D spatial vectors [52]. As a second step image-processing operators can be applied to these images or they can be blended with other images. Finally all the different layers are composited to yield the output image. Unfortunately it is currently not possible to efficiently implement these algorithms using OpenGL because extracting the framebuffer is very slow and therefore makes it impossible to achieve interactive framerates, no matter how simple the rendered scene is. Even though

there exist extensions that allow image-processing operators to be applied to the framebuffer as a postprocessing step, these operators are very slow. This is bound to change with the realease of the next generation of 3D graphics APIs (DirectX 9 [3] and OpenGL 2.0 [4]). They will add two new features that will make it possible to implement the method we described above in hardware as a two-pass rendering algorithm. First a number of different special images can be created in the first pass by drawing to different virtual frame buffers (i.e. texture maps that reside in the graphics board memory). In a second pass, a pixel program processes these buffers and combines the results into a single output image that is simply mapped as a 2D image into the framebuffer. It will take some time before these features are widespread enough to allow their use in consumer software, but some preliminary demos have already been released illustrating the effects that will become possible [42].

Particle Rendering

Some of the most interesting NPR techniques are based on particles (see [41] for an example of painterly rendering or [16] for a flexible particle-based rendering system). A large number of particles are placed on the object surface and rendered using either 2D (such as texture splats) or 3D primitives including simple billboards and complex polygons. Currently the global and local properties of particles are calculated in software before the geometry is submitted to the rendering hardware. Global properties include the number of particles that are drawn for a given viewpoint, for instance fewer particles are rendered for an object located at a far distance from the camera. Local properties depend on the chosen primitive and range from base colour to orientation and function (e.g. outline or shading). Unfortunately the properties are often determined using the reference images that we introduced above and are therefore affected by the same drawbacks. Furthermore many techniques require a lot of small particles to be placed and rendered for each frame, which can be very time-consuming. Unless most properties of a particle can be calculated on the fly on the graphics board, it will not be possible to speed-up these methods enough to allow them to be used in real-time applications.

2.3 Character Animation

In this section we are going to present a brief introduction to character animation from the point of view of our work. We are therefore not going to

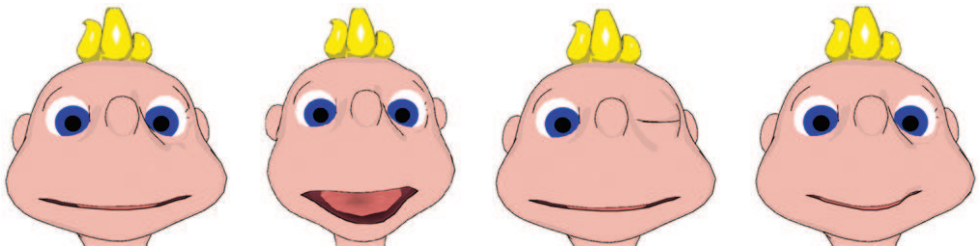
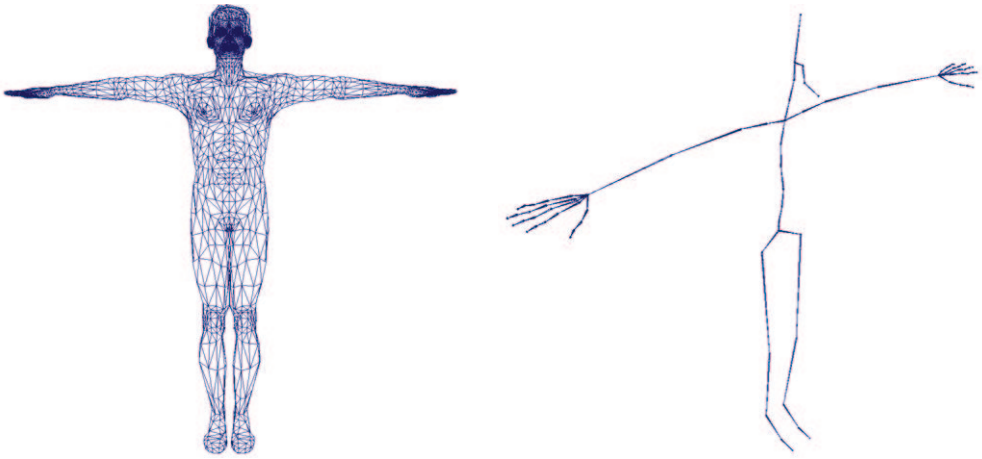


Figure 2.21: Examples of elementary facial expressions for morphing a) neutral b) mouth wide open c) left eye closed d) thin mouth

consider how the animation cycles are created, but only describe basic animation techniques for real-time applications and discuss the implications for rendering.

2.3.1 Character Animation Techniques

Technically one can distinguish between three animation techniques for character animation. *Keyframe animation* works on a per-object basis and can be used to apply a transformation to a whole character (e.g. translation of the character when it performs a jump) or a part of it modelled as an individual mesh (e.g. a hat). This simple transformation can be performed completely in hardware by setting appropriate transformation matrices. Especially for facial animation a different approach is necessary, because facial expressions are based on deformation. The basic idea of *morphing* is to first create a 'construction kit' for facial expressions that consists of models of an identical structure that represent the basic deformations that can be applied to a face such as 'open mouth' or 'raised eyebrow' (see Figure 2.21). A complex facial expression can then be constructed from this kit by interpolating the different models. This animation technique is very fast, because it only requires linearly interpolating vertex data. Finally *skeletal animation* can conceptually be considered as a mix of the two previous methods. First a hierarchical skeletal is transformed using keyframe animation and then the character mesh is deformed based on this skeleton. This has the advantage that complex animations can be created with relatively little effort, because once the skeleton has been 'connected' to the model (this preprocessing step is also called skinning), it is only necessary to manipulate the skeleton (which is much less complex than the mesh) to create believable body deformations. From the computational standpoint, skeletal animation is more expensive



a b

Figure 2.22: a) Wireframe rendering of a virtual character b) The corresponding skeleton (perspective view)

because it requires transforming each vertex and normal vector by a number of matrices and interpolating the resulting vectors.

2.3.2 Rendering Issues

After we have seen which transformations and deformations are applied to a virtual character, we will now discuss the implications for the subsequent rendering step.

- **Performance:** Morphing and skeletal deformation require transforming and interpolating a number of meshes for each frame of an animation. Although it would be possible to delegate some of these calculations to the graphics hardware (e.g. by using vertex programs), this is usually not done because it might interfere with rendering. If for instance vertex blending is performed by a vertex program, the rendering algorithm must either refrain from using vertex programs for its calculations or perform the blending operation itself because only a single vertex program can be active at any time. In the first case we would unnecessarily limit the possibilities of the rendering algorithm and in the second case, the conceptual separation between animation and rendering would be destroyed and any changes in the animation code would require updating all rendering modules. Because all animation computations must be performed in software, it means that

the rendering code must be highly optimised to guarantee interactive frame rates (ca. 25-30fps).

- **Artifacts:** Skeletal animation can cause parts of the mesh to interpenetrate if joints are bent at sharp angles (see Figure 2.23). A rendering algorithm for character animation should be able to handle this situation without introducing disturbing artifacts.

- **Tesselation:** Figure 2.22a shows a typical mesh used for character animation. As one can see, the degree of tesselation varies considerably over the model. Regions where precise control over deformation is required (such as the eye and mouth regions) are subdivided more finely than regions that are less important or less affected by deformation (such as the chest). Therefore rendering algorithms must be able to cope with uneven tesselation. As we will see in Chapter 4 this can pose a problem to some rendering methods.

- **Temporal Coherence:** Another issue of animation in general is temporal coherence. If some features of a rendered image are chosen randomly (e.g. stroke placement for hatching), they will look different in each frame. Although this may be desirable in some cases, it will in general distract the beholders attention and can put a considerable strain on the eyes. Similar artifacts appear if the rendering algorithm is unstable with respect to small changes of the viewing angle or small body deformations. It is therefore important to ensure that small (or no) changes in the scene result only in small changes in the rendered image.

- **Abstraction:** Findings from psychology show that we are able to identify human motion from very few cues [24]. If for instance subjects are shown a movie of a human motion cycle that is only represented by a small number of 'light dots' that are attached to the moving character, they are still able to determine the gender of the moving person and the task it is performing. This shows that more abstract representations should not prevent the beholder from perceiving an animated character correctly. Although Hodgins et al. [32] showed that a more detailed representation of a virtual character can be helpful for expressing subtle bodily expressions, one can also turn this argument upside down and claim that because we tend to analyse the body language of an abstract character less exactly, we will be less confused by subtle imprecisions in animations (see also Section 2.1.3). This could be especially helpful

Figure 2.23: Interpenetrating limbs caused by strong deformation of the leg (marked in red)

Figure 2.24: Structure of the Trick17 Animation Engine

if the animation cycles are blended in real-time because in this case, the animator only has limited control over the final animation.

In the following chapters we will use some of the criteria we described above to evaluate the quality of the non-photorealistic rendering algorithms we are going to present.

2.4 Trick17

In order to test the non-photorealistic rendering algorithms we developed, we integrated them into the Trick17 animation system. In this section we are going to provide a brief overview of the system architecture, present the mechanism for adding new rendering modules and explain the workflow from model creation to real-time animations.

2.4.1 Overview

Trick17 is a real-time rendering and character animation system developed at the Laboratory for Mixed Realities [2]. It is largely platform-independent and so far there exist ports that run on Windows-based systems and under

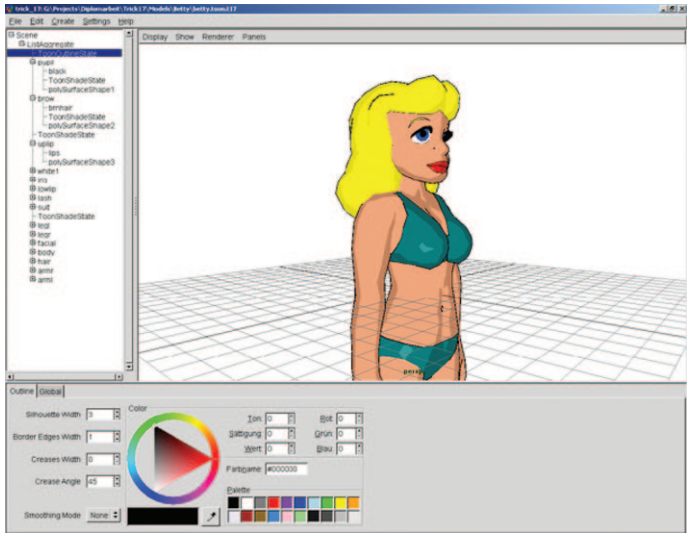


Figure 2.25: The user interface of the Trick17 Editor. Note the scene-graph on the left-hand side and the interface for manipulating a render state in the lower region.

the Linux operating system. As the project is still at an early development stage, we first present the general concept of the system and then describe its current state on which our work is based.

The animation engine used in the Trick17 system can be divided into three consecutive stages (see Figure 2.24). The input to the animation engine is the desired behaviour of the character that is determined by high-level processes. For instance the character could draw a gun while running and its face could show an expression of agressiveness mixed with fear. As it would be tedious and space-consuming to pre-compute individual animation cycles for every possible combination of actions and expressions, only base animation cycles such as 'running', 'walking' or 'draw gun' are created by an animator and stored in a database. The first stage in the animation pipeline now consists in choosing the appropriate animation cycles based on the desired behaviour. In the next stage they are then blended together to obtain a single realistic movement. Finally the resulting animation is used to deform the character mesh before it is submitted to the rendering engine.

Currently, the Trick17 system is composed of two applications, the Trick17 Editor and the Animation Engine. In the Trick17 Editor, the scene-graph of a virtual character can be manipulated (see Figure 2.25). This scene-graph is a tree structure that contains geometry nodes, light definitions, skeletal joints and render states that control the rendering system.

The standard render states correspond to the most common OpenGL features typically used for photorealistic rendering such as texturing, alpha blending and Gouraud Shading. The user can change the look of a character by manipulating these render states through he user interface. Note that the editor is not a modelling tool, in order to adjust the mesh of a body part, it is therefore necessary to resort to a modelling package.

Once a character has been prepared in the editor, it can be tested in the Animation Engine. The idea of the Animation Engine is to load a virtual character and a number of animation cycles that can then be played back and mixed in real-time. Unfortunately it is still in a very early development phase and supports only one standard skeleton and a small number of hardwired animation cycles. Because of these limitations, only virtual characters that are based on the standard skeleton can be animated. Note that the 'high-level process' stage described above has not yet been implemented and that animation selection is currently performed by the user via the keyboard.

2.4.2 Integrating new Rendering Modules

Integrating our non-photorealistic rendering modules was rather straightforward. Each new module consists of three parts: the OpenGL rendering code, a number of custom render states and an interface definition for each render state. The custom render states control the behaviour of our rendering modules. In order to experiment with the rendering modules in an intuitive way, we also added a user interface to each render state. Note that the properties exposed through the interface may not always correspond directly to the internal properties of the render state because we often found it more user-friendly to remap values between the two representations (e.g. angles and threshold values). Furthermore each rendering module uses only a relevant subset of all render states and ignores the rest. This allows users to specify material definitions for more than one renderer in the same scene file and switch between the different rendering styles interactively in order to compare the results.

Although we did not encounter any major problems while implementing our algorithms, a few general limitations of the rendering subsystem should be mentioned.

- **Multipass Algorithms:** Due to the structure of the rendering system, it is currently not possible to implement true multipass algorithms in Trick17. Although it is possible to render each object in multiple passes, this does not always yield the same results as performing first one rendering pass for all objects and then the next one for all of them.

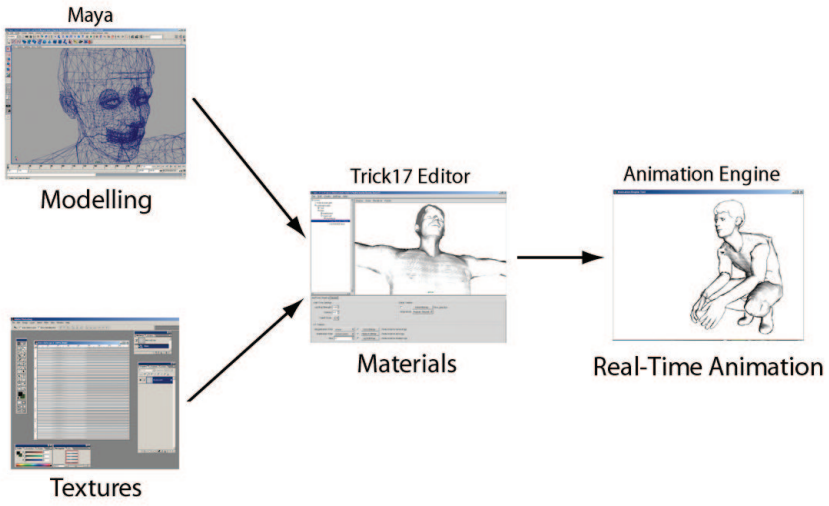


Figure 2.26: The workflow in the Trick17 system.

Some of the outline drawing algorithms for instance rely on a depth buffer of the whole scene to work correctly, which requires that first the depth buffer is filled with all objects (e.g. during a shading pass) and that only then the outlines are added. This is currently not possible which leads to rendering artifacts in some cases (see for instance Section 4.2.3)

- **Pre- and Post-Rendering Processing:** A rendering module can currently not apply any pre- or postprocessing to each frame without resorting to tricks. This limitation can be circumvented by adding a custom render state to the scene graph. If it is placed on top of the tree, it is treated before the rest of the graph and can be used to prepare the rendering system. Similarly, placing a custom state node at the end of the tree allows the rendering module to perform operations such as applying image-processing operators to the framebuffer. We used this mechanism to add a background image (preprocessing) and a structure texture (postprocessing) to the rendered images in order to simulate different media such as paper or canvas (see Appendix B).

2.4.3 Workflow

Figure 2.26 illustrates the workflow in the Trick17 system. First the triangle mesh of a character in modelled in the Maya software tool and attached to a

skeleton. Using a custom export plugin, the scene can then be exported to the proprietary Trick17 file format and imported into the Trick17 Editor. Here the material properties are specified by adding new render state nodes to the scene graph, and textures that were created in an image-processing package are applied. Once the desired visual style has been achieved, the scene can be saved and visualised using the Animation Engine. Note that due to the above-mentioned limitations of the Animation Engine, only characters based on the standard skeleton can be animated.

Chapter 3

Shading

a b c

Figure 3.1: Examples of artistic shading. a) Stippled Illustration of a terracotta statue b) Baldung Grien, *The Fall of Man*, Detail, 1511, Woodcut c) Henry Matisse, *Figure décorative sur fond ornemental*, Detail, 1925-26, Oil on canvas

In Section 2.1.2 we saw that shading constitutes one of the most important cues for deriving volume information from a 2D image, but the shading does not need to be physically correct, as long as the relation between shadowed and illuminated areas is maintained. Fine artists and illustrators have exploited this fact for a long time and developed a wide range of techniques for depicting shading in a stylised way (often imposed by limitations of their respective medium). These techniques include stippling, hatching and painting with a limited palette of tones (see Figure 3.1). In this chapter we are going to present three stylised shading techniques for real-time applications. The first shader uses only two to three different shades to create a visual style similar to traditional cel-animation. The basic method is not new, but we developed a very efficient OpenGL implementation and it constitutes the ba-

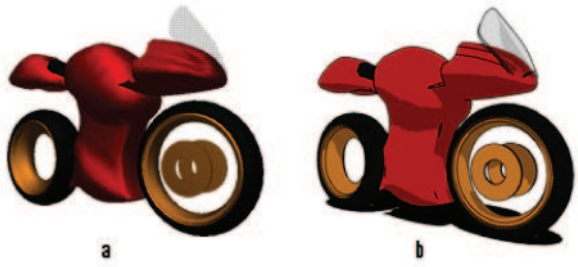


Figure 3.2: Comparison of Gouraud and cartoon shading

sis of the second shader (textured shading). The third shader is based on the halftone shader developed by Freudenberg [23], but we extended it to a real halftoning algorithm that works in a more intuitive manner. For each shader, we first present the underlying model and include an OpenGL implementation that we integrated into the Trick17 system. These are complemented by example images created with the shading modules and a discussion of the results with respect to technical and aesthetic criteria. Finally, we close each section with hints at how these techniques can be refined and extended.

3.1 Cartoon Shading

Very few cartoons are produced live
It puts a terrible strain on the animator's wrist
Matt Groening

Cartoon shading (also called hard shading or toon shading) ranges among the first stylised shading techniques that have been explored in the context of computer graphics (see for instance [36]). This is largely due to the fact that is simple to implement on current fixed-pipeline hardware, but also due to the its attractive visual style ('Everyone loves Cartoons').

3.1.1 Lighting Model

The first step in creating a new shading method is to formulate a lighting equation that establishes the interaction between the light sources in a scene and the surface properties of an object. As a starting point we use the Phong lighting model that most real-time photorealistic shading methods (e.g. Gouraud shading) are based on and introduce a few changes that result

in the desired effects. The (simplified) Phong lighting model can be expressed as

$$C = a_l a_m + (\max\{Ln, 0\} d_l d_m) \tag{3.1}$$

where C is the shading colour of a surface point, a_l and a_m are the ambient light resp. surface colours, d_l and d_m are the diffuse light resp. material colours, L is the light direction at the surface point and n is the surface normal [20]. We omitted the specular reflectance term because it is not relevant in this context. Note that without specular reflectance, the colour of surface point is completely determined by the light and surface properties, independent of the current viewpoint. As can be seen in Figure 3.2, the surface colour of a Phong-lit object (illuminated by white light) varies smoothly from the ambient material colour a_m to the sum of the ambient and diffuse material colours $a_m + d_m$. The cause of this smooth variation is the term $\max\{Ln, 0\}$ which varies from 0 for $\angle(L, n) >= 90°$ to 1 for $\angle(L, n) = 0°$.

If we now compare the two images in Figure 3.2, we can see that the only difference is that in the toon-shaded image the transition from ambient to diffuse colour is not smooth but hard. Up to a certain threshold $Ln < t_d$ the surface point is rendered using the ambient colour and for $Ln >= t_d$ the surface point is assigned the sum of ambient and diffuse colours. This can be formalised by replacing the max function by a threshold function $thresh$ defined as:

$$thresh(x, t) = \begin{cases} 0 & : \quad x < t \\ 1 & : \quad x >= t \end{cases} \tag{3.2}$$

This leads to a new lighting equation:

$$C = a_l a_m + (thresh(Ln, t_d) d_l d_m) \tag{3.3}$$

where t_d is the diffuse threshold ($-1 <= t_d <= 1$). This creates the desired effect of introducing a hard boundary between the shadowed and the lit surface regions.

The value of the diffuse threshold t_d determines the relative sizes of lit and shadowed regions. Small values of t_d result in large illuminated regions and small shadowed regions and vice versa for large values of t_d. If we set t_d to 1 or -1 only one colour is used and we get a flat-shaded effect. As the aim is not to achieve physically correct results, but rather to create aesthetically pleasing images, the threshold value should be chosen by the user as a material property.

Although we omitted the specular term in Equation 3.1, it is often desirable to add a third shading colour to simulate shiny surfaces like metal [34]. In the Phong lighting model a specular term

$$C_s = s_l s_m (vr)^\alpha \tag{3.4}$$

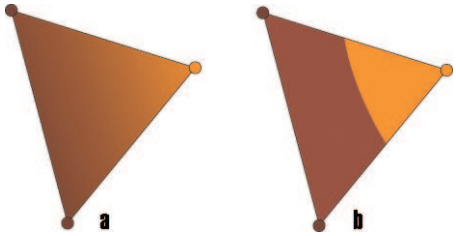


Figure 3.3: Vertex- vs. Pixel-Shading for Cartoon Shading. a) Vertex colours are interpolated across faces. b) The shading colour is calculated per-pixel.

is used to approximate reflectance in a physically correct way [20]. In this equation, s_l and s_m represent the specular light and material colours, v is the view vector, r is the direction of light reflected at the surface point and α is a shininess coefficient that determines the size of the highlight. Cartoon shading usually uses a much simpler method and replaces specular highlights by diffuse highlights. To achieve this, a second diffuse term with a higher threshold is added

$$C = a_l a_m + (thresh(Ln, t_d) d_l d_m) + (thresh(Ln, t_h) s_l s_m) \tag{3.5}$$

where t_s is the specular threshold with $-1 <= t_d < t_h <= 1$. In the context of cartoon shading, this rough approximation produces appealing results and is much easier to compute, as all calculations remain independent of the viewpoint.

3.1.2 Implementation

Based on the cartoon lighting model, we are now going to show how this model can be implemented in OpenGL. Before we get started, we first introduce a change in notation that facilitates discussion and is more intuitive. As we derived the cartoon lighting model from Phong lighting, the resulting surface colours are a_m, $a_m + d_m$ and $a_m + d_m + s_m$. From now on we are going to refer to these three sums as the shadow, diffuse and highlight colours to avoid confusion. This is closer to the way a designer thinks about colours and is also motivated by the implementation presented below. Furthermore we assume the light colour to be white, although coloured light sources could be easily handled. But in practice, coloured light would produce unwanted colour shifts in the final image, which is why artists prefer to change material colours manually to achieve the same results. Finally we are only going to consider the case of a single directional light source.

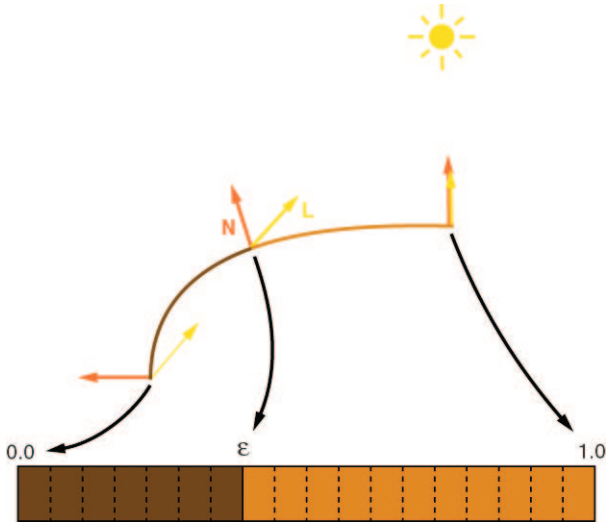


Figure 3.4: 1D Shading Texture. The lower entries contain the shadow colour and the upper entries the diffuse colour.

We first present a very simple method for implementing cartoon shading that does not produce the expected results to highlight the pitfalls of cartoon shading. This will serve as a basis for a more complex implementation that achieves higher image quality. The first method calculates the dot-product Ln at each vertex and assigns it a colour according to:

$$C = \begin{cases} C_s & : \quad Ln < t_d \\ C_d & : \quad t_d <= Ln <= t_h \\ C_h & : \quad t_h < Ln \end{cases} \tag{3.6}$$

where C_s, C_d and C_h are the shadow, diffuse and highlight colours respectively. One advantage of this method is that it can be easily implemented using any 3D API that supports vertex colours. On modern programmable hardware all computations could be executed in a vertex program, thus freeing CPU resources for other calculations. The major drawback of this algorithm is that is only produces correct results at the vertices of a triangle. As vertex colours are interpolated across the inner points of triangles, a smooth colour transition appears if not all vertices of a triangle are assigned the same colour (see Figure 3.3). Furthermore the size of this transition depends on the screen-space size of a triangle which is why artifacts are less noticeable on small triangles. As meshes for real-time character animation often exhibit large differences in tesselation, the resulting look is not very 'cartoony'.

In order to achieve a more pleasing hard transition, the decision which colour to assign to a surface point needs to be taken at each pixel that

is drawn. This insight leads directly to the second approach that uses a more complex setup and needs to employ some texture manipulation tricks because current fixed-pipeline hardware does not allow computations at the pixel level. The basic idea is to move only the thresholding step to the per-pixel level [36]. As the surface normals can safely be interpolated across a triangle and the light direction of a directional light is the same at every vertex, we can compute Ln for each vertex and interpolate the dot product across the surface of a triangle. The important step is then to evaluate the threshold function at every pixel. This can be achieved by using Ln as an index into a *1D shading texture* that stores the corresponding surface colour for every value of Ln (see Figure 3.4). To be precise, Ln is remapped to the interval $[0, 1]$ by using $l = \frac{(Ln+1)}{2}$ where l is the resulting texture coordinate.

So far the discussion was largely independent of the 3D graphics API. We will now show how to implement the algorithm using standard OpenGL and how to take advantage of extensions if available. The first step is the calculation of l. This step could either be accomplished in software and submitted as a 1D texture coordinate or we can take advantage of the texture matrix. Before texture coordinates are used to look up colour values in a texture map, they are transformed by a texture matrix similar to vertex transformations. We set the texture matrix to

$$M_t = M_d M_m \tag{3.7}$$

where M_m is the inverse-transpose modelview matrix needed to transform normals into eye-space and M_d is defined by

$$\begin{pmatrix} -L_x & -L_y & -L_z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \tag{3.8}$$

where $(L_x, L_y, L_z, 0)^T$ is the light direction in eye-space. If for each vertex we submit the vector $(n_x, n_y, n_z, 1)^T$ (where $(n_x, n_y, n_z)^T$ is the vertex normal) as texture coordinates, the transformed coordinates are

$$M_t \begin{pmatrix} n_x \\ n_y \\ n_z \\ 1 \end{pmatrix} = M_d \begin{pmatrix} n'_x \\ n'_y \\ n'_z \\ 1 \end{pmatrix} = \begin{pmatrix} -(L_x n'_x + L_y n'_y + L_z n'_z) + 1 \\ 0 \\ 0 \\ 2 \end{pmatrix} \tag{3.9}$$

where $(n'_x, n'_y, n'_z)^T$ is the vertex normal in eye-space. As texture coordinates are divided by the w-component (similar to the perspective divide applied to

vertices after transformation to screen-space), we finally get

$$\begin{pmatrix} \frac{-(L_x n'_x + L_y n'_y + L_z n'_z)+1}{2} \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.10}$$

It is necessary to invert the light vector as we need the direction from a vertex to the light source. If the OpenGL driver supports automatic texture coordinate generation using the vertex normal, we don't need to supply any texture coordinates as the driver will automatically submit the transformed normal vector to the texture engine. In this case, the texture matrix M_t is simply defined as $M_t = M_d$.

The next step is rather straightforward and consists in supplying a 1D shading texture. This texture encodes all material properties which we will briefly discuss now. In addition to the colours and threshold values introduced earlier, the size of the texture map S and the interpolation scheme influence the rendering in the following way:

- **Size:** High-resolution shading textures allow the artist to more precisely influence the exact location of the border between the different areas, but they contain also more redundancy (although this is not as much of an issue any more with modern graphics hardware)

- **Interpolation:** To obtain a very crisp border between the different shaded areas, the interpolation mode should be set to 'nearest neighbour'. For very high-resolution textures, using linear interpolation provides a cheap means of implementing antialiasing, but it also introduces artifacts in areas of low curvature. For a better antialiasing strategy see section 3.1.4.

On hardware that supports multitexturing, a 2D texture can be combined with the surface shading to add more detail. Depending on the desired effect, this detail texture can either be modulated by the shading colour or placed on top of the shading similar to a sticker. This second option is especially useful if a small black and white drawing (e.g. a logo or tattoo) is to be combined with coloured shading (in modulation mode, the texture colour is affected by the shading colour which is why one usually uses only grayscale shading to avoid colour shifts).

3.1.3 Results

Figure 3.6 shows some images rendered with our implementation of the above algorithm and Figure 3.5 presents the user-interface for manipulating mate-

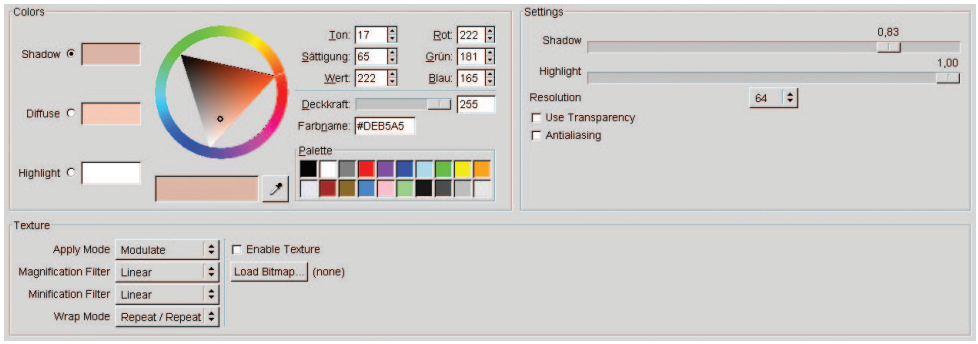


Figure 3.5: Cartoon Shading Material Interface in Trick17

rial properties in the Trick17 system. Before showing how this method can be extended, we briefly discuss the advantages and disadvantages of our implementation.

- **Performance:** As all calculations are performed in hardware using the standard fixed rendering pipeline without lighting, no performance penalty is involved. Even if a detail texture is used, the second texture fetch only adds negligible overhead.

- **Image Quality:** Although the rendered images are visually pleasing, two aspects could still be improved. First the boundary between differently shaded areas is composed of straight segments and lacks the nicely rounded look of hand-drawn images. This is due to the fact that each mesh face is flat and the effect is most noticeable in areas that are coarsely tesselated. The second kind of artifact is related to texture interpolation. Because the shading texture uses the 'nearest neighbour' scheme, the boundaries between shaded areas are very hard and especially at low screen resolutions, 'steps' become visible. These issues will be addressed in 3.1.4.

- **Variety:** Even this basic cartoon shading technique offers a lot of creative freedom to the artist, especially when shading is combined with detail textures. Many traditional cartoon-drawing styles can be simulated including flat-shading, two-tone shading and three-tone shading for shiny surfaces. One drawback is that only the colour of the detail texture is affected by shading whereas artists would like to have more influence on the look of the different areas. This issue will be addressed in 3.2.

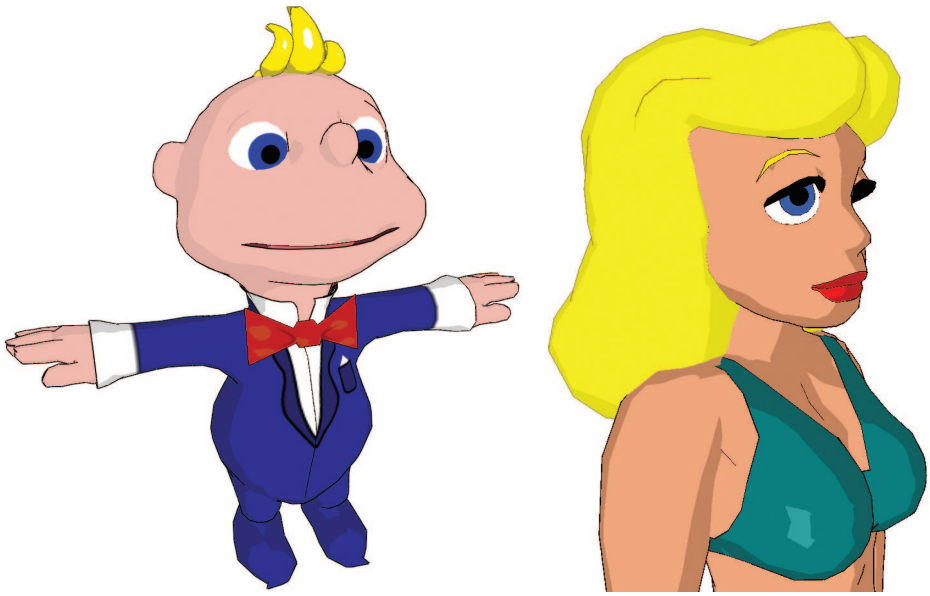
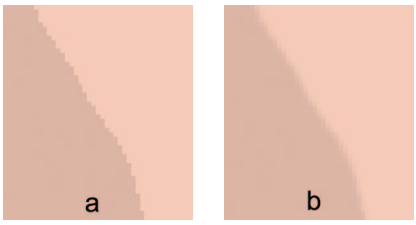


Figure 3.6: Examples of Cartoon Shading

3.1.4 Future Work

As we have seen above, a number of aspects of cartoon shading could still be enhanced. We will now briefly present possible extensions to the basic algorithm.

- **Antialiasing:** With the advent of programmable pixel processors, it becomes possible to apply image-processing operators to a rendered image. As described in [42], the different layers of an image are first rendered to texture maps. In a second step, these layers are processed



a b

Figure 3.7: Antialiasing for Cartoon Shading: a) Aliasing Artifacts b) Using Antialiasing (Simulation)

Figure 3.8: A black outline is used to delineate differently shaded areas.

by pixel-programs and composed into the final image. As a pixel-program has access to all texels, it is possible to apply a blurring operator with a small filter kernel to the shading texture resulting in a less pixilated look. Figure 3.7 simulates this effect with a blur filter in an image-processing toolkit. As all calculations are performed in screen-space, the artifacts associated with the earlier mentioned interpolation method are avoided.

- **Outlines:** The same image-processing method can be used to add a black outline between differently shaded areas, a technique that is often used by comic artists as can be seen in Figure 3.8. This effect is achieved by replacing the blurring filter by an edge detection filter as described in [42].

- **Threshold Maps:** With the basic cartoon shading method, it is difficult to refine the surface structure of an object. The obvious way is to subdivide the triangle mesh, which is often prohibited by performance limitations especially for complex scenes. Similar to the bump-mapping techniques developed for Gouraud shading, a normal map could be used to locally disturb surface normals and simulate more complex surface structures. Veryovka [58] has developed a method to achieve a similar effect by using a threshold texture to specify t_d and t_h at every surface point. Although it is not as powerful as bump-mapping, this method creates more surface detail and still maintains the stylised look of cartoon-shading.

- **Rounded Boundary:** Claes et al. [14] present a real-time method for creating nicely rounded shading boundaries by selectively subdividing triangles where not all vertices are assigned the same colour. This technique has the advantage that it selectively refines areas where

higher mesh detail makes a difference in the final image instead of using higher tesselation for the whole mesh. A similar effect can be achieved by storing the surface normals in a texture and calculating Ln on a per-pixel basis. If the resolution of the normal map is high enough, this will eliminate all artifacts caused by low tesselation without sacrificing any performance on graphics hardware that supports fragment shaders. For smoothly rounded objects, a normal map can be generated automatically from a high-resolution version of the low-resolution mesh (this technique has been successfully applied by id Software [1] in the computer game 'Doom 3'). Depending on the modelling technique used to create the low-resolution mesh, the high-resolution version can be derived without additional costs (e.g. NURBS or subdivision surfaces).

- **Specular highlights**: Although diffuse highlights are often sufficient to communicate the shininess of a surface, using real specular highlights could provide a better sense of curvature. Winnemoeller et al. [60] present a method involving 2D shading textures to achieve this goal. Although their algorithm runs at interactive frame rates, all lighting calculations are performed in software. Using a combination of vertex- and pixel-programs it would be possible to transfer these calculations to the graphics hardware to speed up rendering considerably.

3.2 Textured Shading

In the previous section we introduced an algorithm for creating cartoon-shaded images with little effort. This leaves room for visual enhancements. In this section we are going to extend the basic cartoon-shading algorithm to use two detail textures which offers more freedom to 3D artists. The underlying idea of this rendering method was first presented in [35]. In our implementation we added a more flexible blending scheme and we present ideas for further enhancements in the subsequent discussion.

3.2.1 Textured Shading

Comic artists often apply different styles to the regions of an object or character that face the light source than those that lie in the shadow. We are now going to develop an algorithm that achieves a similar effect for 3D cartoon images.

The basic idea is to use two detail textures, a *shadow texture* that is applied to surface points that lie in the shadowed regions of an object and a

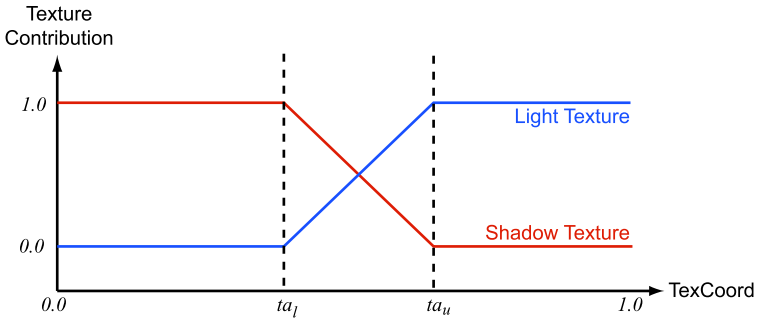


Figure 3.9: Detail Texture Composition in Textured Shading. The x-axis represents the illumination value encoded as a 1D texture coordinate (see previous section). The y-axis shows the contribution of the light and shadow textures depending on the illumination value with higher values representing a larger contribution.

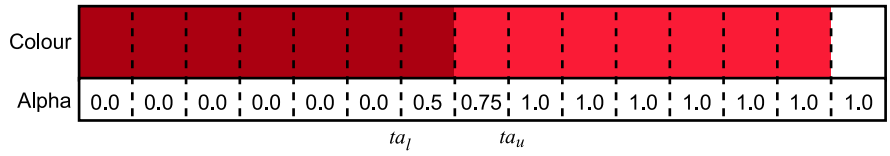


Figure 3.10: 1D Shading Texture with Alpha Channel for Textured Shading

light texture that is applied to points illuminated by the light source. Around the boundary, dividing shadowed and lit areas a mix of both textures is used to avoid too sudden transitions between both textures (see Figure 3.9). The size of this transitional area can be adjusted by the artist.

3.2.2 Implementation

In order to control blending of the shadow and light textures, an alpha component is added to the 1D shading texture. As can be seen in Figure 3.10, texels that contain the shadow colour are assigned an alpha value of 0 and all other texels are assigned an aplha value of 1 except for a transitional area where alpha values vary linearly from 0 to 1. This is necessary to achieve a smooth transition between the two detail textures when combining layers. In our implementation we made the two alpha threshold values ta_l and ta_u independent of the shading threshold values to give artists more room for experimentation even though in most cases the two values will be centred around the diffuse threshold t_d.

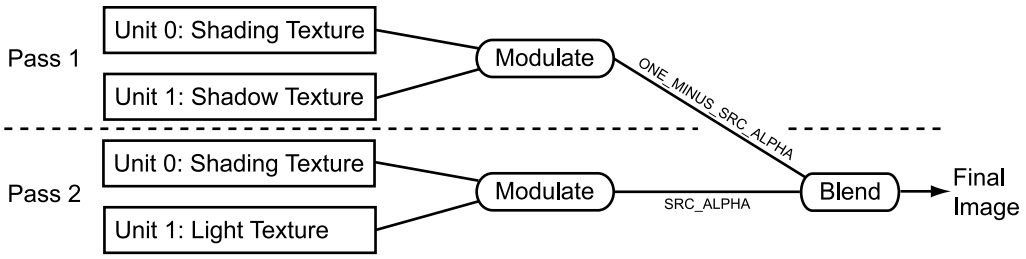


Figure 3.11: 2-Pass Rendering Algorithm in OpenGL

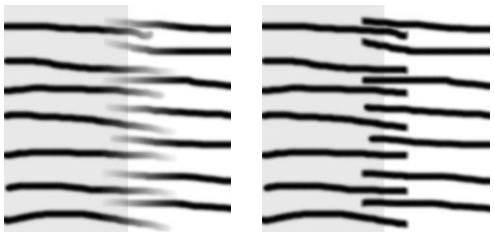


Figure 3.12: Comparison a different transition functions. a) Symmetrical b) Non-Symmetrical (Simulation)

Layer composition is done by a two-pass rendering algorithm in OpenGL. Although modern graphics hardware allows three textures to be combined in one rendering pass, no suitable blending mode is available to compose all three textures in one pass (except if register combiners are used). Therefore we first combine the shading texture with the shadow texture (discarding alpha information). In a second rendering pass, we combine the shading texture with the light texture and blend it with the previously rendered layer using the

```
(SRC_ALPHA,ONE_MINUS_SRC_ALPHA)
```

blending mode. This causes the first layer to 'shine through' in all areas where the alpha value of the second layer is smaller than 1 which is either due to transparency in the light texture or an alpha value that is smaller than one in the shading texture. Figure 3.11 summarises the image compositing steps.

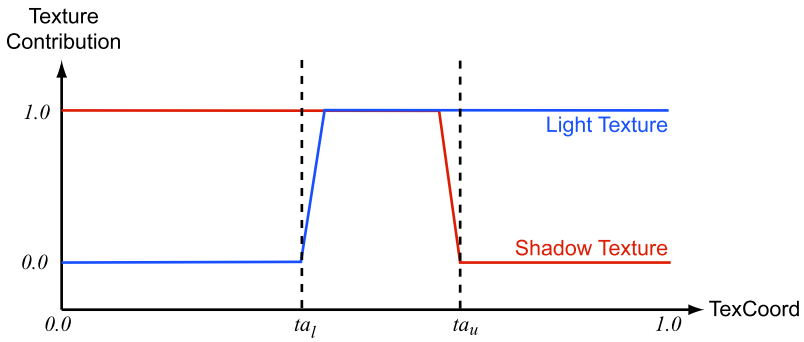
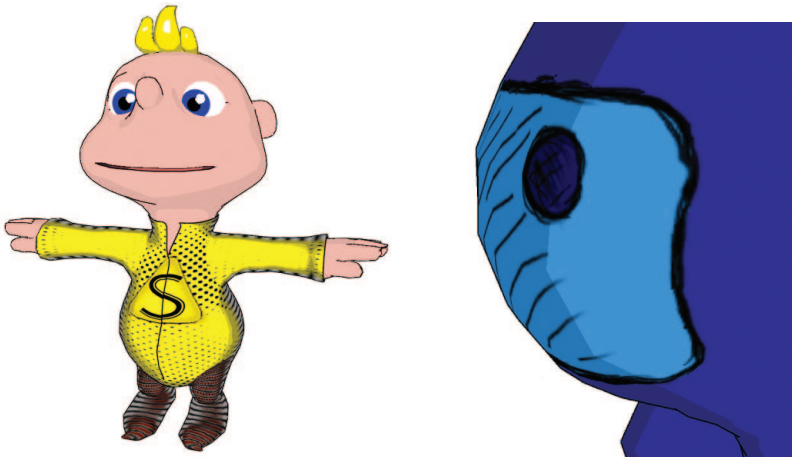


Figure 3.13: Example of a non-symmetrical transition function for stroke-based textures



a b

Figure 3.14: Examples of Textured Shading: a) Superfred b) Detail using Stroke-Based Textures

3.2.3 Results and Discussion

Figure 3.14 shows some example images created with the Textured Cartoon Renderer in Trick17. As can be seen in Figure 3.14a the method works rather well for colourful textures that do not contain many strokes. For purely stroke-based textures, the fading of lines in the transitional area does not look very natural (see Figure 3.14b and 3.12a). This could be remedied by using non-symmetrical transition functions for the light and shadow textures, which would allow both textures to overlap in the transitional area combined with a much steeper falloff (see Figure 3.13). This effect is simulated in Figure 3.12b using image-processing software. Similar approaches have been used by [59] for texture blending in real-time hatching algorithms.

The main difference to the single detail-texture approach from Section 3.1 is that with textured shading, the artist has much more control over the look of the differently shaded areas by specifying two separate textures. It is however not mandatory to use two textures. If for instance, only a shadow texture is applied, the lit region will be coloured with the light colour from the shading texture, whereas in the shadowed regions, the shading colour is combined with the shadow texture. In this way, artists can choose the degree of detail they want to add to each region.

Another application of textured shading is related to highlights. With the basic toon shading technique, the detail texture is also applied to highlights. This is not very natural, because one does not expect to be able to see details in an area that is very bright. With textured shading, we can simulate this effect by setting the shadow texture to the detail map and centring the alpha threshold values around the highlight threshold t_h (see the eyes in Figure 3.14a for an example of this style).

3.3 Half-Toning

In this section we are going to introduce a shading model based on traditional halftoning techniques that can be used to create a variety of artistic styles ranging from charcoal to pen and ink drawings. We first give a brief overview of halftoning as a technique to improve print quality and then present a halftoning method for 3D graphics based on the work of Freudenberg [23] that we extended in two ways. This is followed by an implementation in OpenGL. Finally we discuss the results and give hints about how to further refine the technique.

Figure 3.15: Example of a halftone screen used in printing presses.

3.3.1 Half-Toning

Halftoning was first developed as a method for overcoming the limitations of black and white printing devices. Laser printers and printing presses can often only produce black and white dots. In order to faithfully reproduce grayscale images, it is necessary to approximate gray levels by rendering patterns of black dots with varying spacing. If this is done at a high resolution, it becomes impossible for the human eye to distinguish between individual dots and they are melted into gray 'blobs' depending on the ratio of black dots and white space between these dots. The conversion from gray levels to halftone patterns (also called dither patterns) is accomplished by a halftone screen that associates intensity values to dot patterns (in the case of a device that can only render dots such as laser printers) or forms (for printing presses). As it is often impossible to specify a separate pattern for each possible gray value, thresholding is used to choose the 'closest' pattern. Figure 3.15 shows an example of a halftone screen used in printing presses. Traditionally only round or elliptical primitives have been used to create screens, although in recent years methods have been developed that allow arbitrarily shaped dots to be turned into a halftone screen (see [45]).

3.3.2 Shading Model

Bert Freudenberg was the first to explore the possibilities of halftoning for non-photorealistic shading. He developed a technique that allows the user to specify up to three different halftoning patterns that are used to render dark, medium and light surface points. This may sound like a very rough approximation (traditional screens use many more different levels as can be seen in Figure 3.15), but individual levels can contain very detailed shapes that are spread over more than one surface point. In this respect it is similar to the pen and ink technique developed by Praun et al.[47]. Furthermore in transitional areas halftoning patterns are mixed together which produces smooth transitions.

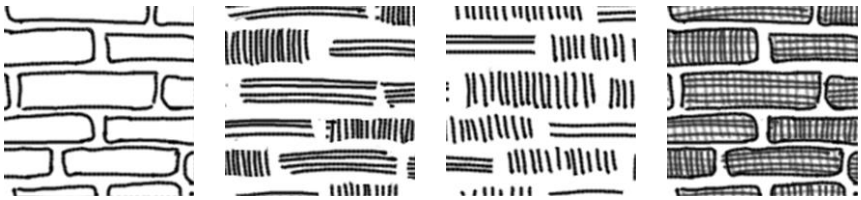


Figure 3.16: Example of halftone patterns used by Freudenberg. A,B and C show the individual patterns associated with light, medium and dark shades and $A'B'C'$ is the rescaled and composited texture map used by the rendering algorithm

The basic halftoning algorithm

The basic algorithm as developed by Freudenberg calculates an intensity value at each surface point using standard Gouraud shading with achromatic light and materials. This intensity value is then used to determine which halftone pattern is applied to the surface point. Figure 3.16 shows an example of such halftone patterns. Up to three different black and white patterns can be specified for each material. For high illumination values, only map A is applied to a surface point, for medium values, the composition of maps A and B is used and for low values, the sum of all three patterns is used (see Figure 3.16).

Although three halftone screens are used, they can be combined into one grayscale halftone texture for rendering using the following scheme:

- The 'lightest' texture (A) is not scaled.
- The grayscale values of the 'medium' texture (B) are remapped to the interval $(0.33, 1)$.
- The grayscale values of the 'darkest' texture (C) are remapped to the interval $(0.66, 1)$.
- The final texture is constructed by setting the grayscale value of each texel to $min\{A', B', C'\}$ where A',B' and C' are the (remapped) intensity values of the corresponding texels in the light, medium and dark textures ($A'B'C'$).

If the source textures are pure black and white images, the final halftone texture contains only 4 different intensity levels, although for most applications better results are yielded by drawing antialiased halftoning patterns that contain some intermediary intensities. Figure 3.17b shows the halftone texture for the brick example.

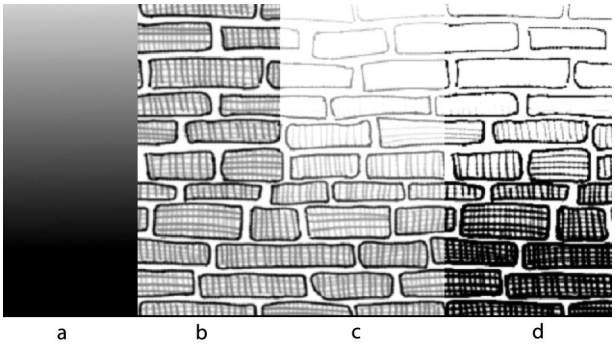


Figure 3.17: Halftone shading using a brick pattern as a halftone screen. a) Illumination value b) Halftone screens combined in one grayscale texture c) Illumination combined with halftone screen d) Final image after applying thresholding

In order to choose the corresponding combination of halftone patterns for each illumination value, one could choose two different approaches: A step function with three levels could be used to associate A, AB or ABC to an intensity level, but this would result in very noticeable transitions between the different areas. Or one could smoothly blend the three patterns (as illustrated in Figure 3.17c), but this would destroy the illusion of halftoning, because too many intermediary levels would be introduced. Freudenberg therefore chose the middle way and used a 'smooth thresholding function' that can be expressed with the following shading equation:

$$C_f = 1 - S(1 - I - H) \tag{3.11}$$

where C_f is the output fragment colour, I is the illumination value, H is the intensity value of the composited halftone texture and S is a scaling factor that determines the smoothness of the thresholding function. Because fragment colours are clamped to the interval $[0, 1]$, the combination of the scaling operation with the double inversion provides the desired result (Figure 3.17d).

Unfortunately this basic technique is rather limited. First of all the Gouraud shading model that is used to calculate the illumination value does not give the artist much control over the relative size of the three differently patterned regions of an object. Secondly, this model offers little support for applying a detail texture to a mesh. We will address these two issues now and present ways of extending the basic algorithm to solve them.

Non-Linear Lighting Model

The original algorithm works by comparing the intensity value (calculated using standard Gouraud shading) at each surface point to the gray value of the corresponding halftone screen texel and choosing one of the three patterns based on the result of this thresholding operation. This gives artists little control over the relative size of the differently patterned region on the object surface (due to implementation issues it is not possible to freely choose the values of the thresholds that determine which pattern is chosen). We therefore replaced the linear diffuse lighting equation

$$I = a_l a_m + max\{Ln, 0\} d_l d_m \tag{3.12}$$

where a_l and d_l are the ambient and diffuse light intensities (as we use no colour information), a_m and d_m represent the ambient and diffuse material intensities, L is the light direction at the surface point and n is the surface normal. We introduced a non-linear *gamma function* for calculating the diffuse component of I, resulting in

$$I = a_l a_m + \left(\frac{Ln + 1}{2}\right)^{\alpha} d_l d_m \tag{3.13}$$

Applying a gamma function (see Figure 3.18) to the diffuse lighting term causes either the middle tones to be accentuated (for $\alpha < 1$) and the very light and dark tones to be compressed and vice versa for $\alpha > 1$. As a result, the boundary between differently textured regions is shifted. As a byproduct, this dynamic range compression can also be used to achieve charcoal-like effects as we will see in 3.3.4.

Adding a Detail Map

The original algorithm does not offer the opportunity to apply a separate detail map to a mesh. Although one could incorporate a detail map into the halftone screens, this solution is not very satisfying, because it requires a lot of preprocessing using image-processing software. This preprocessing would have to be repeated any time the detail texture or the halftone screen changed. Furthermore it would be necessary to fix the size of the halftoning primitives before creating the final screens, because once they are combined, it is not possible anymore to address the two maps independently. Finally, the halftone screens and the detail texture usually have very different characteristics: For halftoning, a detail texture does not need to have a high resolution and it is normally applied without tiling. Halftoning screens on the other hand are very regular and can therefore be easily tiled which saves a lot

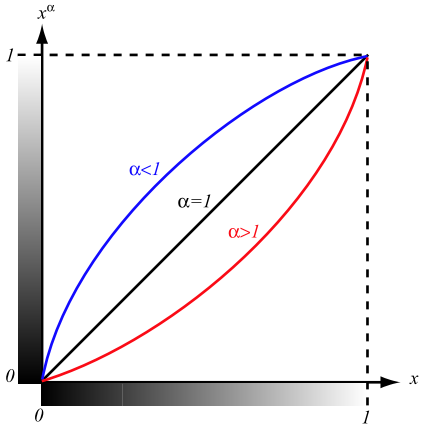


Figure 3.18: Examples of gamma functions using different values for α

of texture memory. As the individual primitives (e.g. dots) should only become visible at close distance, the halftone screens are usually high-resolution maps. As a consequence, we decided to change the shading equation such as to allow two texture maps to be specified, a grayscale detail map T and the combined halftone screen H:

$$C_f = 1 - S(1 - IT - H) \tag{3.14}$$

In this shading model, the intensity value of a surface point is first modulated by the detail texture (similar to standard OpenGL shading) before the thresholding scheme is applied. This small alteration adds a lot of flexibility to the shading model and makes interactive manipulation more intuitive.

Texture coordinates for the halftone screen can be derived from the coordinates for the detail texture by simply multiplying them by a scaling factor s_H. This adds the benefit of allowing the artist to adjust the screen-space size of the halftoning primitive interactively, as higher values of s_H result in smaller, finer patterns.

3.3.3 Implementation

In order to implement the shading model we just described, it is necessary to use multitexturing in combination with the register combiner extension [56]. The illumination value I can be computed with the same approach as with cartoon shading by calculating $\frac{Ln+1}{2}$ using the texture matrix and setting a 1D texture that encodes x^α for $0 <= x <= 1$. As the shading computations also need a texture unit, we need either use a two-pass algorithm or a graphics

board that has at least three texture units. As this is the case with modern graphics hardware (e.g. nVidia Geforce3 or higher), we chose the one-pass approach because it is much faster. The following table shows the texture setup we used:

Unit	Map
0	Non-Linear Shading Function
1	Halftone Screen
2	Detail Map

As the texture coordinates for shading texture are computed automatically from the vertex normal, we only need to specify coordinates for the detail and halftone textures. For the detail map, we simply use the coordinates that are provided with the model. The halftone screen uses the same coordinates, but in order to adjust the size and density of halftone primitives we scale the values by a factor s_H supplied by the user. In this way, one can interactively adjust the resolution of the halftone screen with small values of s_H corresponding to large screen-space primitives.

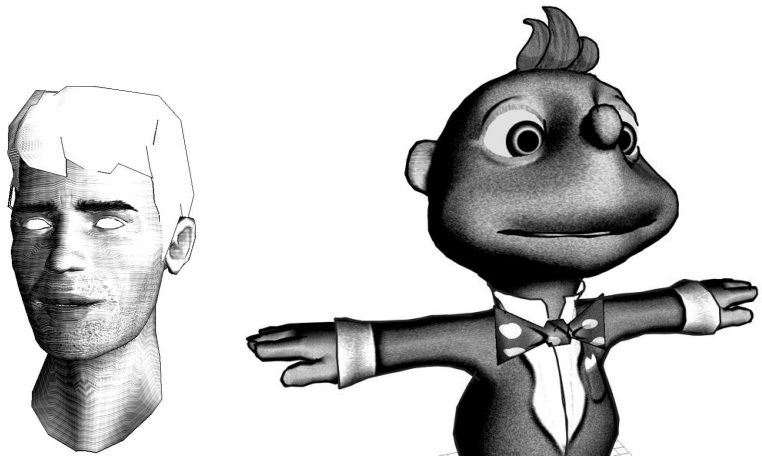
In order to evaluate Equation 3.14, it is necessary to use one general combiner to compute $S(1 - IT - H)$ and the final combiner to invert the result. Because the scaling factor S can not be set to an arbitrary value (only the values 1,2 and 4 are supported) we set $S = 4$. The following code fragment shows the `nvparse` code that configures the combiners:

```
const0 = (1,1,1,1);
{
   rgb{
        discard = const0 * unsigned_invert(tex1.rgb);
        discard = tex2.rgb * -tex0.rgb;
        spare0 = sum();
        scale_by_four();
   }
}
out.rgb = unsigned_invert(spare0);
out.a   = zero;
```

3.3.4 Examples

By using appropriate halftone patterns, a number of different drawing styles can be simulated.

Figure 3.19b was created using a grayscale detail texture and a noise map as a halftone screen with a high gamma value. This results in high contrast



a b

Figure 3.19: Examples of drawing styles generated with different halftone primitives. a) Lines b) Noise

images reminiscent of charcoal drawings. As the noise map already contains many different intensity values, it is not necessary to use more than one halftone pattern.

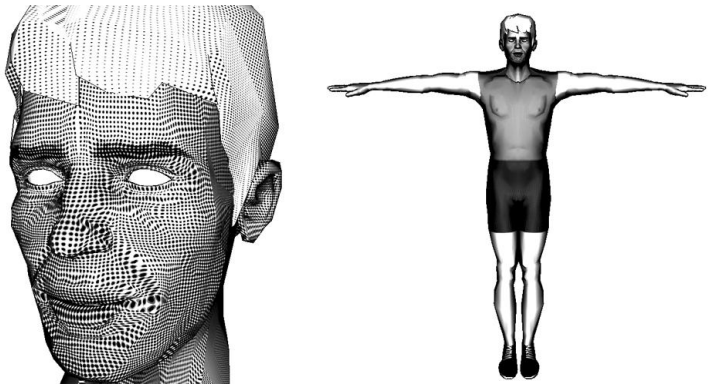
The effect in Figure 3.19a was achieved by combining a grayscale detail texture for the facial features with three halftone patterns that contain lines of differing densities. In the lightest screen, the spacing between lines is highest, whereas in the darkest screen, lines are drawn much denser.

Finally Figure 3.20 shows the effect of using dots as halftoning primitives. Three different dot sizes were used in combination with a detail texture.

3.3.5 Results

As one can see in Figures 3.19 and 3.20, the effect of different halftoning primitives combined with detail textures can be used to create a variety of different visual styles, but there are a number of problems related to the 3D halftoning technique:

- **Distance:** Because the halftoning screen is applied in object space, the screen-space size of halftoning primitives is strongly dependent on the distance of an object from the viewer. As one can see in Figure 3.20, the individual dots are discernible in a close-up view, whereas in a wide-angle view they completely disappear. For still images this is not much of an issue, as one can always adjust the size of the primitives



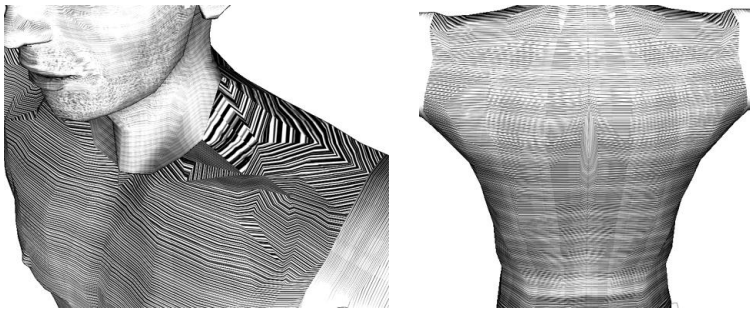
a b

Figure 3.20: The influence of distance on the halftoning effect using dots as primitives. a) Close-Up of the face b) Wide-Angle view

for a given viewpoint, but in real-time animation this is impossible and most models only look as intended at a specific distance.

- **Texture Coordinates:** In order to avoid visual artifacts, special care needs to taken when assigning texture coordinates to a mesh. Most importantly, if texture coordinates are not regularly spaced over the mesh, the screen-space size of halftoning primitives will vary considerably. In Figure 3.21a for instance, the shoulder area is mapped to a much smaller portion of the texture than the rest of the body which causes the line pattern to be scaled differently. Ideally, the ratio of the size of a face to the size of the texture map assigned to it should be constant over the whole mesh. Unfortunately this is very difficult to achieve, especially if a detail map is used that might have different requirements. One solution would be to use two different sets of texture coordinates, one for the detail map and one for halftoning. For directed primitives (such as the line primitives used in Figure 3.21a) another issue arises that makes it even more difficult to generate appropriate texture coordinates, because the orientation of adjacent faces should not differ too much in texture space (in Figure 3.21a the faces of the shoulder region have a very different orientation which results in the zigzag pattern). The standard texturing tools provided by 3D modelling packages are of little help in this case and it would probably be necessary to use a more complex approach such as the Lapped Textures technique developed by Praun et al.[46] to achieve better results.

- **Moiré effects:** A well-known (and much hated) phenomenon from



a b

Figure 3.21: a) Example of artifacts caused by inappropriate texture coordinates. b) Example of Moiré-patterns

printing is the so-called 'Moiré-Effect' that occurs when the regular patterns (such as halftone patterns) are output at certain resolutions. Depending on the relation between the density of the patterns and the frequency at which the pattern is sampled by the output device, different complex patterns can emerge that were not contained in the original image (see Figure 3.21b). As a CRT monitor is a rather low-resolution output device, it is affected by Moiré-effects, especially when regular patterns are scaled, as is the case in texture mapping. This becomes most annoying in animation, because very small changes in the viewing parameters or small deformations cause the Moiré-patterns to morph constantly which is very distracting.

3.4 Conclusions

The three shading models we presented show that it is possible to encode shape information in images without restricting oneself to physical models. One might criticise that the visual styles created by these methods do not differ very much, but this is mostly due to hardware limitations. Especially the capabilities of pixel-processors are currently very limited which is bound to change in the near future with the advent of freely programmable hardware and high-level shading languages. It has already been shown, that this combination will allow the general concept of G-Buffers [52] to be implemented in real-time (see [42]).

Chapter 4

Outlines

As we have seen in Section 2.1.2, outlines provide important cues to the human perceptive system to determine the spatial structure of a scene and especially inter-object relationships. This is also one of the reasons why technical illustrators prefer outline drawings to visualise complex setups (see Figure 4.1a). In this chapter, we are first going to give an overview of the different approaches to outline image generation that have been explored, and then present three techniques together with an OpenGL implementation and a discussion of their advantages and problems. Our main contribution in this chapter lies in the geometric method presented in Section 4.4. Although this method is partly similar to the work of Isenberg et al. [33], it includes some novel approaches to solving the visibility problem and drawing textured quads using the graphics hardware.

4.1 Introduction

4.1.1 Definitions

Before we can start creating outline images, we first need to identify the elements that make up such images. Even a simple drawing, such as the sketch from Picasso shown in Figure 4.1b, uses many different classes of lines including:

- **Silhouettes** are those outlines that separate an object from the background or from objects that are further away. Mathematically, silhouettes can be defined in different ways depending on the space (i.e. image space or object space) in which the algorithm operates.
- **Creases** (also known as valleys and ridges) are caused by a sharp change in the curvature of an object surface, e.g. wrinkles. For triangle

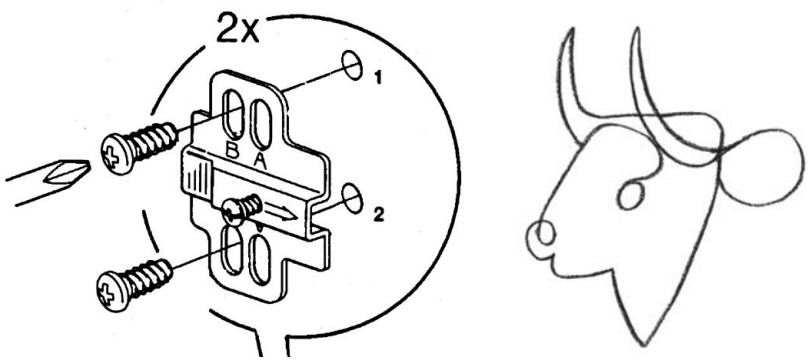


Figure 4.1: a) Technical Illustration b) Sketch by Picasso

meshes, this corresponds to a large dihedral angle between adjacent faces.

- **Boundary Edges** are only defined for non-closed objects. In a triangle mesh, a boundary edge is an edge that is only connected to one face.
- **Material Edges** separate regions of different surface properties. As material properties are usually defined on a per-face basis, material edges correspond to triangle edges.

In this chapter, we will mainly be concerned with silhouettes, because they constitute the largest and most important class of outlines, although we will also hint at strategies for handling the other types of outlines.

4.1.2 Overview

We are now ready to look at the various approaches for silhouette depiction that have been explored.

Image-Space Algorithms

Image space algorithms try to extract silhouettes from 2D images [17]. Usually they operate on special images that encode properties of the rendered objects as colour values. One such kind of image is the depth map that stores the depth of surface points corresponding to framebuffer pixels as greyscale values (see Figure 4.2a). In image space, silhouettes are then simply defined as C^0 discontinuities of a depth map, i.e. pairs of neighbouring pixels that expose large intensity differences. As border edges also generate discontinuities

a b c

Figure 4.2: a) Depth Buffer b) Result of edge detection c) Final Image

in the depth map, they are detected as a byproduct of silhouette extraction. Similarly, creases can be defined as either C^1 discontinuities in the depth map or equivalentely as C^0 discontinuities in a normal map. To generate a normal map, a scene is rendered without lighting and using the normals as vertex colours. Finally, material edges can be identified by rendering a special ID image where every material is assigned a different fixed colour. In this case, material boundaries are characterised by discontinuities in the ID image. In order to extract discontinuities from 2D images, edge detection operators (i.e. first-order differential operators) can be applied (Figure 4.2b). As edge detection is a common problem in computer vision, many efficient algorithms have been developed for this purpose including the Sobel edge detection operator and the LoG (Laplacian of Gaussian) operator that combines edge extraction and smoothing, which yields better results under some circumstances. In general, these operators generate lines that are exactly one pixel wide, but morphological operators can be used to expand these lines to any desired thickness. After the various maps have been processed by image operators, they can be composited into a single image that contains all the different types of lines, similar to the stacking of transparent sheets in cel animation (Figure 4.2c).

Although the resulting images look visually pleasing, artistic freedom is very limited. As the 2D maps contain no connectivity information or spatial data, only the line width and colour can be influenced. To overcome these limitations, some approaches combine image space silhouette extraction and geometric methods, but even without the overhead added by these dual methods, image space silhouette extraction is currently too slow for real-time applications as accessing the framebuffer that contains the special maps is very time-consuming on current graphics hardware. As shown by Mitchell

[42] this will change with next-generation graphics boards.

Object-space Silhouette Extraction

Another approach tries to extract silhouettes from 3D objects instead of 2D images. Given the position and orientation of an object in the world and the current viewpoint, they try to identify those points on an object surface that make up the silhouette. Two fundamentally different approaches can be chosen to achieve this goal [30].

Firstly, *individual surface points* can be tested and classified as either silhouette or inner points. This method is independent of the underlying surface description (e.g. triangle meshes or NURBS surfaces) and works best for very smooth surfaces. Unfortunately, it suffers from the same lack of stylistic freedom as image-based methods, because it only considers individual surface points, but it can be implemented very efficiently on modern graphics hardware as will be described in section 4.3.

Secondly, for triangle meshes, one can examine the object geometry to determine those *edges* that belong to the silhouette. As the silhouette data still contains spatial information (positions of the edge vertices and normals) as well as connectivity information, it can be further processed and rendered using different primitives (e.g. lines or triangles). The biggest problem of geometric methods is visibility determination. Because the stylised outlines can partially overlap with the underlying shading, it is no longer possible to rely on z-buffering for visibility testing. In Section 4.4, we will present a geometric silhouette extraction algorithm and a fast visibility test using the graphics hardware.

4.1.3 Character Animation Issues

Rendering animated characters as line drawings adds a few special requirements that rule out some rendering methods that have been explored by computer graphics researchers:

- Many silhouette extraction algorithms have only been designed for static meshes. Hertzmann et al. [31] for instance use extensive preprocessing to generate an alternative representation of the mesh structure in dual space to speed up silhouette extraction. For an *animated character*, this preprocessing step would have to be repeated every time the mesh is deformed. Although the actual silhouette detection algorithm is very fast, building the dual space representation of the mesh is too expensive for real-time applications.

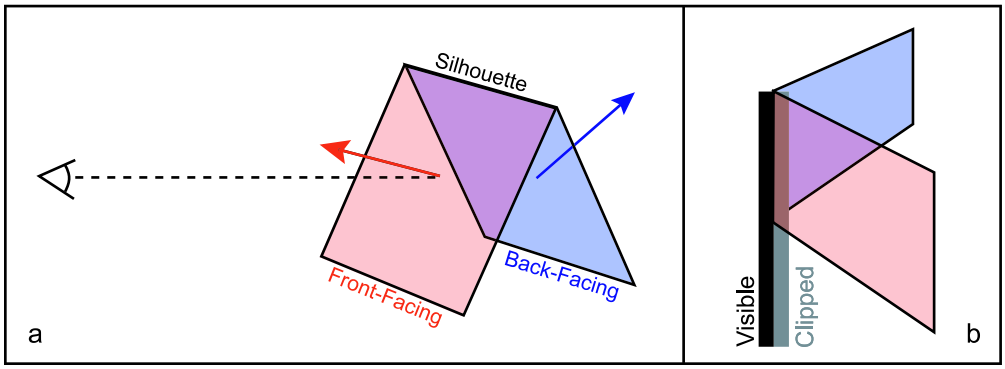


Figure 4.3: a) Definition of Silhouette edges b) Clipping of Silhouette Lines. If a silhouette edge is rendered as a wide line (¿=2 pixels wide), about half of the fragments pass the depth test.

- Another important issue of character animation is *temporal coherence*. If a character is deformed or moved, outlines should stay as stable as possible. Especially popping effects, caused by new edges being added to or removed from the silhouette, catch the beholder's attention and distract them from the animation or story, thus weakening immersion. This is mainly an issue for geometrical methods as with all other techniques only individual pixels are added or removed.

4.2 Wireframe Outlines

In this section we are going to present a simple method for drawing silhouettes. As all computations are performed by the graphics hardware, this method is very fast, although it requires two rendering passes. This technique was first presented by Rossignac et al. [51] and has been applied successfully to complex real-time applications [12].

4.2.1 Method

The wireframe algorithm works only for triangle meshes as it uses a definition of silhouettes that is only valid for models composed of triangular faces. For this class of objects, an edge is part of the silhouette if it connected to one front-facing and one back-facing triangle. As can be seen in Figure 4.3a, this definition is sufficient and complete. The basic algorithm for drawing silhouettes uses two rendering passes:

1. The depth buffer is first filled with the depth values of the front-facing triangles. If the outlines are to be combined with shading, rendering the shaded front faces provides the depth buffer with no additional cost.

2. Back-facing polygons are then drawn in wireframe mode with the depth test enabled and with line width greater than one .

As depth testing is used, edges that lie behind the shaded front-faces are culled and only a halo appears around the object boundaries, because the lines are centred on the object vertices and about half of the pixels are drawn outside the object (see Figure 4.3b).

Unfortunately, this algorithm does not detect border edges and creases. We therefore need to identify them by performing an exhaustive test of all mesh edges. Luckily connectivity is not affected by skeletal animation or morphing, so we only have to identify border edges once as a preprocessing step and can store them with the mesh. On the other hand, it is necessary to recalculate creases every time the mesh is deformed. For an edge E with adjacent faces F_1 and F_2, this can be achieved by thresholding the dot product $n_1 n_2$ against a reference value t where n_1 and n_2 are the face normals of F_1 and F_2. If $n_1 n_2 < t$, the dihedral angle between F_1 and F_2 is larger than the threshold angle and E is classified as a crease.

We will now present an OpenGL implementation of this technique and afterwards discuss its advantages and problems.

4.2.2 Implementation

The algorithm we just described can be implemented in OpenGL in a straightforward fashion. It requires no extensions and therefore works on any 3D graphics board that supports OpenGL. The following table shows the important state settings for the two rendering passes.

Pass	**Depth Function**	**Culling Mode**	**Fill Mode**
Shading	`GL_LESS`	`GL_BACK`	`GL_FILL`
Outlines	`GL_LEQUAL`	`GL_FRONT`	`GL_LINE`

Note that the algorithm only works correctly if all objects use consistent vertex ordering, because else face culling will generate unpredictable results and some lines will be drawn on the object surface. In addition to the silhouette, border and crease lines can be rendered as simple lines using the `GL_LEQUAL` depth function.

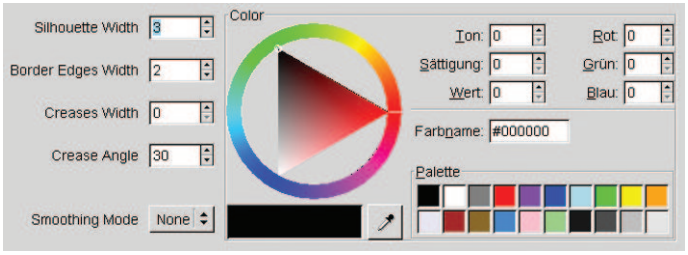


Figure 4.4: The user interface for wireframe outlines in Trick17

By default, OpenGL does not use antialiasing when rendering lines. We therefore added the option to draw antialised lines using a combination of line smoothing and alpha blending. Furthermore the artist can choose the line colour and set the line width for silhouettes, creases and border edges independently of each other.

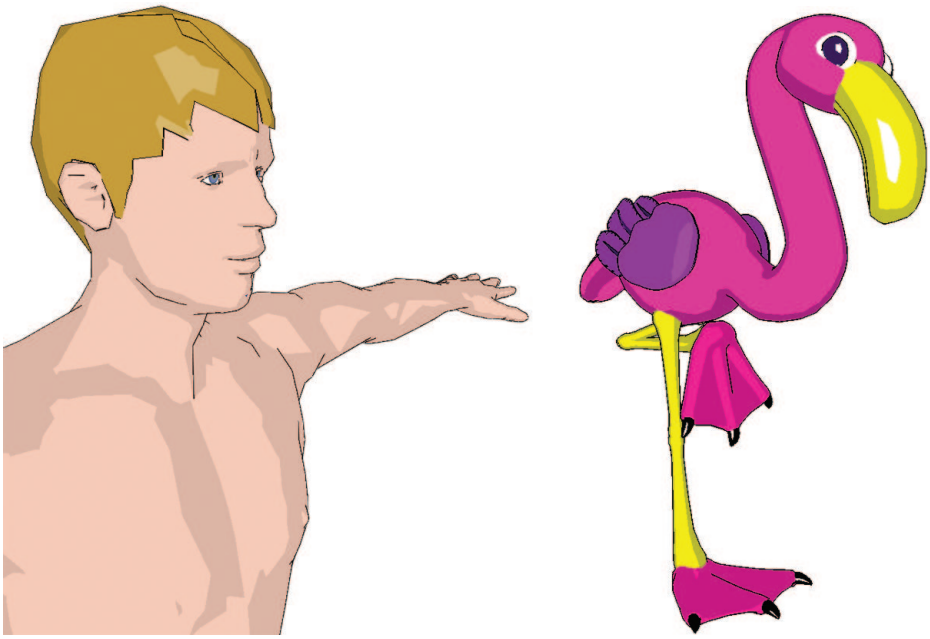
4.2.3 Examples and Discussion

Figure 4.5 shows examples of images generated using the wireframe technique and Figure 4.4 presents the user interface for adjusting the rendering style in the Trick17 system. We will now examine in detail the properties of our OpenGL implementation and hint at possible enhancements.

- **Performance:** Although two rendering passes are necessary to generate a silhouette drawing, the algorithm is still fairly fast (see Section 4.5 for a comparison of rendering times for the different methods). Even extracting and drawing creases adds only negligible overhead. On graphics boards that do not perform line smoothing in hardware, enabling antialiasing slows down rendering considerably.

- **Stylistic Freedom:** Only a few properties of the outlines can be influenced by the artist. These include line colour, the threshold angle for crease extraction and line width for the different classes of outlines. This is sufficient if the outlines are combined with cartoon shading, but for pure line drawings, it would be desirable to draw more expressive strokes with varying line width and textures.

- **Rendering Artifacts:** As the outlines are clipped against the depth buffer generated by the first rendering pass, the quality of the images strongly depends on the resolution of the depth buffer. Especially if the near and far clipping planes of the camera are far apart, isolated

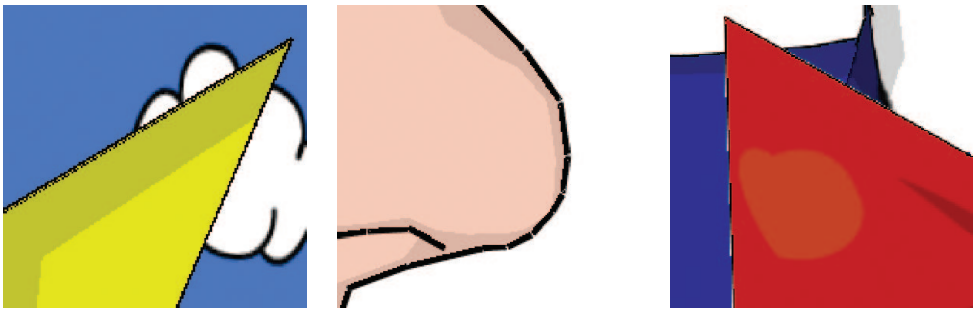


a b

Figure 4.5: Examples of wireframe silhouettes. a) Bumba b) Flamingo

outline pixels appear on the inside of the object (see Figure 4.6a). This can be remedied by using the stencil buffer to cull fragments that would be drawn over the shaded areas[37].
For large line widths, small gaps become visible between line segments that are not colinear in screen-space (see Figure 4.6b). This is caused by the fact that OpenGL renders individual lines as rectangles. One solution to this problem consists in adding an extra rendering pass where all object vertices are rendered using point primitives with a radius equal to the line width used for drawing silhouettes.
If antialiasing is enabled, the algorithm needs to be implemented as a true two-pass algorithm, i.e. first the depth buffer is filled with the depth values of all objects and then outlines are drawn, because else z-ordering of objects can conflict with alpha blending, which creates white halos around some outlines (see Figure 4.6c).

- **Temporal Coherence:** In general no popping of lines is visible because internally the algorithm tests individual pixels and not line segments, but only small changes in viewing parameters or small deformation of the mesh are sufficient for the artifacts caused by depth-buffer



a b c

Figure 4.6: Rendering artifacts. a) Depth buffer imprecision b) Gaps between line segments c) z-ordering conflict with antialiasing

imprecision to appear and disappear, creating a disturbing flickering effect.

- **Prerequisites:** The algorithm can handle any kind of triangle mesh and is fairly stable with respect to large differences in mesh tesselation.
- **Aesthetics:** In combination with cartoon shading, wireframe outlines look nice if a small line width is chosen, but if a more hand-drawn style is aimed at, one of the other presented methods should be employed.

4.3 Per-Pixel Outlines

The second approach to outline drawings that we investigated, does not rely on mesh edges to determine silhouette lines, but tests all surface points of an object. It could therefore be easily adjusted to other surface descriptions than triangle meshes such as NURBS surfaces. In this section, we will present a very fast hardware-accelerated rendering method that can combine silhouettes and shading in a single rendering pass.

4.3.1 Method

For smooth surfaces, silhouettes can be defined in an alternative manner to the one we used in Section 4.2: Given a surface point P with normal vector n and the current viewpoint v, P is a silhouette point if it satisfies $(v - P)n = 0$, in other words if the suface normal at P is orthogonal to the view vector as illustrated in Figure 4.7a. Unfortunately this definition

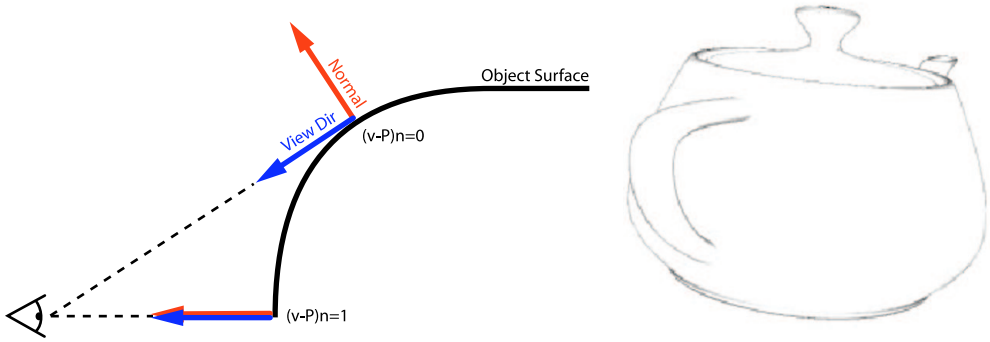


Figure 4.7: a) Per-Pixel definition of silhouettes. b) Example of per-pixel outlines rendered using cubemaps.

works only correctly if the object surface is very smooth. As we work with triangle meshes, which can only approximate smooth surfaces by using high tesselation, we need to slightly alter this equation to $(v - P)n < \epsilon$, where ϵ is a very small constant that needs to be adjusted to the structure of the considered triangle mesh. Using this definition, we are now ready to develop an OpenGL implementation.

4.3.2 Implementation

There exist two ways in which the algorithm can be implemented in OpenGL, each having different advantages and problems. As described in [21], a combination of cubemaps and register combiners could be used to calculate the dot product $(v - P)n$ and perform thresholding. This approach generates thin greyish outlines as shown in Figure 4.7b. The technique we chose calculates the dot product in a vertex program and uses the result to look up colour values in a 1D threshold texture map, similar to the cartoon shading method described in Section 3.1.2. We will now describe these two steps in more detail.

In Section 2.2.1 we saw that a vertex program is a small SIMD assembler program that takes object space vertex data and material definitions as input, and outputs the transformed vertex position together with texture coordinates and vertex colours. Figure 4.8a illustrates the input and output data of the vertex program we use to calculate the texture coordinates for shading and silhouette drawing. The shading calculations are exactly the same as in Section 3.1.2 with the advantage that the vertex program could

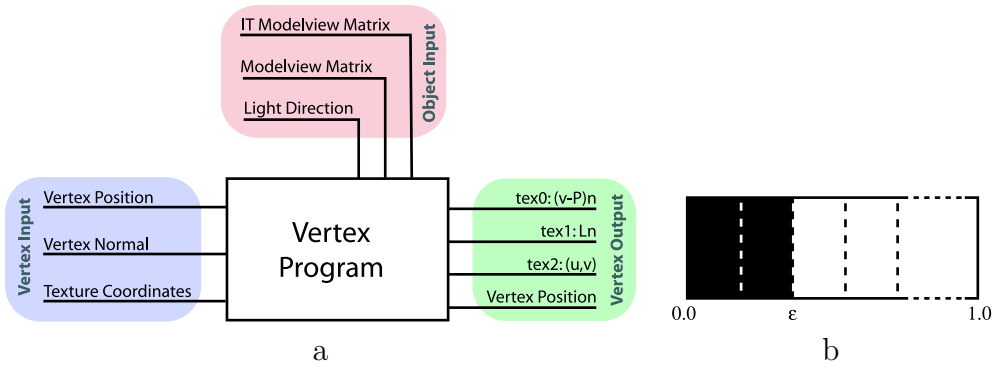


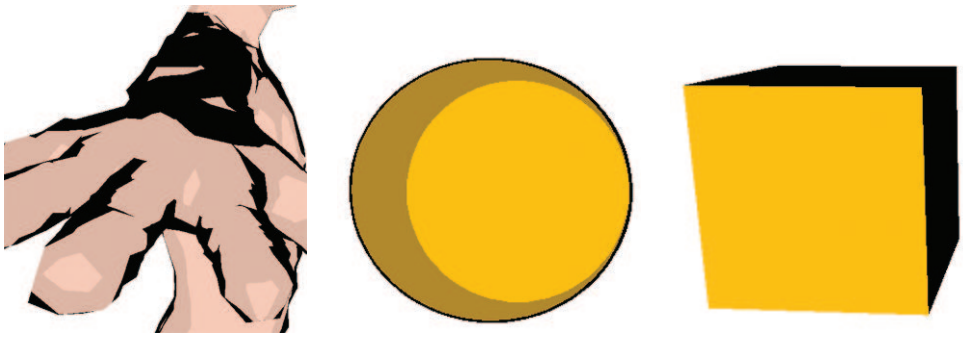
Figure 4.8: a) Vertex Program input and output data b) Threshold Texture

be easily extended to support point light sources, which is not possible with the texture matrix approach. To calculate $(v - P)n$ we first transform P and n to eye space using the modelview matrix, respectively the inverse transpose of the modelview matrix yielding P' and n'. As, in eye space, the viewpoint is always located at the origin we now only need to compute $-P'n'$ and store the result in `tex0`. The complete vertex program assembler code can be found in Appendix A.1.

The texture coordinates calculated by the vertex program are then used to access a 1D threshold texture that contains black texels in the lower entries and white texels in the upper entries (see Figure 4.8b). The percentage of black texels depends on the threshold value ϵ. For low threshold values, the texture contains mostly white entries, whereas for higher values, the number of black texels increases. As we want a crisp transition between the silhouette and inner regions, the texture interpolation mode is set to `GL_NEAREST`. This setup creates an image where surface points near the silhouette are drawn in black and all other points are set to white. This layer can then optionally be combined with a shading layer and a detail texture depending on the number of texture units available. As the different layers are blended together using the `GL_MODULATE` mode, the black outline is preserved while all white inner surface points are assigned the shading colour.

4.3.3 Discussion

- **Performance:** If enough texture units are available (max. 3 units are needed for silhouettes, shading and a detail texture), all silhouette and shading calculations can be performed on the graphics hardware in one rendering pass. Therefore this method is the fastest of the methods we



a b c

Figure 4.9: a) View angle b) Best-Case Example c) Worst-Case Example

implemented and tested (see 4.5).

- **Stylistic Freedom:** Per-pixel outlines offer even less stylistic freedom than wireframe outlines.In our implementation, the silhouette colour needs to be set to black if the silhouette layer is to be combined with shading or a detail texture as else blending causes colour shifts in the outline pixels. The silhouette width can be varied, but only reasonably small values make sense as higher values for ϵ (which controls line width) introduce too many artifacts (see below).

- **Rendering Artifacts:** Although the algorithm introduces no true artifacts, the uneven silhouette width can look very unnatural under some viewing angles, especially when the view vector is almost colinear to a large face near the silhouette, as can be seen in Figure 4.9a. In this case it can happen that the whole face is drawn in black, destroying the illusion that it represents the object outline. The effect depends strongly on the tesselation and smoothness of the object surface. It is less noticeable for very smooth surfaces that consist of small faces (see Figure 4.9b). One potential remedy to this problem might consist in the usage of custom mipmaps similar to the approach used by Everitt [21] in the cubemap method, although this would probably not be sufficient to account for the large variations in tesselation of a typical character mesh.

- **Temporal Coherence:** In general no popping artifacts have been observed and all transitions of the outline are smooth, but as the silhouette width depends on the size and orientation of mesh faces, it can vary considerably if the viewpoint is slightly shifted or the model is an-

imated. As this variation is not linked to any intuitive object feature, it can be very distracting.

- **Prerequisites:** Theoretically the algorithm can be applied to any closed or open triangle mesh, but it can only handle smooth objects reasonably well, it totally fails for objects that contain sharp features (worst case example: a cube or tetrahedron, see Figure 4.9c).

- **Aesthetics:** Under some viewing angles this technique can generate interesting images with varying outline width, but as soon as the object is deformed or the viewpoint is changed, the black outline can expand to the inside of the object resulting in unbalanced proportions between the outlines and shading. Furthermore the thickness of the silhouettes varies in a counterintuitive manner: Whereas an artist would use thicker lines in areas of high curvature, the algorithm does exactly the opposite and emphasises flat areas.

4.4 Geometric Strokes

The next method we are going to describe, is by far the most complex of all, but it also yields the most versatile results. It first extracts the silhouette edges in 3D space, processes them and finally renders them as stylised strokes. Conceptually our technique is similar to the work of Isenberg et al. [33], but we developed a very efficient hardware-accelerated visibility test for lines and used an alternative approach to rendering stylised strokes.

4.4.1 Method

We first provide an overview of the stages involved in rendering silhouettes as geometric strokes (see also Figure 4.10). In the next subsections, we will then describe in detail how these subproblems can be solved.

- **Preprocessing:** In order to accelerate silhouette extraction and stroke construction, we first build a list of mesh edges with connectivity information.

- **Silhouette Extraction and Edge Linking:** The recursive edge extraction algorithm automatically generates connected chains of silhouette edges. This is especially useful as we will render connected strokes in a later step.

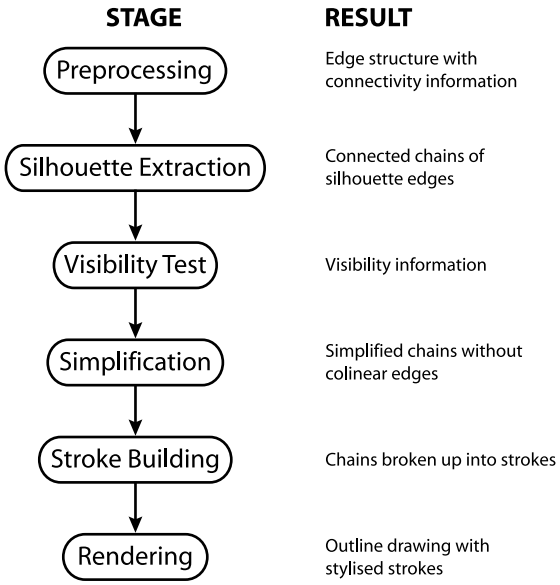


Figure 4.10: Program flow for the geometric strokes algorithm. The left-hand side shows the processing stages and the right-hand side describes the data generated by each stage.

- **Visibility Determination:** Some drawing styles require that silhouette lines overlap with shaded areas. It is therefore necessary to determine line visibility before rendering, because we cannot rely on the depth test to eliminate hidden lines. To solve this problem, we developed a fast hardware-accelerated visibility test for line segments with adjustable precision.

- **Simplification and Stroke Building:** The visibility test can generate many small colinear line segments. These can be simplified without loss of precision by replacing them by a single segment. Furthermore we break up segment chains at sharp corners to achieve a more natural-looking drawing style and to eliminate rendering artifacts.

- **Rendering:** Finally strokes are rendered without depth test, using either lines or screen-aligned textured quads.

4.4.2 Preprocessing

In a preprocessing step, we build a list of `EdgeData` structures that contain all information about a single edge that is needed by the silhouette test and

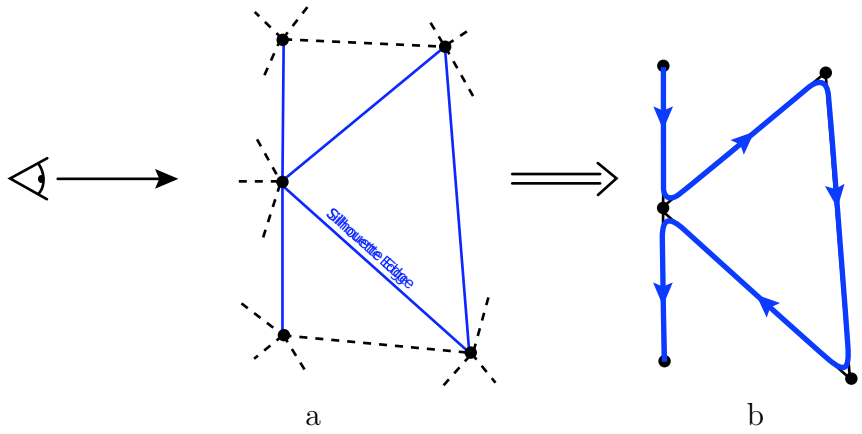


Figure 4.11: a) Example of a silhouette edge with 3 adjacent silhouette edges b) One possible silhouette chain

edge linking. An `EdgeData` entry stores indices for accessing vertex positions of the segment endpoints and the normal vectors of neighbouring faces. Furthermore we build a separate list of neighbour edges for both endpoints. This information can be extracted in a straightforward fashion from the mesh data, but it is time-intensive (the time requirement for building neighbour lists is $O(n^2)$ for a mesh with n edges). As we store indices that never change, we only need to build this structure once and can thereby speed-up per-frame computations considerably. Furthermore every edge endpoint has only a small number of adjacent edges resulting in linear memory requirements that are negligible even for large meshes.

4.4.3 Silhouette Extraction

The result of silhouette extraction is a list of `SilhouetteLoop` structures that store chains of connected silhouette edges. The usefulness of this approach can be illustrated by two observations:

- A silhouette edge can never exist in isolation, each of its endpoints is connected to exactly one other silhouette edge. Thus silhouette edges form closed loops.
- Silhouette loops never intersect in 3D space (although their screen projections will often intersect).

These statements are only valid for closed, perfectly tesselated objects. As we want to support meshes of any resolution that can also contain holes, our

algorithm uses weaker assumptions. Firstly, a silhouette loop does not need to be closed. This case arises when the silhouette would pass through a hole in the mesh. Secondly, a silhouette edge endpoint can be connected to more than one neighbouring edge, but it is still possible to construct a linked chain without bifurcations that encompasses all silhouette edges (Figure 4.11a). This effect is due to low tesselation and the fact that a triangle mesh can only approximate a smooth surface. We first explain the silhouette test that we used to determine which edges are part of the silhouette and then present a recursive algorithm that generates chains of coherently oriented silhouette edges.

Silhouette Test

The silhouette test relies on the same definition that was used in 4.2, with the difference that it is no longer possible to perform the face orientation test in hardware. Given a mesh edge E with endpoints v_1 and v_2, the normal vectors n_1 and n_2 of the adjacent faces and the viewpoint v, E is a silhouette edge if it satisfies the following equation:

$$((v - \frac{(v_1 + v_2)}{2})n_1)((v - \frac{(v_1 + v_2)}{2})n_2) < 0 \tag{4.1}$$

where $(v - \frac{(v_1+v_2)}{2})$ is simply the average view vector of E. Equation 4.1 therefore states that the neighbouring faces of E have to be oriented in opposite directions with respect to the view vector, which is exactly what we aimed at.

The test requires that all vectors are expressed it in the same space. All vertex positions and face normals are provided in object space whereas the viewpoint is either given in eye or world space. As it is faster to extract and invert the current modelview matrix once for every object than to multiply every vertex and face normal by a matrix, we convert the viewpoint to object space. This reduces the expensive per-edge computations to a single dot product.

Recursive Silhouette Extraction

The silhouette extraction algorithm we used is a combination of a brute-force technique that tests all edges in sequential order and the randomised algorithm proposed by Markosian [38]. As some stroke building methods rely on a fixed edge order of the loops, we cannot use randomisation, but similar to Markosian, we try to extend every silhouette edge we detect to a chain of

connected edges during extraction. This saves us an $O(n^2)$ postprocessing step that would be necessary with the simple sequential brute-force method.

Informally the algorithm searches for an unvisited silhouette edge and then recursively tries to find adjacent silhouette edges. This search is first only conducted in one direction. If the loop is closed, we will eventually arrive again at the start vertex and we can stop. Else we restart the search at the first edge, but this time we continue in the opposite direction and prepend the new edges to the chain. This ensures that we always build chains of maximal length. The following code snippet shows the main extraction function as pseudo-code:

```
While there are unvisited edges
{
  Find the first unvisited edge
  If it is a silhouette edge
  {
    Start a new loop
    Recurse (Forward Direction)
    If the loop is not closed
      Recurse (Backward Direction)
    Postprocess loop
  }
}
```

For every silhouette edge we find, a `SilhouetteEntry` structure is created and stored in the current `SilhouetteLoop` object. It contains only information about the starting vertex of the current edge including the vertex position, two tangent vectors and a visibility flag. The main reason for not including the second vertex is performance. As some postprocessing steps refine loop segments or delete vertices, with our approach we only need to update one `SilhouetteEntry` object instead of two. The loop postprocessing operation in the above code updates some tangent vector information in the `SilhouetteEntry` that is needed by the subsequent processing stages.

The following code snippet presents the recursive extraction function that extends a single silhouette edge to a connected chain:

```
Fill-in and store SilouetteEntry structure for current edge
If we have reached the start vertex
  return ClosedLoop
Determine which endpoint to use for testing neighbours
Loop over neighbour edges
```

```
{
  If next neighbour edge is a silhouette edge
    Recurse
}Until no neighbours are left or we've found a silhouette edge
If no silhouette edge has been found
  return OpenLoop
```

At the end of the extraction process we have a set of silhouette loops that consist of a list of vertices with tangents and a visibility flag.

4.4.4 Visibility

Visibility determination has for a long time been the bottleneck of line drawing algorithms. Although there exist a large number of different approaches to this problem, none of them was appropriate for our purposes. At the one end of the spectrum, we have z-buffer methods that work similar to the wireframe outlines algorithm we described in 4.2 which are very fast but do not allow strokes to overlap with shaded areas. At the other end there are analytical methods such as Appel's hidden line algorithm [9] that compute the exact intersection points of line segments, and therefore provide very accurate visibility information, but as they are based on solving a large number of equations, they are not suitable for real-time applications. We are now going to describe a novel solution that combines the advantages of both of these approaches, i.e. it is very fast and can be tuned to provide results of any desired precision.

Our visibility test relies on the `NV_OCCLUSION_QUERY` OpenGL extension described in Section 2.2.1. The basic idea is to determine for every silhouette edge the fraction of the segment that would appear on the screen if we simply rendered the segment as a line of a certain width. The algorithm works in two stages that are repeated for every silhouette loop:

- We submit every line segment twice using different query IDs. The first time we draw the line without depth test, the second time with depth testing enabled.

- We then read back the test results and calculate the fraction of the line segment that is visible. Disabling depth testing during the first query provides the maximum number of pixels that could be drawn if the line was entirely visible whereas enabling depth testing for the second query results in the number of pixels that were actually drawn. This allows us to calculate the visibility ratio $V_r = \frac{\text{number of visible pixels}}{\text{maximum number of pixels}}$ and decide whether the line segment should be drawn.

a b

Figure 4.12: Problems of the Visibility Test a) Edges are only culled if they are completely invisible b) All edges that are not completely visible are culled leaving gaps

Note that we disable framebuffer and depth-buffer writing during visibility testing. In this case, occlusion queries return the number of potentially drawn pixels which is enough for our purposes and speeds up the drawing operations.

After we have determined the fraction of pixels that would be drawn, we can decide the visibility problem by thresholding. If $V_r <= t_{vis}$ only a small portion of the segment is visibility and we therefore cull it. Conversely for $V_r > t_{vis}$ the line segment is considered to be visible. Note that depth-testing will always eliminate about half of the pixels as part of the line overlaps with shading thus $t_{vis} > 0.5$ will eliminate virtually all segments (some segments may still pass the test because of calculation imprecisions, see 4.4.8). Setting $t_{vis} = 0$ will cull only lines that would be totally hidden, resulting in 'sketchy' images where strokes are extended a bit further than they should if a precise drawing style was used.

Unfortunately, it is impossible to choose a suitable value of t_{vis} for all line segments. If t_{vis} is too low, long lines that would only be partially visible will be extended too far (see Figure 4.12a). On the other hand for $t_{vis} > 0$, gaps will start to appear around intersection points (see Figure 4.12b). This is mainly caused by differing screen-space segment lengths. We therefore refined the algorithm by adding a subdivision scheme for ambiguous segments. The new approach works basically by splitting problematic segments in two and testing each of the new segments individually (see Figure 4.13a-d). This refinement step can either be repeated until all segments have been classified or it can be limited to a maximum number of iterations. We implemented the algorithm using two threshold values: for $V_r <= t_{invis}$ the segment is not visible and for $V_r >= t_{vis}$ it is visible. If neither of these conditions is fulfilled,

the segment is ambiguous and we split it in the middle. The following code snippet illustrates the query submission part of the algorithm for a single loop:

```
Disable Frame- and Depth-Buffer writing
Set line width
Loop over SilhouetteEntries
{
  If visibility of the segment is VS_UNKNOWN
  {
    Disable depth test
    Start new query
    Submit line vertices
    End query

    Enable depth test
    Start new query
    Submit line vertices
    End query
  }
}
```

The pseudo-code for the evaluation phase looks as follows:

```
Loop over SilhouetteEntries
{
  If visibility of the segment is VS_UNKNOWN
  {
    Retrieve query results
    Compute fRatio
    If fRatio<fInvisibleThreshold
      Mark edge as VS_INVISIBLE
    If fRatio>fVisibleThreshold
      Mark edge as VS_VISIBLE
    Else
    {
      Fill-in new SilhouetteEntry
      Mark it as VS_UNKNOW
      Store object in current loop
    }
  }
}
```

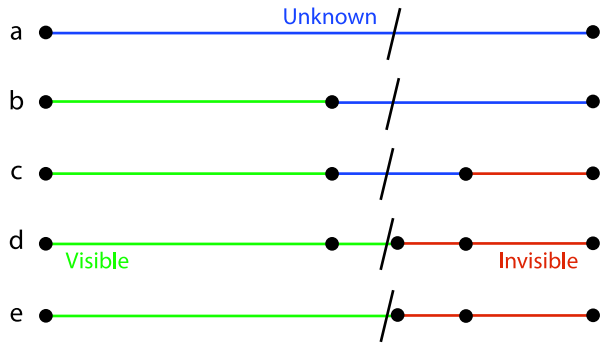


Figure 4.13: Edge subdivision for visibility determination a) The blue silhouette segment is ambiguous and needs to be refined. b) After subdivision, the left segment is completely visible whereas the right segment is ambiguous. c) After a second refinement step, only the middle segment is still ambiguous. d) After three iterations, all segments can be classified as hidden or visible with a small error. e) As all segments are colinear, the two visible segments can be collapsed into one.

These steps are repeated until either all edges have been classified or no further refinements are allowed. If there are still unclassified edges left after the maximum number of refinement steps, we use the average of the two threshold values to decide their visibility state.

4.4.5 Simplification

One drawback of the refinement scheme for visibility determination is that it can create many small colinear segments that could, without loss of precision, be replaced by a single segment, that can be handled better by the subsequent drawing step (see Figure 4.13d+e). The simplification algorithm simply tests all adjacent loop edges for colinearity and collapses edge pairs that pass the test. It would be tempting to also collapse pairs that are almost colinear, but this destroys too many subtle variations in line direction that are immediately noticeable. Of course only visible edges need to be considered by the simplification algorithm.

4.4.6 Stroke Building

We would like to draw the chains of visible edges as strokes similar to an artist. As at this point chains can be very long and contain very sharp angles, we break them up into smaller chains that look more natural. This is achieved by applying two different criteria:

- The maximum length of a chain can be limited to a fixed number of segments. This works best if all edges have approximately the same length in screen-space.

- If the screen-space angle between two adjacent edges is very sharp, the chain is broken up into two separate strokes.

As these two rules can conflict with each other, the latter rule takes precedence over the former. One could also devise further rules for stroke building, for instance in the first rule, the number of edges could be replaced by the screen-space length of a stroke, thus yielding more balanced strokes.

4.4.7 Rendering

We now have all necessary information to render stylised strokes. One could devise many different strategies to translate the 3D vertex data into outlines. We will present two examples of stylised rendering modules we developed.

Sketchy Renderer

The first outline renderer we implemented uses lines as drawing primitives. To simulate a very sketchy look, it draws every stroke multiple times using jittering and alpha blending. It is based on an early prototype of the silhouette extraction algorithm that does not use subdivision for visibility determination (and therefore does not need to perform stroke simplification). To maintain temporal coherence, jittering uses precomputed random offsets that are stored with the mesh. The following list gives an overview of the factors that influence the final look of an outline drawing:

- **Colour:** The line colour also comprises an alpha value, that determines how transparent lines will be. A combination of low alpha values and overdrawing creates a style reminiscent of watercolour drawings.

- **Line width:** In 'stroke' mode, the line width for each stroke segment is calculated based on a minimum and a maximum line width and a scaling factor using the formula $w = \min\{maxWidth, minWidth * (scale)^{n-1}\}$ where n is the index of the segment in the current stroke. This generates strokes that grow from minimum to maximum width. In 'light' mode, line width depends on the angle between the light direction and the edge normal with small angles resulting in thinner lines. A third mode that linked line width to local curvature was also implemented but failed to deliver the desired

results because the algorithm to determine curvature was too simple and unstable.

- **Overdrawing:** A stronger sketchiness impression can be evoked by drawing every stroke multiple times with slightly perturbed anchor points. The artist has control of the number of rendering passes per stroke and the jitter mode. If jittering is enabled, a 3D offset is added to every vertex in a local space spanned by the edge tangent, the view-vector and the cross-product of these two. In 'random' mode, jittering is applied uniformly to all three axes, whereas in 'normal' mode the offset along the edge normal is limited to positive values . This mode creates nicer results if z-buffering is used for visibility testing, because moving a vertex along the negative normal vector causes it to overlap with shading and to be clipped.

- **Stroke settings:** As described earlier, long chains of silhouette edges are broken into smaller strokes using a smoothness constraint and a maximum stroke length.

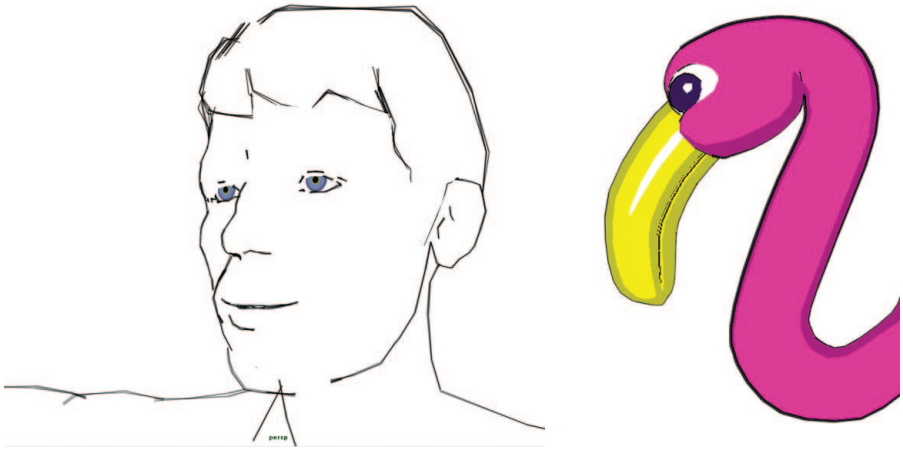
- **Visibility test:** As the early prototype of our visibility test algorithm did not use refinement, it always generated a very sketchy look. We therefore also included a standard z-buffer visibilty test that clips silhouette lines more precisely.

Figure 4.14 shows examples of the different visual styles that can be achieved using the sketchy renderer.

Textured Outlines

The second, more powerful rendering module is based on textured triangle strips. Every stroke segment is rendered as two triangles that are assigned a base colour and a stroke texture. Before we explain the details of the rendering process, we first give a list of properties that the algorithm should implement:

- **Line width:** The drawing algorithm should be able to create two different styles: In the first case, lines that are close to the viewer should be drawn thicker than lines that are further away. Additionally line width should be constrained by minimum and maximum values. The second style requires that all lines have the same width independently of their position in 3D space.



a b

Figure 4.14: Examples of the Sketchy Renderer: a) Overdrawing b) Long Strokes

- **Strokes:** To achieve a more natural look, the first and last segments of a stroke should use a different texture map from the intermediary segments. Furthermore strokes should contain no gaps between segments.

- **Performance:** Because silhouette extraction and postprocessing are expensive, rendering should be delegated as much as possible to the graphics hardware.

These requirements do not sound very demanding, but the problem is that they are stated in screen-space describing what the result of rendering should be like. Unfortunately all information the rendering module has is expressed in local object space. So in order to fulfill the screen-space constraints we need to transform our geometric data to screen-space, test the constraints and derive correction factors that are applied to the 3D data. This last step is only necessary because we do not have full control of the rendering pipeline (e.g. perspective division cannot be switched off). As we need to perform these computations for every vertex, we decided to implement them as a vertex program. We will now explain the general concept of the rendering algorithm and then explain step by step how this was achieved using vertex programs.

The drawing algorithm operates on pairs of 3D vertices and their associated tangent vectors. Given the two endpoints of a stroke segment it fits a (not necessarily rectangular) billboard through these points, orients it to face the viewer and adjusts its width according to the constraints described

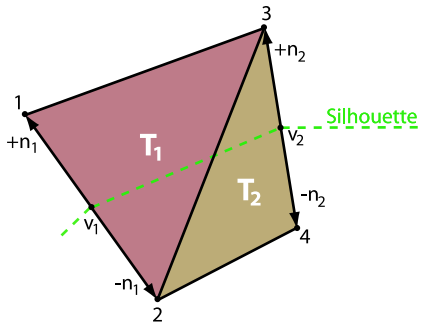


Figure 4.15: Illustration of a stroke segment billboard

above. The corners vertices of the billboard are obtained by adding the vertex normal resp. the negative vertex normal to the endpoints as illustrated in Figure 4.15. For a perfectly tesselated surface, the vertex normal would (by definition) lie in a plane that is parallel to the viewplane. Unfortunately this is not given for general meshes. To avoid distortions, we calculate a pseudo-normal by taking the cross-product of the edge tangent and the view vector which garuantees that the offset we apply to endpoints is parallel to the viewplane and that the billboard faces the viewer.

Now that we have determined the billboard vertices, we need to scale them to obtain the correct line width. This step is different for the fixed width mode than for variable width. The fixed width condition requires that all billboards have the same width on the screen. In pseudo-code this can be achieved by:

```
calculate (pseudo-)normal n
transform n to clip-space
normalise n
transform initial position p to clip-space
scale n by p.w
add n to p and store the result as final clip-space position
```

The only step that requires further explanations is scaling of the normal vector. If a perspective viewing setup is used, it is necessary to divide the clip-space position of a vertex by its w-component to correctly render foreshortening effects. If we omitted the scaling step, line width would depend on the distance of a line from the viewer, as points that are further away from the viewpoint are scaled down more by perspective division than points that are nearer. Scaling `n` by `p.w` ensures that the same screen-space offset is added to all vertices.

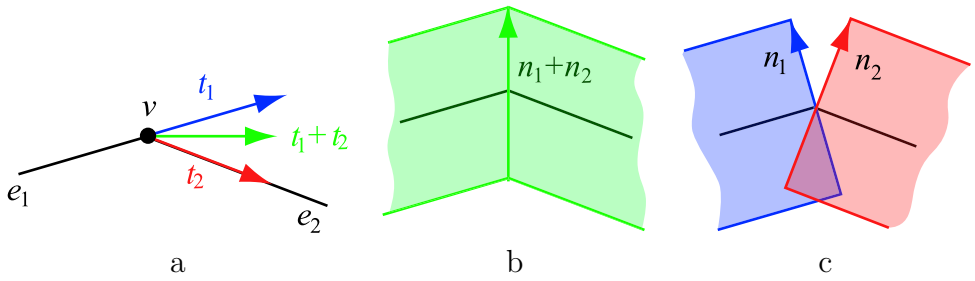
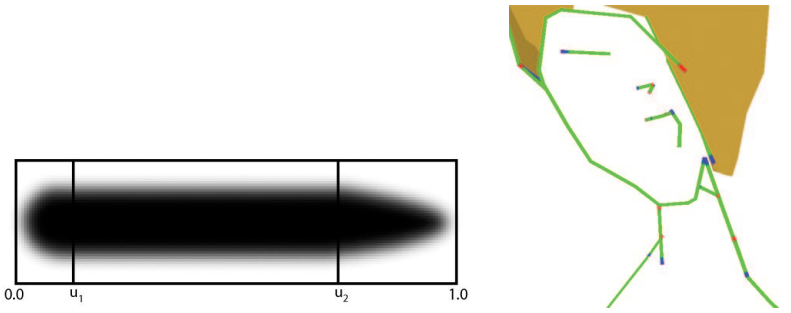


Figure 4.16: a) Diagram of per-vertex tangent vectors. b) Continuous egdes share the same normal vector. c) For broken edges, two different normal vectors are used.

We just saw that if we omit scaling in the above algorithm, lines that are nearer to the viewer will be drawn wider than those further away, thereby fulfilling the requirements for variable width strokes. Tests showed however that it is better to limit the width of a segment to a maximum or a minimum value, because else lines that are far away will have sub-pixel width values that cause rendering artifacts, and lines that are very close to the near clipping plane become too wide. We therefore only need to scale `n` if `p.w` is too large or too small. In pseudo-code this yields:

```
calculate (pseudo-)normal n
transform n to clip-space
normalise n
transform initial position p to clip-space
if p.w < minValue
  s = p.w / minValue
else if p.w > maxValue
  s = p.w / maxValue
else
  s = 1
scale n by s
add n to p and store the result as final clip-space position
```

So far, we have not explained how the vertex tangents and texture coordinates are calculated. Every inner vertex of a stroke is part of two line segments. We therefore store two different tangent vectors for every vertex that are colinear to the edges the vertex is connected to (see Figure 4.16a). We need these two vectors because tangents are treated differently for inner vertices of strokes and endpoints. We want to avoid gaps between inner line segments and thus need to use the same tangent for both billboards that a



a b

Figure 4.17: a) Example of a stroke texture b) Illustration of stroke start and end capping. Start segments are coloured red, inner segments green and end segments blue.

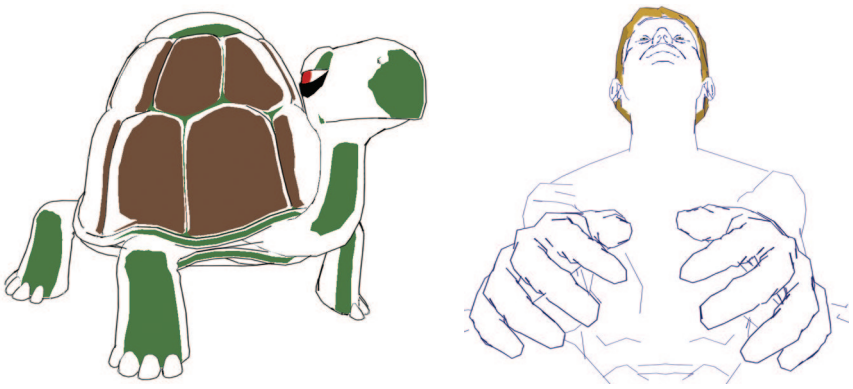
vertex is part of. If we simply use the average of the two vectors, the result looks similar to Figure 4.16b. For vertices that are part of two different strokes, we cannot use the same tangent for both strokes because the angle between the tangents can become very large which results in an average vector that is unsuitable for both segments. But as strokes are independent of each other, it is often even desired to have a gap between two adjacent strokes. We therefore use t_1 when drawing e_1 and t_2 for e_2 (see Figure 4.16c).

We want the first and last segments of a stroke to look different from the inner segments. We could achieve this by simply using three different textures, but in this case it would be difficult to adjust the texture maps to avoid cracks at the transition. We therefore decided to use a single that encodes a prototype of the stroke. As can be seen in Figure 4.17a, a stroke texture is composed of a tileable middle part and two stroke end parts. The rendering module simply submits $(0, 0)$ and $(0, 1)$ for the first two vertices of a segment if it is the start segment of a stroke and $(u_1, 0), (u_1, 1)$ if it is an inner segment. Symmetrically for the third and fourth vertex of a segment we use $(1, 0), (1, 1)$ for the last segment of a stroke and $(u_2, 0), (u_2, 1)$ else. Figure 4.17b illustrates this by using different colours for the three parts of the stroke texture.

This concludes the description of our geometric strokes algorithm. The full Vertex Programs described above can be found in Appendix A.2.

4.4.8 Examples and Discussion

Figure 4.18 shows examples of images created with the 'Textured Outlines' Renderer. In the following discussion we will restrict ourselves to this ren-



a b

Figure 4.18: Examples of textured outlines a) Turtle model with fixed stroke width b) Bumba model with variable stroke width

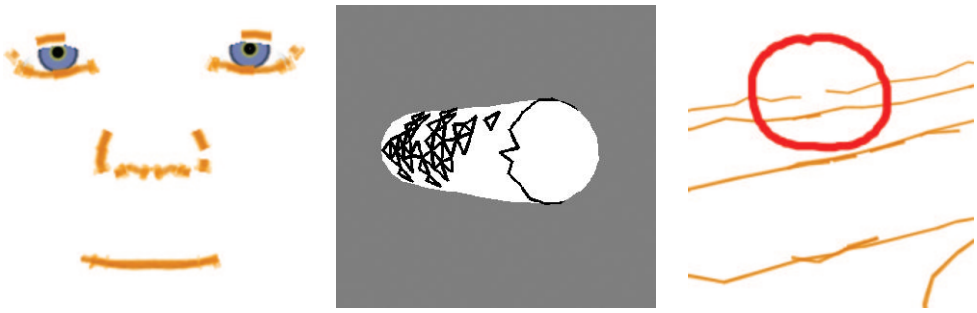
dering module as it is the more interesting one.

- **Performance:** Although many steps are necessary to render a single frame, rendering times are still suitable for real-time applications (see Section 4.5). The most expensive stage is silhouette extraction as it needs to check every mesh edge at least once. All subsequent postprocessing steps operate on a subset of of the mesh edges (only a small fraction of all mesh edges are silhouette edges for any given view direction). Especially if refinement for visibility determination is cut off after a few iterations (2-3 iterations have been sufficient in most cases), all postprocessing operations have only linear time requirements. Indeed it might be useful to add more postprocessing stages to eliminate some of the artifacts described below.

- **Stylistic Freedom:** Geometric strokes offer artists many opportunities to influence the rendering style. As all postprocessing operations are controlled by threshold values, adjusting these values changes the look of the rendered image. For instance the visibility test can create precise or sketchy strokes, and setting the maximum stroke length to a high value results in long strokes that look calmer than a series of very short strokes. Finally custom textures can be used to simulate a variety of different drawing techniques, from pencil to chalk or watercolour. One could also add the overdrawing and jittering methods from the sketchy renderer, yielding even more possible styles.

- **Rendering Artifacts:** Unfortunately the rendering methods we implemented generate a number of artifacts that are strongest for small, broad segments that lie in areas of the mesh with a complex surface structure (see Figure 4.19a). These artifacts are mainly due to the complicated spatial structure of silhouettes on triangle meshes. As can be seen in Figure 4.19b, in 3D space the silhouette can fold back on itself and what appears to be a single line when seen from the viewpoint turns out to consist of multiple lines than can be slightly offset with respect to each other. Especially when stroke textures with highly transparent areas are applied, the colour of a stroke changes if it is overdrawn multiple times in a small area only. Two different approaches can be used to overcome these problems. The highest possible image quality is achieved by combining geometric methods with image-space methods that first render the silhouette edges as simple lines that are then traced and drawn as strokes [44]. Although this technique produces very clean and smooth outlines, it is not suitable for real-time applications because accessing the framebuffer and tracing are expensive operations. Another way of getting rid of the artifacts is to perform more postprocessing operations on the 3D edges to further simplify the edge chain and remove all edges that overlap (see [33] for some examples of additional operators). Ideally we would like to arrive at edge chains that lie entirely in one plane and do not fold back on themselves.

- **Temporal Coherence:** If not enough refinement iterations are performed by the visibility test, some popping artifacts appear as the deformation of the mesh causes ambiguous edges to continually change their status from visible to invisible. This effect is mainly noticeable for long line segments and can be remedied by increasing the number of refinement steps. Nevertheless some popping persists, especially for short lines, that seems to originate from the fact that the number of pixels that are drawn by OpenGL for a line primitive depends on its orientation on the screen. Combined with the limited resolution of the depth buffer, this causes some edges that should be perfectly unambiguous to fail the threshold test under some viewing angles (see Figure 4.19c). Increasing the width of the lines used by the depth test helps, but cannot eliminate all classification errors.
  All other mesh operations have been adjusted to maintain interframe coherence. For instance we decided to implement a brute-force silhouette extraction algorithm, because randomisation would have caused the stroke starting- and endpoints to change from frame to frame even for static scenes.



a b

Figure 4.19: a) Rendering artifacts b) Illustration of the 3D structure of silhouette edges c) Silhouette edges that failed the Visibility test

- **Prerequisites:** Theoretically the algorithm can handle any kind of triangle mesh, but the best results are obtained for objects with very sharp corners or for very smooth, evenly tesselated objects. In the case of character animation, the usual problematic body regions (e.g. the eyes) are most affected by rendering artifacts.

- **Aesthetics:** Even though images rendered with textured outlines contain some ugly artifacts, it can be foreseen that this method can generate very pleasing results once these problems are eliminated. The biggest advantage of geometric methods is that they can be used to create a variety of different visual styles without the need to change the rendering code. Furthermore, antialiasing can be included into the stroke texture without additional costs by slightly fading the stroke contours.

4.5 Discussion

None of the three silhouette drawing techniques that we presented in this chapter is ideal for character animation applications.

The *Wireframe* rendering method has the advantage that it is very simple and can be implemented using only standard OpenGL features. It is therefore guaranteed that it will work with any 3D graphics board. Its main drawbacks are the poor image quality and the lack of stylistic freedom that makes the method uninteresting for pure outline drawings. Furthermore it fares best if scenes have only a limited depth range, because else the low resolution of the depth buffer introduces too many artifacts.

Algorithm	**Performance** (approx.)
Wireframe	45fps
Per-Pixel	55fps
Geometric	25fps

Table 4.1: Performance Comparison of the three algorithms. All measurements were executed on a 1.6GHz Athlon XP+ with a GeForce 3 Ti200 graphics board. The 3D character model contains 9423 faces and the rendering times include the animation overhead.

The *Per-Pixel* outlining technique is the fastest of the three (See table 4.5), but it only delivers visually pleasing results for highly refined meshes. As it would be too inefficient to simply use high-resolution meshes, one could experiment with mesh subdivision schemes that only refine mesh areas near the silhouette. If one managed to solve this problem, the technique would be the most interesting of the three for highly complex scenes (such as encountered in games).

Finally, the *Geometric* silhouette extraction and stylised drawing technique is by far the slowest, but it also offers most opportunities for creating a unique visual style. It still suffers from a number of artifacts, but it should be possible to avoid them by adding more postprocessing stages that filter out superfluous line segments. The advantage of this method is that it often works better for low-resolution models than for highly refined ones, which partly compensates the lower performance due to extensive geometric calculations.

Chapter 5

Discussion

In this chapter we are going to provide a critical assessment of the work that was done during this thesis and discuss directions for future research.

5.1 Critical Assessment

Our initial aim was to develop and implement NPR modules in the context of a real-time character animation system (Trick17) and to evaluate the rendering modules with respect to a number of criteria. The first part of the requirements was met, as we implemented three different shading modules and three silhouette drawing techniques. Most of these techniques were not completely new, but had to be adjusted for our purposes as they had only been designed for static meshes. The only problem that arose in this context was the depiction of motion. We had initially planned to implement algorithms for 3D motion lines [39], but the lack of appropriate animation sequences made it impossible to judge their usefulness.

The evaluation part proved to be more problematic. First of all the animation engine was still at a very early development stage and therefore lacked the flexibility that would have been necessary to evaluate the modules in a systematic way. The fact that only one virtual character could be animated using a small number of hard-wired animation sequences made it difficult to make general statements about the suitability of the rendering techniques. Furthermore the virtual character model was a very 'realistic' human model that was not appropriate for most rendering module. For instance the Cartoon Shader delivered much better results when it was used with more cartoony models, but as we had no animation data for these, we could only judge them as static models. In order to decide whether NPR techniques are better suited for storytelling purposes or in the context of

computer games, it would of course have been necessary to test them under real-world conditions.

Another problematic area was artwork. In order to evaluate the full aesthetic power of a rendering technique, it is necessary to let artists experiment with it and to refine and extend it based on their feedback. Although it is common practice for graphics researchers to create their own 'artwork' to illustrate their rendering algorithms, these images are usually not optimal, because standard models are used (such as the 'Stanford Bunny' [8]). Furthermore, the artistic skills of developers are usually less developed than those of a professional 3D artist. It would therefore have been useful to develop and evaluate the rendering algorithms in cooperation with a professional 3D artist, which was impossible, because no such person was available.

5.2 Future Work

The outlook we are going to present now, can be divided into two parts. The first is related to NPR techniques for character animation and the second part will treat the future of computer graphics in a more general context. For a discussion of specific aspects of the different rendering techniques we investigated, we refer to the corresponding sections in Chapters 3 and 4.

5.2.1 NPR Techniques for Character Animation

The rendering techniques we presented in this thesis can still be improved. Most are affected in some way or another by rendering *artifacts* or performance issues. Especially the geometric approach to stylised silhouettes might be worthwhile refining and extending. These issues are tightly related to limitations of the graphics hardware and one can expect a wealth of new approaches to be developed with every new generation of graphics boards.

So far, we have not considered rendering techniques that were developed especially for character animation. It would be interesting to investigate how the rendering style can support *animation*. Although some research groups have focused on the depiction of speed through the use of motion lines or artistic motion blur using techniques developed by comic artists [39], these ignore the unique properties of 3D graphics and are only useful for visualising fast motion.

At this point we are unable to decide whether NPR techniques are better suited for character animation than photorealistic techniques. In order to answer this question it would be necessary to conduct *controlled experiments*, where subjects are confronted with specific tasks. By varying the rendering

method and measuring the performance of subjects, it would be possible to judge which depiction style is appropriate for a given task. In the context of computer games, one could for instance measure properties such as overall performance, reaction time, learning effects etc., although one has to be very careful to choose expressive properties (which is why it would be useful to conduct this kind of experiment in cooperation with cognitive scientists).

5.2.2 The Future of Computer Depiction

We are now going to present some ideas about the direction that future research in Computer Depiction should take in order to exploit its full potential.

- **Rendering Techniques:** So far most efforts in NPR research have focused on the simulation of traditional painting and drawing techniques. Although this is probably a good starting point for exploring the strengths and weaknesses of this new medium, it should not be its ultimate goal. Most artistic techniques were developed to overcome the specific limitations of a given medium and can therefore not be mapped in a straightforward way to a different medium. For instance, hatching techniques do not work as well on a computer screen as they do on paper, because the resolution of a typical screen is too low. It would therefore be more rewarding to investigate the underlying principles of traditional artistic techniques and to develop new rendering methods based on these principles and taking into account the properties of digital processing and display devices. Another rich source of inspiration can be findings from cognitive psychology and especially research on perception. As we have seen in Section 2.1.2 we can learn a lot on how we perceive the world and images from this field and use this knowledge to develop more appropriate depiction methods for a given task.

- **Integration:** So far most research projects have focused on very specific aspects of depiction. In the case of cartoons for instance, we have rendering techniques that aim at imitating hard shading and outline drawings (see Chapter 3), animation techniques that simulate toon-style physics with rubbery deformation models [13] or modelling techniques based on view-dependent geometry [48]. Furthermore one could also investigate camera models that support a cartoony style. The next step is to put it all together and to explore the interaction of the different building blocks. In this way it would be possible to create a much more compelling experience, because depiction is not only about rendering, but includes all aspects involved in image generation.

- **Cooperation:** Developing new depiction methods is not only about solving technical problems, but also about aesthetics. As most computer graphics researchers are no artists, it is important to cooperate with visual artists that have at least a minimal background in computer depiction in order to refine existing techniques and to find inspiration for new ones. As the case of this project shows, it is not always easy to do so, but it can be very rewarding. Many 3D artists have excellent ideas for projects they would like to realise, but lack the necessary programming skills. If both worlds join their forces, the result will be new depiction techniques that are not only of theoretical interest.

- **Interaction:** Even though automation plays an important role in real-time applications, not all aspects of image generation can or should be automated. It is therefore important to develop tools that support the creative process on all levels. Today the workflow in most 3D graphics and animation packages is still strongly related to the underlying technology and does therefore not correspond to the way a designer (who usually has little knowledge of rendering algorithms) would like to work. This can be observed as well on the level of user interfaces that map properties directly to variables that control the renderer, as to the whole workflow imposed on the user by the toolkit. Some promising efforts have been undertaken in this area (see for instance [28]), but we are still a long way from ideal tools. Even experimental tools should also provide an intuitive interface if they are to be evaluated by artists (see previous point).

- **Theoretical Framework:** So far, we do not know much about the digital medium we work in. In order to fully exploit the potential of computer depiction, we need to identify what it is that makes digital images and the digital creation process unique. This is not only of theoretical interest, as it can help to guide research efforts and to avoid wasting resources on projects that are doomed to fail because of inherent limitations. Durand has recently started a theoretical discussion of computer depiction that is still at an early stage, but already now it raises a number of interesting questions that should be further investigated [18].

5.3 Summary

In this thesis, we investigated the application of non-photorealistic rendering techniques to real-time character animation. Based on established render-

ing methods, we developed three non-photorealistic shading algorithms and three outline drawing methods that we implemented in OpenGL and integrated into the Trick17 system. We evaluated the rendering techniques with respect to a number of criteria and a strong focus on real-time character animation and presented ideas for extending and refining them. Furthermore we embedded our work in the general context of computer depiction and included findings from related fields such as fine arts and psychology. Finally we hinted at directions in which our work could be continued.

Appendix A

Vertex Programs

A.1 Per-Pixel Silhouettes

The following listing contains the assembler code for calculating texture coordinates for toon shading and silhouette drawing as described in 4.3.2.

```
!!VP1.0
// Input Registers
// c[0-3]  Modelview Matrix
// c[4-7]  Projection Matrix
// c[8-11] Inverse-Transpose Modelview Matrix
// c[20]   Light Direction
// c[21]   Constants: 0.5, 0.5, 0, 0

// Output Registers
// TEX0: 1D Silhouette Texture
// TEX1: 1D Shading Texture
// TEX2: 2D Detail Texture

// XForm Pos to EyeSpace (Modelview) into R0
DP4 R0.x, c[0], v[OPOS];
DP4 R0.y, c[1], v[OPOS];
DP4 R0.z, c[2], v[OPOS];
DP4 R0.w, c[3], v[OPOS];

// XForm Pos to ClipSpace into o[HPOS]
DP4 o[HPOS].x, c[4], R0;
DP4 o[HPOS].y, c[5], R0;
DP4 o[HPOS].z, c[6], R0;
```

```
DP4 o[HPOS].w, c[7], R0;

// XForm Normal to EyeSpace (IT-Modelview) into R1
DP3 R1.x, c[8], v[NRML];
DP3 R1.y, c[9], v[NRML];
DP3 R1.z, c[10], v[NRML];

// Normalise Normal
DP3 R1.w, R1, R1;
RSQ R1.w, R1.w;
MUL R1.xyz, R1, R1.w;

// Normalise Position ( = View Direction )
DP3 R0.w, R0, R0;
RSQ R0.w, R0.w;
MUL R0.xyz, R0, R0.w;

// Calculate TexCoord 1 as DotProduct of LightDir and Normal
// And Remap to [0,1]
DP3 R2, R1, c[20];
MAD o[TEX1].x,R2,c[21].x,c[21].y;

// Calculate TexCoord 0 as DotProduct of ViewDir and Normal
// ViewDir = - Pos
// Remapping not necessary as negative values are clamped to
// Silhouette Texels
DP3 o[TEX0].x, -R0, R1;

// Copy TexCoords for Unit 2
MOV o[TEX2], v[TEX2];

END
```

A.2 Geometric Strokes

In this section we present the vertex programs that were used to draw outlines as textured billboards (see section 4.4 for details). The first listing shows the assembler code for the variable line width mode:

```
!!VP1.0

// Input Register Mapping
// c[0-3]  Modelview Matrix
// c[4-7]  Projection Matrix
// c[8-11] Inverse-Transpose Modelview Matrix
// c[20]   0, Width, 0, 0
// c[21]   1, AspectRatio, 0, 0
// c[22]   MinWidth, MinWidth, MinWidth, 0
// c[23]   MaxWidth, MaxWidth, MaxWidth, 0
// c[24]   1, 1, 1, 0

// XForm Pos to EyeSpace (Modelview) into R0
DP4 R0.x, c[0], v[OPOS];
DP4 R0.y, c[1], v[OPOS];
DP4 R0.z, c[2], v[OPOS];
DP4 R0.w, c[3], v[OPOS];

// XForm Pos to ClipSpace into R2
DP4 R2.x, c[4], R0;
DP4 R2.y, c[5], R0;
DP4 R2.z, c[6], R0;
DP4 R2.w, c[7], R0;

// Normalise ES-Position -> ViewVector
DP3 R0.w, R0, R0;
RSQ R0.w, R0.w;
MUL R0.xyz, R0, R0.w;

// XForm Tangent to EyeSpace (IT-Modelview) into R1
DP3 R1.x, c[8],  v[NRML];
DP3 R1.y, c[9],  v[NRML];
DP3 R1.z, c[10], v[NRML];
```

```
// Normalise Tangent
DP3 R1.w, R1, R1;
RSQ R1.w, R1.w;
MUL R1.xyz, R1, R1.w;

// Normal = ViewVector x Tangent in R3
//        = Pos(ES) x Tangent
MUL R3, R0.zxyw, R1.yzxw;
MAD R3, R0.yzxw, R1.zxyw, -R3;

// Compensate Aspect Ratio and set z to 0
MUL R3, R3, c[21];

// Normalise Normal
DP3 R3.w, R3, R3;
RSQ R3.w, R3.w;
MUL R3.xyz, R3, R3.w;

// Min Line Width
// If w of vertex is larger than a max. value
// compensate by multiplication factor
MUL R4, R2.wwww, c[22];
MAX R4, R4, c[24];
MUL R3.xy, R3, R4;

// Max Line Width
MUL R4, R2.wwww, c[23];
MIN R4, R4, c[24];
MUL R3.xy, R3, R4;

// Rescale Normal to s (c[20].y)
MUL R3.xy, R3, c[20].y;

// Add XFormed Normal to position in ClipSpace
// Not precise because normal is calculated in EyeSpace
ADD R2.xy, R2, R3;
MOV o[HPOS], R2;

// Copy TexCoords for Unit 0
MOV o[TEX0], v[TEX0];
```

```
// Copy Stroke Colour
MOV o[COL0], v[COL0];

END
```

The second listing shows the vertex program code for rendering geometric strokes with a fixed screen-space width:

```
!!VP1.0

// Input Register Mapping
// c[0-3]  Modelview Matrix
// c[4-7]  Projection Matrix
// c[8-11] Inverse-Transpose Modelview Matrix
// c[20]   Constants: 0, Width, 0, 0
// c[21]   Constants: 1, AspectRatio, 0, 0

// XForm Pos to EyeSpace (Modelview) into R0
DP4 R0.x, c[0], v[OPOS];
DP4 R0.y, c[1], v[OPOS];
DP4 R0.z, c[2], v[OPOS];
DP4 R0.w, c[3], v[OPOS];

// XForm Pos to ClipSpace into R2
DP4 R2.x, c[4], R0;
DP4 R2.y, c[5], R0;
DP4 R2.z, c[6], R0;
DP4 R2.w, c[7], R0;

// Normalise ES-Position -> ViewVector
DP3 R0.w, R0, R0;
RSQ R0.w, R0.w;
MUL R0.xyz, R0, R0.w;

// XForm Tangent to EyeSpace (IT-Modelview) into R1
DP3 R1.x, c[8],  v[NRML];
DP3 R1.y, c[9],  v[NRML];
DP3 R1.z, c[10], v[NRML];

// Normalise Tangent-Vector
DP3 R1.w, R1, R1;
```

```
RSQ R1.w, R1.w;
MUL R1.xyz, R1, R1.w;

// Normal = ViewVector x Tangent in R3
//        = Pos(ES) x Tangent
MUL R3, R0.zxyw, R1.yzxw;
MAD R3, R0.yzxw, R1.zxyw, -R3;

// Compensate Aspect Ratio and set z to 0
MUL R3, R3, c[21];

// Normalise Normal
DP3 R3.w, R3, R3;
RSQ R3.w, R3.w;
MUL R3.xyz, R3, R3.w;

// Rescale Normal to s (c[20].y)
MUL R3.xy, R3, c[20].y;
// Compensate Perspective Divide
MUL R3.xy, R3, R2.w;

// Add XFormed Normal to position in ClipSpace
// Not precise because normal is calculated in EyeSpace
ADD R2.xy, R2, R3;
MOV o[HPOS], R2;

// Copy TexCoords for Unit 0
MOV o[TEX0], v[TEX0];

// Copy Stroke Colour
MOV o[COL0], v[COL0];

END
```

Appendix B

Artistic Examples

While experimenting with our non-photorealistic rendering modules we generated a number of images that go beyond simple technical demonstrations. In this appendix we present some of the results together with a small description of how they were created.

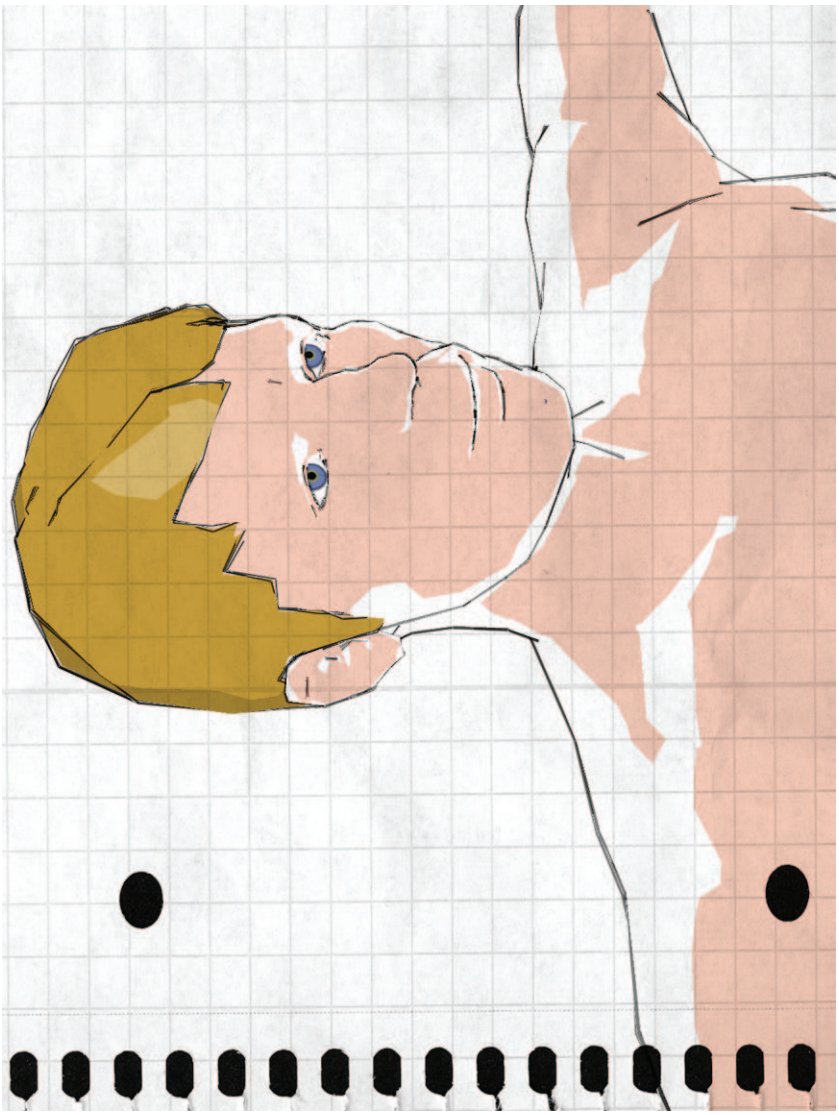


Figure B.1: Sketchy Portrait. Screenshot from the Trick17 editor. The outlines were rendered with the Sketchy Lines renderer using translucent lines overdrawn multiple times with varying line width. The 'watercolour' style is achieved by combining Toon Shading with a scanned paper texture blended with the rendered image in real-time

.

Figure B.2: Propaganda. Screenshot from the Trick17 editor post-processed in Adobe Photoshop and Adobe Illustrator. The outlines were created with the Wireframe renderer. The shading on the shell uses Textured Shading (see Figure B.5).

Figure B.3: The Scribbling on the Wall. Composition of screen shots from the Animation Engine, retouched in Adobe Photoshop. The outlines were created by the Sketchy Lines renderer (Geometric Method) with transparent shading.

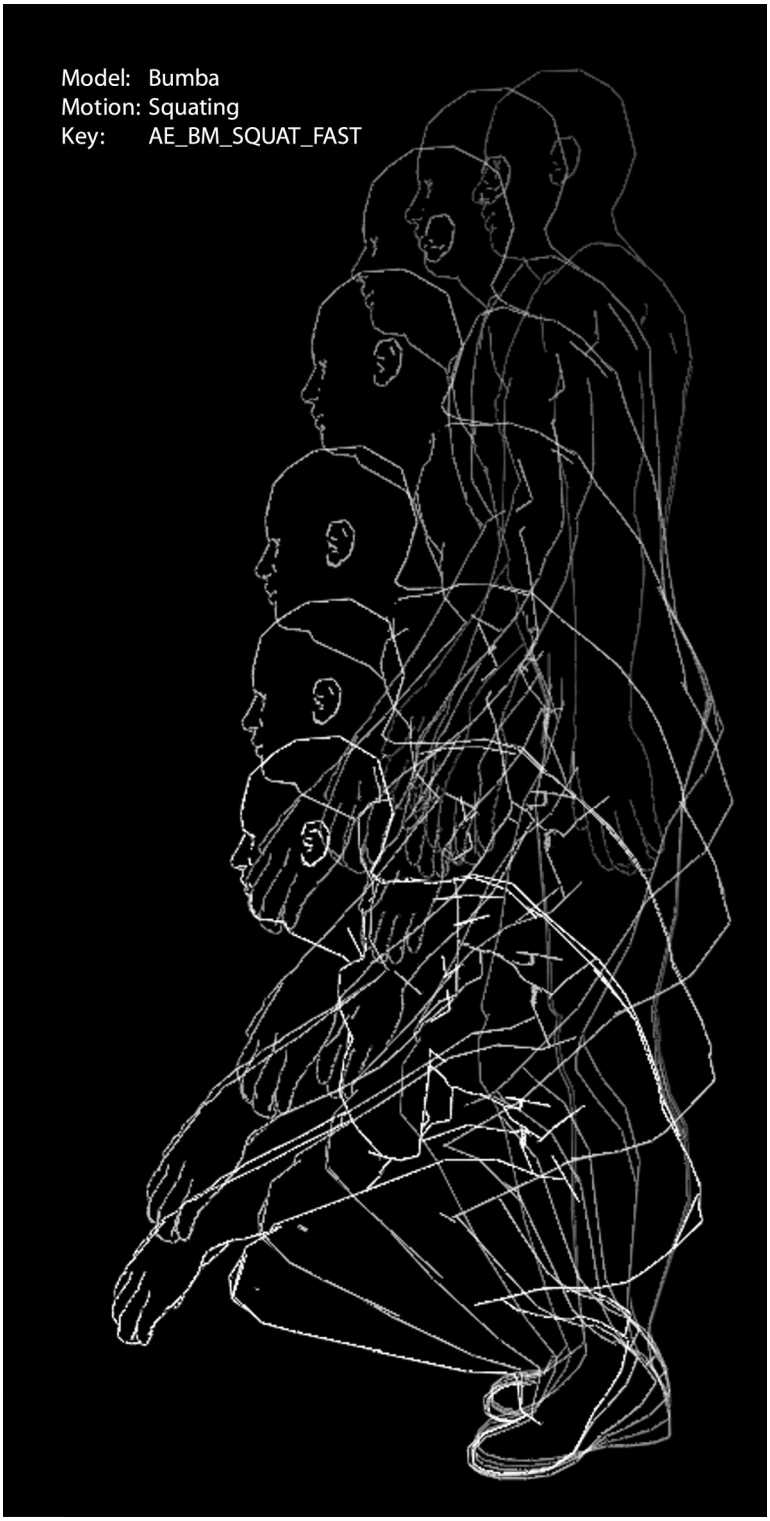


Figure B.4: Motion Study. Composition of 7 frames from a movie rendered in real-time by the Animation Engine and post-processed with Adobe Photoshop. The outlines were rendered with the Wireframe technique without shading.

Figure B.5: Detail from the turtle model used in Figure B.2. The camo pattern is used as a light texture and the shadow texture contains a line pattern with transparency (this causes the camo pattern to shine through between the strokes).

List of Figures

2.1 The power of expectation 7
2.2 Rorschach-Test example 8
2.3 Examples of Gestalt principles 9
2.4 Difficulties in shape perception 10
2.5 The visual pipeline according to Marr 11
2.6 Influence of high-level knowledge on shape perception 12
2.7 Can this image be considered as photorealistic? 14
2.8 Perspective distortion and perception 16
2.9 Progression from concrete to abstract depiction of a face . . . 17
2.10 Extrinsic vs. Intrinsic properties 18
2.11 Example of 12th century art 20
2.12 Artwork by Dürer and Munch 21
2.13 Examples of images created with Piranesi 25
2.14 Intercepting the OpenGL command stream 26
2.15 Examples of different HijackGL Rendering modules 27
2.16 Spirit: Stallion of the Cimarron 28
2.17 The OpenGL Rendering Pipeline 30
2.18 Standard OpenGL texture combination 31
2.19 Overview of the Register Combiner Architecture 32
2.20 Internal structure of a general combiner 33
2.21 Morphing for facial animation 37
2.22 Example of a virtual character 38
2.23 Geometric artifacts caused by extreme deformation 40
2.24 Structure of the Trick17 Animation Engine 40
2.25 The user interface of the Trick17 Editor 41
2.26 The workflow in the Trick17 system. 43

3.1 Examples of artistic shading 45
3.2 Comparison of Gouraud and cartoon shading 46
3.3 Vertex- vs. Pixel-Shading for Cartoon Shading 48
3.4 1D Texture map for hard shading 49
3.5 Cartoon Shading Material Interface in Trick17 52

3.6 Examples of Cartoon Shading 53
3.7 Antialiasing for Cartoon Shading 53
3.8 Cartoon shading with inner outlines 54
3.9 Detail Texture Composition in Textured Shading 56
3.10 1D Shading Texture for Textured Shading 56
3.11 2-Pass Rendering Algorithm for Textured Shading 57
3.12 Comparison a different transition functions 57
3.13 Example of a non-symmetrical transition 58
3.14 Examples of Textured Shading 58
3.15 Example of a halftone screen used in printing presses. 60
3.16 Example of halftone patterns used by Freudenberg 61
3.17 Halftone shading using a brick pattern as a halftone screen . . 62
3.18 Examples of gamma functions 64
3.19 Examples of different halftone primitives 66
3.20 The influence of distance on the halftoning effect 67
3.21 Halftone Shading Artifacts 68

4.1 Examples of outline drawings 70
4.2 Image-Space silhouette extraction 71
4.3 Concept of wireframe silhouette drawing 73
4.4 The user interface for wireframe outlines in Trick17 75
4.5 Examples of wireframe silhouettes 76
4.6 Rendering artifacts of wireframe silhouettes 77
4.7 Concept of per-pixel outlines 78
4.8 OpenGL implementation of per-pixel outlines 79
4.9 Rendering artifacts of per-pixel silhouettes 80
4.10 Program flow for the geometric strokes algorithm 82
4.11 Ambiguities of geometric silhouettes 83
4.12 Visibility Test issues 87
4.13 Edge subdivision for visibility determination 89
4.14 Examples of the Sketchy Renderer 92
4.15 Illustration of a stroke segment billboard 93
4.16 Strokes and normal vectors 94
4.17 Stroke textures 95
4.18 Examples of textured outlines 96
4.19 Rendering artifacts of the Geometric Outlines method 98

B.1 Sketchy Portrait 114
B.2 Propaganda 115
B.3 The Scribbling on the Wall 116
B.4 Motion Study 117

B.5 Detail of the model used in 'Propaganda' 118

Bibliography

[1] id Software. WebSite: www.idsoftware.com, December 2002.

[2] Laboratory for Mixed Realities. Website: www.lmr.khm.de, December 2002.

[3] Microsoft DirectX.
Official Website: www.microsoft.com/windows/directx, November 2002.

[4] OpenGL Architecture Review Board. Official Website: www.opengl.org, December 2002.

[5] OpenGL Utility Library (GLU).
Official Website: www.opengl.org/developers/documentation/glx.html, November 2002.

[6] OpenGL Utility Toolkit (GLUT).
Official Website: www.opengl.org/developers/documentation/glut/index.html, November 2002.

[7] Piranesi Homepage. www.informatix.co.uk/piranesi/index.shtml, November 2002.

[8] The Stanford 3D Scanning Repository.
Web-Site: graphics.stanford.edu/data/3Dscanrep/, December 2002.

[9] Arthur Appel. The notion of quantitative invisibility and the machine rendering of solids. *Proc. ACM Natl. Mtg.*, page 387, 1967. Held in Washington, DC.

[10] R. Arnheim. *Kunst und Sehen.* de Gruyter, Berlin, expanded and revised edition, 1987.

[11] Gernot Böhme. *Theorie des Bildes.* Wilhelm Finkel Verlag, München, 1999.

[12] David Buttgereit, Bernd Hentschel, Alexander Hornung, and Jerome Thoma. S.W.A.R.M. Echtzeitsimulation und 3D-Visualisierung komplexen Schwarmverhaltens. In *GI Informatiktage 2002 Tagungsband*, Bad Schüssenried, Germany, 2003.

[13] Stephen Chenney, Mark Pingel, Rob Iverson, and Marcin Szymanski. Simulating Cartoon Style Animation. In *Proceedings of NPAR 2002, International Symposium on Non Photorealistic Animation and Rendering (Annecy, France, June2002)*, 2002.

[14] Johan Claes, Fabian Di Fiore, Gert Vansichem, and Frank Van Reed. Fast 3D Cartoon Rendering with Improved Quality by Exploiting Graphics Hardware. In *Proceedings of Image and Vision Computing, New Zealand 2001*, pages 13–18, November 2001.

[15] Doug Cooper. 2D/3D Hybrid Character Animation on Spirit. In *SIGGRAPH'02 Sketches*, 2002.

[16] Derek Cornish, Andrea Rowan, and David Luebke. View-Dependent Particles for Interactive Non-Photorealistic Rendering. In B. Watson and John W. Buchanan, editors, *Proceedings of Graphics Interface 2001*, pages 151–158, 2001.

[17] Philippe Decaudin. Rendu de scènes 3D imitant le style "'dessin animé"'. Technical Report 2919, INRIA Rocquencourt, 1996.

[18] Fredo Durand. An Invitation to Discuss Computer Depiction. In *Proceedings of NPAR 2002, International Symposium on Non Photorealistic Animation and Rendering (Annecy, France, June2002)*, 2002.

[19] Gabriele Knappe (editor). Hyperrealistische Fantasie. *Digital Production*, (3):22–39, 2001.

[20] J. D. Foley et al. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2nd edition, 1990.

[21] Cass Everitt. One-Pass Silhouette Rendering on the GeForce and GeForce2. Nvidia Online Developer Resources (developer.nvidia.com), November 2002.

[22] Andreas Feininger. *Die hohe Schule der Fotografie*. Heyne Verlag, 1982.

[23] Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Real-Time Halftoning: A Primitive for Non-Photorealistic Rendering. In *Proceedings of the 13th Eurographics Workshop on Rendering*, Pisa, Italy, 2002.

[24] E. B. Goldstein. *Wahrnehmungspsychologie*. Spektrum Verlag, Heidelberg, 2nd edition, 1997.

[25] E. H. Gombrich. *Art and Illusion: A Study in the Psychology of Pictorial Representation*. Princeton University Press, Princeton, 11th edition, 2000.

[26] Amy A. Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. In Michael Cohen, editor, *Proceedings of SIGGRAPH'98 (Orlando, July 1998)*, pages 447–452, New York, 1998. ACM SIGGRAPH.

[27] Bruce Gooch. Ramachandran and Hirstein's Neurological Theories of Aesthetic for Computer Graphics. In *SIGGRAPH'02 Course Notes. Course on Perceptual and Artistic Principles for Effective Computer Depiction*, pages 193–204. 2002.

[28] Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. Creating Non-Photorealistic Images the Designer's Way. In *Proceedings of NPAR 2002*, Annecy, France, 2002.

[29] Klaus Herding. Realismus. In Werner Busch, editor, *Funkkolleg Kunst: Eine Geschichte der Kunst im Wandel ihrer Funktion*. Piper, München, 1987.

[30] Aaron Hertzmann. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In Stuart Green, editor, *SIGGRAPH'99 Course Notes. Course on Non-Photorelistic Rendering*, chapter 7. New York, 1999.

[31] Aaron Hertzmann and Denis Zorin. Illustrating Smooth Surfaces. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000 (New Orleans, July 2000)*, pages 517–526, New York, 2000. ACM SIGGRAPH.

[32] Jessica K .Hodgins, James F. O'Brian, and Jack Tumblin. Perception of Human Motion With Different Geometric Models. *IEEE Transactions on Visualization and Computer Graphics*, 4(4), October-December 1998.

[33] Tobias Isenberg, Nick Halper, and Thomas Strothotte. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Stylised Strokes. *Eurographics 2002*, 21(3), 2002.

[34] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *Non-Photorealistic Animation and Rendering 2000 (NPAR '00)*, Annecy, France, June 5-7,2000.

[35] Jeff Lander. Return to Cartoon Central: Adding Texture to a Non-Photorealistic Renderer. *Game Developer Magazine*, August 2000.

[36] Jeff Lander. Shades of Disney: Opaquing a 3D World. *Game Developer Magazine*, March 2000.

[37] Jeff Lander. Under the shade of the rendering tree. *Game Developer Magazine*, February 2000.

[38] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. In Turner Whitted, editor, *Proceedings of SIGGRAPH'97 (Los Angeles, August 1997)*, pages 415–420. ACM SIGGRAPH, 1997.

[39] Maic Masuch, Stefan Schlechtweg, and Ronny Schulz. Speedlines: Depicting Motion in Motionless Pictures. In *SIGGRAPH'99 Conference Abstracts and Applications*, page 277, New York, 1999. ACM SIGGRAPH.

[40] Scott McCloud. *Understanding Comics*. HarperPerennial, New York, 1st edition, 1993.

[41] Barbara J. Meier. Painterly Rendering for Animation. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH'96 (New Orleans, August 1996)*, pages 477–484, New York, 1996. ACM SIGGRAPH.

[42] Jason L. Mitchell. Radeon 9700 Shading. In *SIGGRAPH 2002 Course Notes, Course 17: State of the Art in Hardware Shading*, chapter 3. 2002.

[43] Alex Mohr and Michael Gleicher. HijackGL: Reconstructing from Streams for Stylised Rendering. In *Proceedings of NPAR2002*, 2002.

[44] J. D. Northrup and Lee Markosian. Artistic Silhouettes: A Hybrid Approach. In *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 2000)*, pages 31–37, New York, 2000. ACM.

[45] Victor Ostromoukhov and Roger D. Hersch. Artistic screening. *Proceedings of SIGGRAPH 95*, pages 219–228, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[46] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped Textures. In *Proceedings of SIGGRAPH 2000*, pages 465–470, 2000.

[47] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. *Proceedings of SIGGRAPH 2001*, pages 579–584, August 2001. ISBN 1-58113-292-1.

[48] Paul Rademacher. View-dependent geometry. *Proceedings of SIGGRAPH 99*, pages 439–446, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[49] Vilayanur S. Ramachandran. Perceiving Shape from Shading. *Scientific American*, pages 76–83, August 1988.

[50] Ashu Rege. Occlusion (HP and NV Extensions) GDC2002 presentation. Nvidia Online Developer Resources (developer.nvidia.com).

[51] J. Rossignac and M. van Emmerik. Hidden contours on a frame-buffer. In *Proceedings of the 7th Eurographics Workshop on Computer Graphics Hardware*, pages 188–204, Cambridge, UK, September 1992.

[52] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 197–206, August 1990.

[53] Georg Schmidt. *Umgang mit Kunst*. Walter Verlag, Olten, 1946.

[54] Georg Schmidt. *Kleine Geschichte der modernen Malerei*. Friedrich Reinhardt Verlag, Basel, 1982.

[55] R. L. Solso. *Cognitive Psychology*. Harcourt Brace Jovanovich, New York, 1st edition, 1979.

[56] John Spitzer. Texture Compositing with Register Combiners. Nvidia Online Developer Resources (developer.nvidia.com).

[57] Daniel Teece. *Three Dimensional Interactive Non-Photorealistic Rendering*. PhD thesis, University of Sheffield, England, 1998.

[58] Oleg Veryovka. Animation with Threshold Textures. In *Proceedings Graphics Interface*, pages 9–16, Calgary, Alberta, May 2002.

[59] Matthew Webb, Adam Finkelstein Emil Praun, and Hugues Hoppe. Fine Tone Control in Hardware Hatching. In *Proceedings of the NPAR 2002*, 2002.

[60] Holger Winnemöller and Shaun Bangay. Geometric Approximations Towards Free Specular Comic Reflections. *Eurographics*, 21(3), 2002.

[61] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1999.

[62] Chris Wynn. Implementing Bump-Mapping using Register Combiners. Nvidia Online Developer Resources (developer.nvidia.com).

[63] Chris Wynn. Vertex Programs. Nvidia Online Developer Resources (developer.nvidia.com).