Integration of Non-Photorealistic Rendering Techniques for 3D Models in Processing

Diploma Thesis

submitted by Katrin Lang
matriculation number: 215854

in partial fulfillment of the requirements for a Diploma Degree in Computer Science
Technical University of Berlin

March 24, 2009

Advisors:

Marc Alexa
Professor of Computer Graphics
Faculty of Electrical Engineering and Computer Science
Technical University of Berlin

Adam Finkelstein
Associate Professor of Computer Graphics
Computer Science Department
Princeton University
Abstract

This thesis presents the implementation of a non-photorealistic renderer for the processing API (Application Programming Interface).

For decades, the goal of traditional computer graphics has been to create artificial images of simulated 3D environments that look precisely like a high-quality photograph of the real world. Yet, this may result in an “overkill” of visual information, when in many cases abstraction is the more appropriate way to convey shape and meaning. Examples include technical, scientific, medical and narrative illustration, computer aided design, architecture, and computational art.

In contrast to photorealistic rendering, the branch of computer graphics called NPR (Non-Photorealistic Rendering) is inspired by styles derived from painting, drawing, and animated cartoons. Several of these techniques have made their appearance in video games and movies, most prominent among them cel or “toon” shading, and have been integrated into professional commercial rendering packages.

However, the state of the art in NPR exists only in research papers and experimental academic implementations. There are virtually no freely available applications practitioners could use to experiment with such techniques.

Conversely, Processing lacks the ability to create meaningful abstractions in 3D. Although it allows for a wide range of expressive styles for digital art in 2D, 3D techniques are for the most part limited to gouraud shading.

Being an open source platform with a wide community and offering a comprehensive and easy-to-use graphics API, Processing is the ideal platform for making latest developments by researchers in non-photorealistic rendering available to designers and artists.

The Processing NPR renderer presented in this thesis is completely hardware based, as opposed to the original Processing renderer, but the interface remains the unaltered Processing language. It comes with a comprehensive set of NPR algorithms for shading, shadows and feature lines, which are carefully designed for real-time applications. Moreover, however, this thesis provides a framework of NPR data structures and rendering infrastructure that allows NPR researchers to plug in their algorithms in order to gain valuable feedback by users and create a synergy effect between researchers and practitioners.
# Contents

1 Introduction 2
   1.1 Non-Photorealistic Rendering ............................. 2
      1.1.1 Photorealism ......................................... 6
      1.1.2 Comprehensibility .................................... 7
      1.1.3 Computer Art .......................................... 8
      1.1.4 Input for NPR .......................................... 9
      1.1.5 Interactive vs. Deterministic Techniques.......... 11
      1.1.6 Further Reading ....................................... 12
   1.2 Processing .................................................. 12
      1.2.1 Fields of Application ................................. 14
      1.2.2 The Processing API and Development Environment .... 16
      1.2.3 Licensing ............................................. 16
      1.2.4 Processing and Java .................................. 18
      1.2.5 Further Reading ....................................... 19
   1.3 Requirements for a Processing NPR renderer ............ 21
      1.3.1 Integration with the Processing Language .......... 22
      1.3.2 The Non-Photorealistic Rendering API .............. 23
      1.3.3 Documentation ......................................... 24
      1.3.4 Source .................................................. 24
      1.3.5 Web Page ............................................... 24
      1.3.6 Examples ................................................ 26
      1.3.7 Performance issues .................................... 27
      1.3.8 Compatibility ......................................... 27

2 Architecture 28

3 NPR Algorithms 32
   3.1 Shading ..................................................... 32
      3.1.1 Cel Shading ............................................ 33
      3.1.2 Using Look-up-Tables ................................ 33
      3.1.3 Results ................................................. 33
   3.2 Shadows ..................................................... 33
      3.2.1 Shadow Mapping ....................................... 33
      3.2.2 Stylization of Shadows ................................ 33
   3.3 Line Drawings ............................................... 33
      3.3.1 Image Space Algorithms .............................. 33
      3.3.2 Hybrid Algorithms .................................... 66
      3.3.3 Object Space Algorithms .............................. 71
4 Example Program

5 API

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Shading</td>
<td>74</td>
</tr>
<tr>
<td>5.1.1 Look-up Table</td>
<td>74</td>
</tr>
<tr>
<td>5.1.2 Diffuse Shader</td>
<td>76</td>
</tr>
<tr>
<td>5.2 Shadows</td>
<td>76</td>
</tr>
<tr>
<td>5.2.1 Dual Map Shadows</td>
<td>77</td>
</tr>
<tr>
<td>5.3 Line Drawings</td>
<td>77</td>
</tr>
<tr>
<td>5.3.1 Sobel edge detection</td>
<td>79</td>
</tr>
<tr>
<td>5.3.2 Suggestive Contours</td>
<td>80</td>
</tr>
<tr>
<td>5.3.3 Wireframe Contours</td>
<td>80</td>
</tr>
<tr>
<td>5.3.4 Offset Contours</td>
<td>80</td>
</tr>
<tr>
<td>5.3.5 Boxy Ridges</td>
<td>81</td>
</tr>
<tr>
<td>5.3.6 Boxy Valleys</td>
<td>81</td>
</tr>
<tr>
<td>5.4 Developing Custom Algorithms</td>
<td>82</td>
</tr>
<tr>
<td>5.4.1 GLSL (GL Shading Language) API</td>
<td>82</td>
</tr>
<tr>
<td>5.4.2 Rendering to Textures</td>
<td>82</td>
</tr>
</tbody>
</table>

6 Performance

7 Further Work

8 Conclusion
# List of Figures

1.1 Medical illustration - generated with Processing NPR .......... 4
1.2 Technical illustration - generated with Processing NPR .......... 4
1.3 Comic style rendering - generated with Processing NPR .......... 5
1.4 "Charcoal" Drawing - generated with Processing NPR .......... 5
1.5 The Uncanny Valley (figure taken from Wikipedia ............... 7
1.6 The Processing Development Environment ....................... 17

2.1 Processing Coordinate System: The x axis is pointing to the right, the y axis is pointing downward, and the z axis is pointing towards the viewer ........................................... 31

3.1 There are three classes of NPR algorithms implemented: Shading, Shadows, and Line Drawings .......................... 32
3.2 Class hierarchy of all algorithms implemented .................... 33
3.3 Knot - rendered with unaltered diffuse shading ................. 34
3.4 Knot - rendered with simulated ambient light .................... 34
3.5 Torus - rendered with Cel Shading ................................. 34
3.6 Blade - rendered with sharp specular highlight .................. 35
3.7 Knot - simulated metal ............................................. 35
3.8 Cubehole - Image Space Sobel edge detection .................... 36
3.9 Cubehole - Image Space Sobel edge detection and 3x3 dilatation filter with threshold 1 ................................. 36
3.10 Hippo - Image Space Sobel edge detection ....................... 37
3.11 Hippo - Image Space Sobel edge detection with Smoothstep Interpolation ............................................. 37
3.12 Hippo - Image Space Sobel edge detection with Smoothstep interpolation and 3x3 Gaussian smoothing. Source: ? ........ 38
3.13 Hippo - Image Space Sobel edge detection with Smoothstep interpolation and 5x5 Gaussian smoothing. Source: ? ........ 38
3.14 Elephant - Image Space Sobel edge detection .................... 39
3.15 Elephant - Image Space Sobel edge detection with Smoothstep interpolation ............................................. 39
3.16 Elephant - Image Space Sobel edge detection with Smoothstep interpolation and 3x3 Gaussian smoothing ................ 40
3.17 Elephant - Image Space Sobel edge detection with smoothstep interpolation and 5x5 gaussian smoothing .................. 40
3.18 Rockerarm - Image Space Sobel edge detection with Smoothstep interpolation ............................................. 41
3.19 Rockerarm - Image Space Sobel edge detection with Smoothstep interpolation medium steep curve ................................. 41
3.20 Rockerarm - Image Space Sobel edge detection with smoothstep interpolation steep curve ........................................ 41
3.21 Sandal - Smoothstep interpolation, Median filter and Dilatation filter combined .......................................................... 42
3.22 Contours and Suggestive Contours ................................................. 42
3.23 Suggestive Contours - anticipation and extension ....................... 43
3.24 Armadillo - Object Space Suggestive Contours ............................ 44
3.25 Armadillo - Image Space Suggestive Contours with Smoothstep interpolation ................................................................. 44
3.26 Armadillo - Image Space Suggestive Contours without Smoothstep interpolation .............................................................. 45
3.27 Armadillo - Image Space Suggestive Contours with Median filter 45
3.28 Armadillo - Contours only ............................................................. 46
3.29 Lucy - Object Space Suggestive Contours ...................................... 47
3.30 Lucy - Image Space Suggestive Contours with Smoothstep interpolation ................................................................. 47
3.31 Lucy - Image Space Suggestive Contours without Smoothstep interpolation ................................................................. 48
3.32 Lucy - Image Space Suggestive Contours with Median filter ......... 48
3.33 Lucy - Contours only ................................................................. 49
3.34 Lion - Object Space Suggestive Contours ...................................... 50
3.35 Lion - Image space algorithm with Smoothstep interpolation ....... 50
3.36 Lion - Image Space Suggestive Contours without smoothstep interpolation ................................................................. 51
3.37 Lion - Image Space Suggestive Contours with Median filter ......... 51
3.38 Lion - Contours only ................................................................. 52
3.39 Brain - Object Space Suggestive Contours .................................... 53
3.40 Brain - Image Space Suggestive Contours with Smoothstep interpolation ................................................................. 53
3.41 Brain - Image Space Suggestive Contours without Smoothstep interpolation ................................................................. 54
3.42 Brain - Image Space Suggestive Contours with Median filter ......... 54
3.43 Brain - Contours only ................................................................. 55
3.44 Elephant - Object Space Suggestive Contours .............................. 56
3.45 Elephant - Image Space Suggestive Contours with Smoothstep interpolation ................................................................. 56
3.46 Elephant - Image Space Suggestive Contours without Smoothstep interpolation ................................................................. 57
3.47 Elephant - Image Space Suggestive Contours with Median filter .... 57
3.48 Elephant - Contours only ............................................................. 58
3.49 Hippo - Object Space Suggestive Contours .................................. 59
3.50 Hippo - Image Space Suggestive Contours with Smoothstep interpolation ................................................................. 59
3.51 Hippo - Image Space Suggestive Contours without Smoothstep interpolation ................................................................. 60
3.52 Hippo - Image Space Suggestive Contours with median filter ....... 60
3.53 Hippo - Contours only ................................................................. 61
3.54 Lucy - Image Space Suggestive Contours with radius 1 ............... 62
List of Tables

1.1 Artifacts classified according to dimensionality of data source (adapted from [22]) .......................................................... 10
List of Acronyms

3D ............ three-dimensional (space)
API ............ Application Programming Interface
DBN ............ Design by Numbers
FBO ............ Frame Buffer Object
GLSL ........... GL Shading Language
GNU GPL ....... GNU General Public License
GPU ............ Graphics Processing Unit
GUI ............ Graphical User Interface
IDE ............ Integrated Development Environment
IEEE ........... Institute of Electrical and Electronics Engineers
ITP ............ Interactive Telecommunications Program  graduate program
                at Tisch School of the Arts, NYU (New York University)
JOGL .......... Java bindings for OpenGL
GNU LGPL ....... GNU Lesser General Public License
MIT ............ Massachusetts Institute of Technology
NPAR ........... International Symposium on Non-Photorealistic Animation and
                Rendering
NPR ........... Non-Photorealistic Rendering
NYU ........... New York University
OpenGL ......... Open Graphics Library
P2D ............ custom Processing 2D software renderer
P3D ............ custom Processing 3D software renderer
PDE ............ Processing Development Environment
UCLA ........... University of California, Los Angeles
UDK ........... Universitaet der Kuenste  University of Arts Berlin
VBO ........... Vertex Buffer Object
Chapter 1

Introduction

The Processing authors Casey Reas and Ben Fry write in [18]:

The computer originated as a tool for fast calculations and has evolved into a medium for expression.

The same quotation might appear in the introduction to a book on NPR. Both Processing and NPR are dedicated to conveying concepts and emotions with the help of software, rather than aiming for lifelike depiction. Processing is already used in particular for information graphics, an area which NPR is also strong in. The NPR techniques of the Processing NPR renderer presented in this thesis can significantly enhance the legibility of such graphics in 3D. Like Processing, NPR brings together art and science.

This chapter will give an introduction both to Non-Photorealistic Rendering (Section 1.1) and the Processing language and environment (Section 1.2). Section 1.3 will derive the demands for a Processing NPR renderer based on the Processing philosophy introduced in section 1.2 and will show how the renderer fulfills these requirements.

1.1 Non-Photorealistic Rendering

For decades, there has been a quest in computer graphics to achieve ever more lifelike images, images that seek to resemble a high-quality photo of the real world. Although it is hard to pin down who coined the term photorealism, the most prominent pioneer presumably was Ivan Sutherland working in the early 1960s. His goal still has its fascination today for many people, especially as the results can easily be judged by laypersons. Yet, given the enormous progress in photorealistic computer graphics since the 1960s, the question is in the air where new frontiers may lay.

Conversely, NPR is any technique that produces images in a style other than realism. This definition, implied by the rather unfortunate choice of defining a genre by what it is not, of course is rather vague. One could certainly imagine a “Turing test” for NPR, by confronting users both with computer generated and hand-made images, with the test being passed when both are absolutely
indistinguishable. Yet, this would be rather void of meaning. Instead, images generated with NPR techniques have to be judged by how effectively they communicate. This requires looking into adjacent sciences like cognitive sciences, as well as the fields of art, graphics design and traditional illustration, that deal with structuring and abstracting information.

As Bruce and Amy Gooch [6] stress, in NPR

simulating reality is not as important as creating the illusion of reality.

The assumption is that the viewer will be able to build up a mental model of the object being portrayed. Viewers will not suspect, for example, that a surface has disappeared just because it has not been drawn.

Albeit being a very young branch of computer graphics (the first papers appeared in 1986/87), Non-Photorealistic rendering has quickly attracted increasing interest. While the field began to emerge as a number of unrelated publications, soon the generality of the underlying principles became obvious. The NPAR (International Symposium on Non-Photorealistic Animation and Rendering), sponsored by ACM SIGGRAPH in cooperation with Eurographics and co-located with SIGGRAPH, was established in the year 2000. Also, the theme for the July/August 2003 issue of IEEE (Institute of Electrical and Electronics Engineers) Computer Graphics and Applications was NPR.

The present chapter will give a taxonomy and an overview of the various branches of Non-Photorealistic Rendering, positioning the renderer presented in this thesis within this spectrum.

The chapter begins with a discussion of the traditional photorealistic approach and its drawbacks in section 1.1.1.

NPR generally refers to techniques for visually communicating ideas and information (Section 1.1.2). Major applications of NPR therefore are technical, scientific and medical illustration, product design and architecture, where comprehensibility of shapes and proportions is a goal superior to lifelike depiction. For example, the Piranesi system has proved more useful than photorealistic rendering in an application that pre-visualizes kitchen remodeling.

Other applications focus on expressiveness, drawing from artistic techniques like animated cartoons, drawings and paintings, which have a distinct expressive quality that allows an artist to convey mood and emotion. These NPR techniques combine the expressiveness of traditional media with the flexibility of computer graphics. Furthermore, digital tools can make possible animated art forms that would be impossible with traditional media (Section 1.1.3).

In addition to this, NPR techniques can be classified by the dimensionality of input they take: simulated 3D worlds, 2D pictures or formal specification (Section 1.1.4). This thesis is concerned with the branch of NPR dealing with 3D models.
Figure 1.1: Medical illustration - generated with Processing NPR

Figure 1.2: Technical illustration - generated with Processing NPR
Figure 1.3: Comic style rendering - generated with Processing NPR

Figure 1.4: “Charcoal” Drawing - generated with Processing NPR
Furthermore, NPR systems range from fully automatic to interactive (Section 1.1.5).

A survey of literature dealing with NPR concludes this chapter (Section 1.1.6).

1.1.1 Photorealism

Photorealistic rendering generates images by meticulously simulating the physical process of lighting. The resulting images should portray all of the object’s details, and only those. In other words, it should only contain artifacts that adhere to the object’s properties, or object artifacts, for short. Like a high-quality photograph should not contain blurry regions stemming from the capturing process, a photorealistic rendering should not contain artifacts stemming solely from the rendering process, like aliasing artifacts.

Although one might think that such high-quality photorealistic images are particularly comprehensible due to their resemblance to the real world, their high level of detail on contrary often renders them confusing. Psychologically, photorealistic rendering seems to imply an exactness and perfection which may overstate the fidelity of the simulated scene to a real object.

Progress in computer power these days allows for rendering ever more detailed images at higher speed, and even simulating complex natural phenomena. Computer-generated imagery and films have become ubiquitous. However, there is evidence that individuals do not always respond positively to this. This has become known by the term “uncanny valley”, introduced by Japanese roboticist Masahiro Mori in 1970 [16]. His hypothesis states that if a creature looks and acts almost like a real human, it causes a reaction of strong revulsion.

A stuffed animal or cartoon character bears enough resemblance with a human to create a sense of familiarity, and thus appreciation. When a figure becomes more human-like, however, the many subtle ways in which this figure is not human-like begin to stand out and the viewer becomes disturbed.

Another way of explaining the uncanny valley is that characters trapped in the valley resemble corpses, seriously ill individuals, or sociopaths, which makes them extremely repellent. The effect becomes even stronger with adding animation.

According to Mori, computers still have to make an incredible progress to make 3D characters really resemble to humans and create a sense of familiarity that is not repulsive.

There has been criticism on Mori’s theory stating he had no sound scientific evidence, since humanoid robots and realistic 3D characters have only recently been possible. Yet, the film industry takes the uncanny valley seriously due to negative audience reactions to the animated baby in Pixar’s 1988 short film “Tin Toy”. Another negative example is “Final Fantasy - The Spirits within”
Figure 1.5: The Uncanny Valley (figure taken from Wikipedia)

(2001), the first major wide-released CGI film to feature photorealistic characters, where a huge amount of money was spent on the animation, but the hair still doesn’t move quite right, and the eyes sometimes look even scary. The characters also lack any visible perspiration, and the lip movements, which are crucially important in perceiving human speech, seem incorrect. Likewise, the human-like characters in “The Polar Express” (2004) were described by many critics as being “disturbing”.

Also if one assumes that a progress overcoming the uncanny valley is possible, many applications remain where photorealism is simply not the means of choice. A survey among architects has shown[22] that architects preferred to show their clients sketches, rather than photorealistic renderings of planned buildings, as clients seemed to more inclined to discuss possible changes on such a basis, whereas photorealistic renderings seemed to imply that the design was already finished.

1.1.2 Comprehensibility

Many photorealistic computer-generated images fail to portray a scene in a comprehensible way. As a simple example, consider two adjacent objects of the same color, where it is not clear where one object begins and the other ends. Furthermore, in shadowed regions, information about surface properties is lost. In such cases, hatching provides stronger clues about shading and curvature than simulating the physical process of lighting. Sometimes, it is also desired to add information about the internal structure of a complex object.

This is where non-photorealistic rendering techniques come into play. These efforts benefit every field in which traditional illustrations thanks to their ability
to clarify, emphasize, and convey very precise meanings offer advantages over photographs or video. Saito and Takahshi explain that [19]

Comprehensibility is mainly created through suitable enhancement rather than by accurately simulating optical phenomena.

In NPR, certain features of an object, like contours, are emphasized, while others are neglected. Feature lines may be drawn in addition to shading, or may completely replace it. This is very similar to the way artists create drawings. Often, such drawings can not be explained by physical laws, yet they make it substantially easier for the viewer to extract the information the author wanted to communicate. An extensive discussion of line drawings can be found in chapter 3.3.

The area of illustration that has perhaps reached the highest level of sophistication is medical illustration, which has emerged from a field of art into an area of scientific research. Today, many universities offer degree programs in medical or scientific illustration. The vision of NPR is to ultimately be able to produce images of similar quality.

Technical illustrations tend to be simpler, given that the industrial objects portrayed have a more uniform structure. They tend to contain more white spaces, omitting detail not yet known, unavailable, or not necessary to the goal of illustrating a technical process, and show only these parts that are of interest in the context at hand. Such technical drawings are nowadays often created with the help of CAD! or CAAD! software, yet support for stylization, often desired in the context of architectural visualization, is limited.

With the ubiquity of online instruction manuals and e-book!s, new demands have been posed on technical and scientific illustrations. In order to become a serious alternative to printed books, the inherent properties of computers, namely interactivity, have to be exploited. Rather than simply displaying scanned images, a human-computer dialog of information exchange is desired, possibly by maintaining image-text coherence algorithmically.

1.1.3 Computer Art

There are various techniques in Non-Photorealistic Rendering that simulate traditional artistic media such as watercolor, pen and ink, or pencil, in conjunction with the type of applicator(brush, pen, or pencil) and substrate(canvas, different paper qualities) used. This can be achieved by using cellular automata, or physical simulation of fluid flow.

Research at the Media Research Lab at NYU, for example, focuses on these techniques. Their work includes research in the following areas:

- Painterly rendering methods can process images and video to make them look hand-painted.

- Multiperspective Rendering images to be rendered with non-linear perspective, freeing artists to combine images of objects with different viewpoints.
Smooth surface illustration creates pen-and-ink illustrations from 3D models.

Image analogies can automatically “learn” non-photorealistic rendering styles from examples.

For results, see http://mrl.nyu.edu/projects/npr/

Yet, any claim that an automatic process can produce drawings or paintings equal or even superior to human ones should be regarded as suspect. Humans always have the artistic freedom to omit certain lines, or place lines on locations current computer algorithms do not (or not yet) have an explanation for, without harming legibility by other humans, and often enhancing the aesthetic quality and meaning of the image. The reason is that comprehensibility depends on the object, purpose, and sometimes the viewers’ preferences, and cannot be expressed with theoretical definitions.

Instead, the computer has to be considered an artistic medium in its own right, with unique properties and new means of expression, like interactivity and genericity, which are different from, but neither inferior nor superior to traditional art. The history of art has shown that new technologies do not improve art, but they enable different forms of communication and expression.

While it is possible to create “charcoal” or “watercolor” images with this renderer, the images produced by the renderer presented in this paper do not conceal to be computer generated.

1.1.4 Input for NPR

In contrast to photorealistic rendering, where only properties of the objects, and nothing else, should be displayed (1.1.1), artifacts in NPR can also stem from a model, or an image to be processed. For example, in medical illustration, it is common to enlarge or rotate certain parts of an organ to allow for better comprehensibility. These artifacts may be classified according to their dimensionality.

Artifacts may stem from linear representation based on symbols, like language or knowledge representation, turning inherently non-geometric data into geometric data. For example, in an illustration, the subject of a sentence accompanying the illustration may be analyzed, and the respective part of a model may be highlighted. The area of information visualization deals with some aspects of this topic. A further discussion is beyond the scope of this thesis, however.

Artifacts may also be computed from 2D images. This includes, for example, pixel manipulation or dithering algorithms. There are also algorithms for computing a water-color or oil-painting look from 2D images. When running a line detection algorithm over an image, the result is an NPR. This approach has its weaknesses, however. For example, not only boundary lines due to geometry may be detected, but also discontinuities due to texture or lighting. In
Table 1.1: Artifacts classified according to dimensionality of data source (adapted from [22])

In general, the results tend to be noisier and more cluttered than using 3D models directly. Furthermore, in shadowed regions, no information can be extracted.

In these cases, it is extremely helpful to exploit properties of 3D models. In most cases, the geometry of the model is identical to the original geometry, and only the surface is rendered differently. Often, certain features of the object being portrayed are neglected, while stressing others, like boundary lines or sharp edges in the model. To extract this information, NPR needs additional information on geometry, as compared to traditional photorealistic rendering, like adjacency information and curvature of a model at a certain vertex.

Computing this information may be costly, so it is often feasible to combine a 2D approach with 3-dimensional models for interactive frame rates. This is done by rendering geometric information into two-dimensional data structures, so-called **G-buffer!s** and then using standard image-processing methods to compute the actual image. In [22], this approach is termed as $2\frac{1}{2}$D.

These **G-Buffer!s** incude:

- the id of the scene object to which the pixels belongs. This is accomplished by rendering each object with a unique color and is useful for masking, or for applying textures in a post-processing step.

- The distance of a given point on an object to the view plane (z-depth). This is also known, albeit in a different context, as z-buffer, and standard graphics processing packages offer capabilities to save the z-buffer to a texture. The z-buffer is useful for drawing lines with thickness according to perspective foreshortening, or for detecting lines when two objects of the same color overlap.

- the surface normal coordinates at the respective points, useful for detecting lines based on normal variation.

- the patch coordinates (u, v) for models consisting of spline surfaces, useful for creating hatching line by connecting pixels of equal parametrization.
- lighting or shadow buffer to determine where to actually draw hatching lines

Many other G-buffer!s are possible. In the image space algorithm for suggestive contours (3.3.1.2), for example, the dot product of the view vector and the normals are rendered to a G-buffer.

Yet, there are also situations where the actual model may be altered. Consider for example, Mickey Mouse comics. You may have noticed that his ears are always drawn in a front view, regardless of whether Mickey itself is drawn from front, back, or side. Providing users with automated systems for renderings like this is still an open area of research, which may be termed stylized modeling.

This thesis deals with NPR for 3D models. Also if in practice sometimes image space algorithms are utilized to produce the actual image, the input is always a 3D model previously rendered to a texture with conventional 3D rendering techniques. NPR for 3D models involves all phases of processing that traditional 3D computer graphics uses.

While the majority of NPR techniques applied to 3D geometry are intended to make scenes appear two-dimensional, the obvious advantage is that they allow for (possibly interactive) exploration of a model from different viewpoints.

1.1.5 Interactive vs. Deterministic Techniques

Once the sources of artifacts are identified, there must be some way of choosing among them.

One way is using random statistical values for image artifacts. Indeed, this may be necessary to make the viewer believe artifacts are not model or object artifacts.

Non-photorealistic rendering techniques for 3D models range from interactive to fully automatic. At the one end of the spectrum is arbitrariness, letting a user choose which artifacts to include. In [9], a technique for interactively drawing strokes on 3D models has been presented.

Non-Photorealism is not only about rendering, however. It also has had an impact on geometric modelling. SKETCH [25] combines gesture recognition and non-photorealistic rendering to rapidly create sketches of 3D scenes. The Teddy modeler [8] uses a sketchy rendering style to suggest the approximate 3D shapes it has inferred from the user’s 2d drawing. Teddy also allows users to paint strokes on object surfaces by projecting line segments on the object’s surface. Inspired by the children’s book “Harold and the purple crayon”, Cohen et al. [3] developed a system the lets the user create and move around in a world of drawings.

At the other end of the spectrum are deterministic algorithms.
While developing the renderer presented in this thesis, special attention has been paid to algorithms permitting rendering at interactive rates. While there are facilities for interactively adjusting parameters, featuring for example an interactive shader (see section 3.1.2), most algorithms are fully automatic. Future integration of interactive techniques like the ones mentioned above is desired.

1.1.6 Further Reading

A good starting point is http://red3d.com/cwr/npr/. Although this page was last updated September 22, 2003, and although there have been many new developments in NPR since then, it features a short and crisp overview and classification of NPR techniques and an ample collection of older books and papers.

Two standard works are “Non-Photorealistic Rendering” (2001) by Bruce and Amy Gooch [6], and “Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation” (2002) by Thomas Strothotte and Stefan Schlechtweg [22]. Again, they do not reflect recent developments, but give a sound introduction to the topic as a whole and systematic assessment of the literature up to their publication.

Gooch and Gooch [6] divide the area of NPR into three sections: Simulating Artistic Media, assisting a User in the Artistic Process, and Automatic Systems. They give in-depth descriptions of the respective algorithms and references to traditional art. Gooch and Gooch also maintain a website on NPR at http://www.cs.utah.edu/npr/

Strothotte and Stefan Schlechtweg give a thorough introduction on the theory and history of NPR. Accompanying a graduate level course in NPR at the University of Magdeburg, Germany, they give exercises at the end of each chapter.


1.2 Processing

Processing is an open project initiated by Ben Fry and Casey Reas in 2001. It evolved from ideas explored in the Aesthetics and Computation Group at the MIT (Massachusetts Institute of Technology) Media Lab, specifically the DBN (Design by Numbers) Project both Fry and Reas worked on under Professor John Maeda. Quoting from the DBN webpage [11],

Design By Numbers was created for visual designers and artists as an introduction to computational design. It is the result of a continuing en-
deavor by Professor John Maeda to teach the “idea” of computation to designers and artists. It is his belief that the quality of media art and design can only improve through establishing educational infrastructure in arts and technology schools that create strong, cross-disciplinary individuals.

Evolving from these ideas, Processing is an open, Java-based programming language and environment intended specifically to program images, sound, animation, and interactions. Initially created to serve as a software sketchbook to simplify the task of prototyping and to teach fundamentals of computer programming within a visual context, Processing quickly developed into a tool for creating finished professional work.

For their work on Processing, Fry and Reas received the 2008 Muriel Cooper Prize from the Design Management Institute. The Processing community was awarded the 2005 Prix Ars Electronica Golden Nica award and the 2005 Interactive Design Prize from the Tokyo Type Director’s Club.

Processing is used by students, artists, designers, researchers, and hobbyists for learning, prototyping, and production. It is developed by artists and designers as an alternative to proprietary software tools so far predominant in the domain, making it accessible to schools and individual students.

Casey Reas and Ben Fry write in [18]:

As a result of being easy to use, these [proprietary graphics and rendering] tools obscure some of the computers potential. To fully explore the computer as an artistic material, it’s important to understand this arcane art of computer programming. Processing strives to make it possible and advantageous for people within the visual arts to learn how to build their own tools - to become software literate.

Alan Kay, explains in [10] what literacy means in relation to software:

The ability to “read” a medium means you can access materials and tools created by others. The ability to “write” in a medium means you can generate materials and tools for others. You must have both to be literate. In print writing, the tools you generate are rhetorical; they demonstrate and convince. In computer writing, the tools you generate are processes; they simulate and decide.

Alan Kay is the creator of the Dynabook vision dating back as early as 1968. His ideas led to a prototype developed at Xerox Parc, which was an early laptop computer designed for giving “children of all ages” access to digital media. In this early vision of the mid-1970s, there was no distinction between a computer user and programmer.

Since these days, we come a long way, and the Dynabook vision has still not been fully realized. Computers are now ubiquitous, but computer software has become increasingly complex, inhibiting users from building their own tools. Yet, Reas and Fry stress [17] that

Understanding software and its impact on culture is a basis for understanding and contributing to contemporary society.
Processing is filling a gap in the graphics domain, allowing non-experts in computer science, or autodidacts, to create their own content, without the restrictions imposed by given graphics tools. The hope is that Processing’s success will encourage similar initiatives in the domain.

The remainder of this chapter is structured as follows: Section 1.2.1 takes a closer look at the various fields of application Processing is used in by now and analyzes how this success could be achieved. The Processing API and Development Environment themselves are explained in Section 1.2.2. Section 1.2.3 addresses the issue of licensing. A comparison between Processing and pure Java is given in section 1.2.4. Besides, the reasons for choosing the Java language as a basis for Processing as opposed to other alternatives are discussed. Finally, further reading on Processing and Aesthetic Computing in general is recommended in section 1.2.5.

1.2.1 Fields of Application

In an endeavor to spread software literacy (for a definition see the introduction to this section in 1.2), students at hundreds of schools around the world use Processing for classes ranging from middle school math education to undergraduate programming courses to graduate fine arts studios.

Only a few of them are

- New York University's Tisch School of the Arts, where Daniel Shiffman is teaching Processing in the graduate ITP (Interactive Telecommunications Program)
- UCLA (University of California, Los Angeles), where undergraduates in the Design | Media Arts program use Processing to learn the concepts and skills needed to imagine the next generation of web sites and video games
- Lincoln Public Schools in Nebraska and the Phoenix Country Day School in Arizona, where middle school teachers are experimenting with Processing to supplement traditional algebra and geometry classes
- UDK (Universitaet der Kuenste) Berlin, where Processing is taught to undergraduates and is used for open work in interactive software applications and installations interfacing with microcontrollers.

Processing makes it easy to learn the basics of programming even for complete beginners. Because the environment is so minimal, students are able to begin programming after only a few minutes of instruction.

The Processing website (http://www.processing.org/) includes tutorials, exhibitions, interviews, a complete reference, and hundreds of software examples. The Discourse forum hosts continuous community discussions and dialog with the developers.

And, because the Processing syntax is derived from widely used programming languages, it is a good base for moving on to more general programming
languages like full Java and C++. Processing may be described as a gateway
drug to higher-level languages for people who, as Reas and Fry put it [18],
are interested in programming but who may be intimidated by or uninter-
tested in the type taught in computer science departments.

At the same time, this very syntax makes Processing understandable to many
people who are already familiar with writing software and opens up media art
concepts to a more technical audience.

Processing has the distinct potential to do two things: give designers and
artists new ways of communicating, and give computer scientists new ways
of attracting people to their field through atadesign as a catalyst. The
code becomes the artists material.

Processing takes advantage of the strengths of web-based communities, and
the authors themselves seems to be surprised that [18]

this has allowed the project to grow in unexpected ways. Thousands of
students, educators, and practitioners across five continents are involved
in using the software.

Users are encouraged to contribute and share code and develop libraries.
Many classes taught using Processing publish the complete curriculum on the
web, and students publish their software assignments and source code from
which others can learn. The Processing authors stress that[18],

Sharing and improving code allows people to learn from one another and
to build projects that would be too complex to accomplish without assis-
tance.

In fact, the NPR renderer presented in this thesis itself builds on two libraries
written by the community.
One of them is Tatsuya Saito’s obj model loader, which was modified to render
geometry to display lists, and enhanced with smooth normal calculation and
the data structures necessary for NPR object space algorithms, which strongly
rely on connectivity information in the model.
The other is JohnG’s GLSL library, which was extended with methods to set
uniform variables in shaders.

This way, despite it’s sketching (making Java feel more like a scripting lan-
guage) approach, Processing has evolved as a tool used for serious work and is
used for even large-scale installations. Yet, it’s complexity is scalable.

- Tens of thousands of companies, artists, designers, architects, and re-
searchers use Processing to create an amazingly diverse range of projects.

- Design firms such as Motion Theory provide motion graphics created with
Processing for the TV commercials of companies like Nike, Budweiser, and
Hewlett-Packard.

- Bands such as R.E.M., Radiohead, and Modest Mouse have featured ani-
mation created with Processing in their music videos.
Publications such as the journal Nature, the New York Times, Seed, and Communications of the ACM have commissioned information graphics created with Processing.

The artist group HeHe used Processing to produce their award-winning Nuage Vert installation, a large-scale public visualization of pollution levels in Helsinki.

The University of Washington’s Applied Physics Lab used Processing to create a visualization of a coastal marine ecosystem as a part of the NSF RISE project.

The Armstrong Institute for Interactive Media Studies at Miami University uses Processing to build visualization tools and analyze text for digital humanities research.

### 1.2.2 The Processing API and Development Environment

Often, confusion arises what is meant by the term Processing. (Is it the language? Is it the graphics library? Is it the development environment?)

In fact, what is termed Processing is composed of two distinct components: the Processing IDE (Integrated Development Environment), called PDE (Processing Development Environment) for short, and the Processing API.

The IDE is just an ordinary development environment (albeit a very user-friendly one) that allows for writing, running, and viewing Processing programs. It also exports applets for the web or standalone applications for Mac, Windows, and GNU/Linux with the click of a button.

The API (the actual commands making up the language) is an independent library of code accessible from within the IDE and structured as a Java package, called processing.core. This package, being written in Java, can also be used outside the PDE in any Java project.

Figure 1.6 shows a screenshot of the PDE and a simple example of how to use the API. A more thorough documentation of the API is beyond the scope of this thesis, but a complete reference can be found at http://www.processing.org/reference/index.html

### 1.2.3 Licensing

Both the IDE and the graphics library are Open Source Software.

The PDE is released under the GNU GPL (GNU General Public License). Using the GNU GPL will require that all the released improved versions be free software in order to avoid the risk of having to compete with a proprietary modified version of your own work.

The export libraries (also known as ‘core’) are released under the GNU LGPL (GNU Lesser General Public License), which permits use of the library
Figure 1.6: The Processing Development Environment

```java
void setup() {
    size(100, 100);
    noLoop();
}

void draw() {
    diagonals(40, 90);
    diagonals(60, 60);
    diagonals(20, 40);
}

void diagonals(int x, int y) {
    line(x, y, x+20, y-40);
    line(x+10, y, x+30, y-40);
    line(x+20, y, x+40, y-40);
}
```
in proprietary programs.

Reas and Fry state [18] that

Processing probably would not exist without its ties to open source software. Using existing open source projects as guidance, and for important software components, has allowed the project to develop in a smaller amount of time and without a large team of programmers.

Furthermore,

individuals are more likely to donate their time to an open source project, and therefore the software evolves without a budget.

1.2.4 Processing and Java

Is Processing Java? While the Processing environment and library is written in Java, and programs written in Processing are translated to Java and then run as Java programs, and while it is possible to use full Java Syntax in the PDE, there are a few differences.

Specifically, the PDE comes with three programming modes: Basic Mode, Continuous Mode, and Java Mode.

People new to programming should begin with the Basic Mode, which is intended for drawing static images and learning fundamentals of programming.

Continuous Mode provides a setup() structure that is run once when the program begins and a draw() structure which by default continually loops through the code inside (for an example, see Image 1.6). This additional structure allows writing custom functions and classes and using keyboard and mouse events.

Java Mode is the most flexible, allowing complete Java programs to be written from inside the Processing Environment. Note that it is not necessary to use this mode just to get features of the Java language.

This explanation adapted from http://processing.org/reference/environment/, where code examples highlighting the above can be found)

The Processing language fosters a simplified programming style that doesn’t require users to understand more advanced concepts like classes and objects (while still making them accessible for advanced users). When hitting the run button, an enclosing class derived from PApplet is automatically added around the sketch code.

With programs becoming more complex, code can be split up into tabs. When a program with multiple tabs is run, the code is grouped together and the classes in other tabs become inner classes.

These mechanisms are completely transparent to the user. That way, new users don’t have to worry about issues normally associated with setting up Java
like packages and classpaths.

Also, modifiers like public can be omitted. Instead, when Processing code is converted to standard Java, these modifiers are automatically put in. Confusing details, often annoying both new and experienced users, like having to put an “f” after each floating point constant are alleviated by the Processing parser.

The reason for choosing Java over other alternatives like JavaScript, Python, or Ruby was its relative speed advantages through bytecode! compilation (important for many graphics applications), making it faster than ActionScript (Adobe Flash) or Lingo (Macromedia Director). On the one hand, Java has very moderate complexity compared to languages like C++, seeing to the fact that many users are absolute beginners in programming. Another important fact was that Java also allows users to export sketches for distribution via the web.

Writing graphics programs in pure Java is a comparatively tedious and cumbersome task. Processing contrasts with Java by the Processing graphics library, insulating users from details like animation and double-buffering, which must be specifically programmed in Java, but are integrated into Processing, making programs shorter and easier to read.

Fry puts it this way[5]:

Why force students or casual programmers to learn about graphics contexts, threading, and event handling functions before they can show something on the screen that interacts with the mouse? The same goes for advanced developers: why should they always need to start with the same two pages of code whenever they begin a project?

1.2.5 Further Reading

An excellent introduction by the Processing developers Ben Fry and Casey Reas themselves, especially, but not only for beginners, is the book “Processing: A Programming Handbook for Visual Designers and Artists”[18]. Feeding upon their 6 years’ experience in software development and teaching, they give step-by-step, illustrated examples on software structures, and how they relate to the visual arts. For more advanced users, topics like computer vision, sound, and electronics are explored. Additionally, the book contains numerous essays and interviews. This book is a good starting point to deepen the topics addressed in sections 1.2.1 to 1.2.4, giving first-hand experience from Processing’s usage in class. Sample chapters are available from:
http://www.processing.org/img/learning/Processing-Sample-070607.pdf

Another good source for learning Processing language is Ira Greenberg’s “Processing: Creative Coding and Computational Art (Foundation)”[7]. Starting out as an introduction to computer programming, it also tackles more advanced topics like object orientation, 3D programming and bare Java programming in the higher chapters. It specifically targets artists, designers, and
other creative professionals and students, giving them the foundation to create code art, visualizations, and interactive applications, and teaching them how to program 2D and 3D animation, pixel-level imaging, motion effects, and physics simulation.

Sample chapters are available from:


This book tells a story. It’s a story of liberation, of taking the first steps towards understanding the foundations of computing, writing your own code, and creating your own media without the bonds of existing software tools. This story is not reserved for computer scientists and engineers. This story is for you.

Drawing from his teaching experience at Tisch School of the Arts at NYU, Daniel Shiffman’s website contains numerous tutorials, examples and exercises.

As a complement to pure Processing programming guides, further reading on the theory of Aesthetic Computing (a full discussion of which is beyond the scope of this thesis), can be found in the following publications:

Somewhat in between is Ben Fry’s “Visualizing Data”[5], detailing specifically on how to use Processing to build comprehensible and aesthetic interactive depictions of complex, large scale data sets, in order, as the O’Reilly website says, to “take advantage of data that you might otherwise never use”. It takes the user from the beginnings of a raw sketch, to beautiful, finished pieces, teaching the basic visualization principles.

To engross the thoughts on information architecture given in “Visualizing Data”[5], reading Fry’s doctoral adviser’s (John Maeda, world-renowned graphic designer, visual artist, MIT trained computer scientist and professor at the MIT Media Lab), publications is recommended.

In his most recent book “The Laws of Simplicity”[15], Maeda goes into simplifying complex systems based on the principles of reduction, organization and efficiency, opting for less - fewer features, fewer buttons and fewer distractions, in order to reach a balance between simplicity and complexity. In short (quoting from http://lawsofsimplicity.com/), the question is: “How simple can you make it?” versus “How complex does it have to be?”. The same questions apply for the branch of Non-Photorealistic rendering which opts for a reduced visualization of 3D models.

“Creative Code”[14] by John Maeda presents a walk through 7 years of his teaching experiments at MIT.

Maeda’s thoughts on the book:
My evolved focus is on the global economy, and how creativity might reshape our society. I hope that a world that endears the arts just as much as it endears its weapons of destruction might be able to enjoy this century to the fullest. We can either just watch what happens, or commit to shaping the events that will come. I choose the latter along with many colleagues all around the world.

The book includes guest essays by Casey Reas and Ben Fry, among others.

“Design by Numbers”[12] describes the DBN language Processing emerged from, establishing the computer as an artistic medium in its own right. DBN features a simplified and reduced syntax allowing designers to create their own software tools with little or no mathematical and programming background.

Finally, his 480-page retrospective “MAEDA@MEDIA”[13] is a complete overview of Maeda’s work and philosophy, his pedagogical approach to digital visual art, and his quest to understand the very nature of the relationship between technology and creativity.

Aesthetic computing is concerned with the impact and effects of aesthetics on the field of computing. “An Introduction to Aesthetic Computing”[4] gives an overview of fields that combine art, mathematics, and computing and discusses the application of the theory and practice of art to the field of computing. It covers a wide range of subjects from different perspectives, with themes including art, emotion, metaphor, mathematics, transdisciplinarity, visualization, auralization, programming, and interface design.

Key scholars and practitioners from art, design, computer science, and mathematics contribute essays ranging from the art of visualization and “the poesy of programming” to discussions of the aesthetics of mathematics throughout history and transparency and reflectivity in interface design.

Also, the “Leonardo Book Series”, published by MIT Press, publishes texts by artists, scientists, researchers and scholars that present innovative discourse on the convergence of art, science and technology. Envisioned as a catalyst for enterprise, research and creative and scholarly experimentation, the series enables diverse intellectual communities to explore common grounds of expertise. Leonardo Books provide for the contextualization of contemporary practice, ideas and frameworks represented by those working at the intersection of art and science.

1.3 Requirements for a Processing NPR renderer

This chapter sums up the requirements for library development in Processing, and, more specifically, for developing an alternative Processing renderer. It also explains step by step how these criteria are met.

First of all, a new renderer should not require the user to learn a new language. The fact that there is a whole different renderer underneath has to be
completely transparent. Everything should have the “look and feel” of the Processing the user is used to (Section 1.3.1).
Second, the additional Non-Photorealistic Rendering API should be as easy-to-use and intuitive as possible, with beginners in programming in mind (Section 1.3.2).
For the same reason, special attention has to be paid to documentation (Section 1.3.3).
The source code is made public domain to foster further development of the renderer and of new algorithms plugged into the renderer (Section 1.3.4).
Supporting this thesis, a web page is provided enabling users to browse source code and documentation online and download the library (Section 1.3.5).
This web page offers detailed explanation and example code for each of the library functions along with images and applets (Section 1.3.6).
Finally, performance issues are discussed in section 1.3.7, and the related issue of compatibility to lower-end or older graphics hardware is raised in section 1.3.8.

1.3.1 Integration with the Processing Language

One concern when developing the renderer at hand was seamless integration with the Processing language.
Luckily, Processing provides a framework to plug in alternative renderers. To use the renderer, all that has to be done is passing the static String variable NPR.renderer to the size() command, through which the rendering mode is specified.

```
size(1024, 1024, \ac{NPR}.renderer);
```

and the renderer library has to be imported using

```
import npr.*;
```

after placing the library in the libraries folder in the sketchbook location. Selecting “Import Library” from the Sketch menu in the PDE and choosing npr adds this line to the sketch automatically.

In fact, Processing itself currently has five rendering modes. 2D programs written with Processing can be rendered using the Java2D drawing libraries, suitable for high-quality 2D vector graphics, but at the expense of speed, or a custom 2D engine called P2D, intended for large amounts of simpler graphics, and offering fast pixel operations when directly manipulating pixels of an image or video. Finally, the PDF renderer draws all geometry to a file instead of the screen based on the Java2D renderer.

3D Graphics can be rendered through OpenGL (Open Graphics Library) using the JOGL (Java bindings for OpenGL) interface, and a custom 3D engine called P3D, an optimized software renderer similarly fast for pixel operations like P2D.

In an attempt to offer a unified mode of thinking about drawing, whether in two or three dimensions, the P2D and P3D renderer were initially one, and in
fact the only renderer. However, because of the number of tradeoffs that must be made between 2D and 3D and different expectation of quality for 2D and 3D, the renderer was split up.

A large effort has been made to make the Processing language behave similarly across the different rendering modes. This goal has for the most part been achieved (exceptions being the lack of support of the P2D renderer for stroke caps and joins on thick lines and the smooth() command being disabled in P3D). There are no additional methods or fields for the OpenGL library. It uses the Processing language, but renders geometry differently.

The renderer presented in this thesis subclasses the PGraphicsOpenGL renderer. Again, the NPR renderer uses the unaltered Processing language, but renders geometry differently. Using this renderer does not require the user to learn a new graphics API. The fact that there is a whole different renderer underneath (see chapter 2), is completely hidden from the user.

When not using NPR rendering, the renderer behaves exactly like the PGraphicsOpenGL renderer. If it does not, this should be considered a bug. Non-photorealistic rendering and normal Processing rendering can be seamlessly integrated in one sketch. All algorithms, except for object space line detection algorithms, work for ordinary Processing rendering commands as well as models loaded from .obj files, which are rendered to display lists using OpenGL commands, although the user is unaware of that fact.

When using NPR rendering, however, there are a few restrictions. Depth sorting, a feature of Processing to enable correct alpha blending, for example, is not possible, as, like mentioned above, geometry may be rendered to display lists upon initialization time. The main alterations, however, the fact that most algorithms need multiple rendering passes, for example, are completely transparent. (These multiple passes are also the reason for the renderer to subclass PGraphicsOpenGL, so the library has access to the protected, low-level rendering methods.)

### 1.3.2 The Non-Photorealistic Rendering API

The API to access NPR algorithms on top of the renderer has been carefully designed to be as intuitive as possible with beginners in programming in mind. All it requires is a basic knowledge of object orientation. For example, the basic code for using shading is:

```java
import npr.*;

DiffuseShader diffuseShader;

void setup(){
  size(1024, 1024, NPR.renderer);
}
diffuseShader = new DiffuseShader();
}

void draw(){
    diffuseShader.begin();
    //draw geometry here
    diffuseShader.end();
}

For more advanced users, a lower level API exists that allows them to develop their own (NPR or not NPR) algorithms.

An extensive documentation of the API can be found in 5

1.3.3 Documentation

While most of the Algorithms provided with the renderer have various options that can be set through a number of methods, all of the options are set to reasonable default values in order to get the user started with a working program as quickly as possible.

To this end, special attention has to be paid to good documentation, because ultimately users will want more control over the results. Processing pursues a strong educational purpose and so should Processing libraries.

Therefore, all code had to be explained thoroughly and annotated to automatically derive JavaDoc documentation. (Javadoc is a tool from Sun Microsystems for generating API documentation in HTML format from doc comments in source code.)

This is crucial for developers being able to plug in algorithms and help to develop the renderer beyond what can be done in a 6 months’ thesis.

1.3.4 Source

While it is not strictly necessary, Processing library developers are encouraged to give away their source code to foster the growth of the community. This is also a requirement for the library to be promoted at processing.org/reference/libraries.

The source of this renderer is made available under the GNU Lesser GPL (GNU LGPL). (see chapter 1.2.3 and http://www.gnu.org/licenses/lgpl.html)

1.3.5 Web Page

The guidelines for Processing library development (http://dev.processing.org/libraries/guidelines.html) also request an online summary page:

A Processing library should have its own web page, stored at a stable URL (or at least as stable as possible), and should include:
- A short abstract that describes the purpose of the library.
- The library has been successfully tested on which platforms? (OS X, Windows XP and Vista, Linux)
- The latest Processing version the library has been tested with?
- Dependencies. Does the library depend on any other library?
- A list of examples that demonstrate the use and potential of the library.
- Source code (if open source). We recommend using Google Code to host the source code of a library in a SVN repository, then it is very easy to browse the code online.
- Keywords that describe the aim and function of the library.
- Last update. When was the last update of the library?
- A link to a zip file that includes the library, documentation and examples.

The webpage where the NPR renderer is hosted is www.katrinlang.de/npr/index.html. From there, a zip file with the following contents can be downloaded:

npr/library/npr.jar
npr/reference
npr/examples
npr/shaders
npr/src

This conforms to the layout required for Processing libraries and allows the examples to be immediately accessible in the PDE. In addition, this is the structure users are familiar with from other Processing libraries.

This zip file should be placed in the libraries folder inside the sketchbook location.

The file npr.jar in the library folder is the actual library.

Reference contains the JavaDoc documentation, which also can be found online.

Examples contains two Processing sketches:
npr_small_example and npr_big_example
The example npr_small_example is a simple example of using a subset of the API programatically, while npr_big_example uses all of the API and features a GUI (Graphical User Interface).
If you followed the installation instructions, you should be able to access the examples in the PDE by selecting File - Sketchbook - libraries - npr - examples. For instructions on how to use these examples see chapter 4.

Shaders contains the necessary GLSL shaders. Whenever you start a new sketch, copy its contents to your sketch's data folder. You need at least:
main-shader.vert and main-shader.frag.

Finally, src contains the source code.

The Google Code link is ???.

1.3.6 Examples

Users tend to learn best from examples, therefore examples are important for a library release.

In addition to small code snippets on the web page explaining individual library features, one big example comes with the release that uses all of the library methods and allows users to interactively explore the features of the API without having to adjust parameters and recompile each time. (the example program is explained in chapter 4).

The GUI used in this example is available as a subpackage so users can use it in their projects independent from the NPR renderer or replace it by their favorite GUI package if desired. Not part of the GUI library, but a core feature of the renderer is an interactive lookup table for shading (for usage, instructions, see section 3.1.2). The lookup table may be used to achieve a cel shaded look, to simulate the effect of an ambient light, specular reflection, material properties or to achieve artistic effects, providing users with an ad hoc approach to shading and lighting without requiring novice users to learn the theory of lighting, and without repeated adjusting of parameters and recompiling.

The following commands create a LookupTable, attaches it to a shader and subsequently turn it into a cel shader.

Import npr.*;

DiffuseShader cel;

void setup(){
    size(1024, 1024, NPR.renderer);
    cel= new DiffuseShader();
    // parameters are x and y position of the lookup table
    lut= new LookUpTable(20, 20);
    cel.setLookUpTable(lut);
    lut.setRange(0, 100, 150);
    lut.setRange(100, 200, 200);
    lut.setRange(200, 255, 255);
}

void draw(){
    cel.begin();
    //draw geometry here
    cel.end();
    lut.draw();
1.3.7 Performance issues

Another issue is speed. Nobody will want to use the renderer if it does not run at interactive speed for models of at least moderate size, always keeping in mind that the renderer needs multiple rendering passes.

An initial implementation has shown that the traversal of the Processing data structures is very slow. Through the use of display lists generated at startup time, which are stored in GPU (Graphics Processing Unit) memory and thus are extremely fast, a considerable speedup could be achieved. In fact, despite using up to 7 rendering passes, the renderer is still an order of magnitude faster than the original PGraphicsOpenGL renderer. This is also due to the fact that virtually all work like lighting and image space silhouette detection is done in GLSL shaders, avoiding Java performance issues.

1.3.8 Compatibility

While it was a major goal for the renderer to achieve speedup through the use of shaders, this restricts the use of this library to graphics cards supporting GLSL.

Still, special attention has been paid to the issue of compatibility to lower end graphics cards wherever possible, as the average Processing user cannot be expected to name the latest gamer hardware their own. For example, the ATI Mobility FireGL T2 the renderer was partly developed with does only support 3x3 filter kernels due to limitations in the number of variables and for loops. Therefore 3x3 filters and 5x5 filters are provided as separate GLSL files so owners of low end graphics cards can at least benefit from image space algorithms in a limited way. If your graphics card is overburdened with the number of for loops in the main shader, you can easily restrict the number of lights by editing a preprocessor constant in the shader. In fact, when sharing your code, make sure to set this constant to the actual number of lights being used to ensure compatibility.

Compatibility with older hardware was also the reason to choose display lists over vertex buffer objects (VBO (Vertex Buffer Object))s also if this limits the ability to animate a model. Besides, the data in vertex arrays resides on the client side (PC main memory), it has to be transferred to the server (graphics hardware) in each frame.

While the renderer does make use of Framebuffer objects (FBO (Frame Buffer Object))s, and offers a concise API for them, an automatic fallback to copying from the framebuffer to textures is provided.
Chapter 2

Architecture

The Processing OpenGL renderer offers considerable speedup on OpenGL accelerated graphics cards, allowing to draw more geometry at bigger screen sizes. Yet, to minimize development effort, all lighting and transformations are still done in software. OpenGL lighting is disabled and pre-lit vertices in eye space are sent to OpenGL. Thus, the OpenGL renderer is not accelerated as much as it could be. Also, this means that the internal data structures holding geometry have to be re-created at each frame.

Unfortunately, this setup excludes the use of models larger than a few thousand vertices. Therefore, the renderer had to be rewritten to do all lighting and transformations in hardware.

This also was a prerequisite for the general use of glsl shaders, which are used for all image space NPR algorithms in order to avoid Java performance issues. These shaders are independent from Java and may be used freely in any other context.

Furthermore, transforming vertices to eye space does not work with shadow mapping, when the scene has to be rendered from the light’s viewpoint.

The OpenGL renderer allows for using pure OpenGL commands in Processing sketches. The beginGL() command pushes the Processing modelview matrix on the OpenGL matrix stack. An example is shown below:

```java
import javax.media.opengl.*;
import processing.opengl.*;

float a;

void setup(){
    size(800, 600, OPENGL);
}

void draw(){
    background(255);
}
```
PGraphicsOpenGL pgl = (PGraphicsOpenGL) g; // g may change
GL gl = pgl.beginGL(); // always use the GL object returned by beginGL

// Do some things with gl.xxx functions here.
gl.glColor4f(0.7, 0.7, 0.7, 0.8);
gl.glTranslatef(width/2, height/2, 0);
gl.glRotatef(a, 1, 0, 0);
gl.glRotatef(a*2, 0, 1, 0);
gl.glRectf(-200, -200, 200, 200);
gl.glRotatef(90, 1, 0, 0);
gl.glRectf(-200, -200, 200, 200);

pgl.endGL();
a += 0.5;
}

Always do the assignment/cast from 'g' to 'pgl' inside draw(), because
'g' may be subject to change. In addition, use the gl object returned by
beginGL() for the reason that the GL context itself may have changed. See
http://processing.org/reference/libraries/opengl/

While this mechanism is used internally (just replace the cast to
PGraphicsOpenGL with NPR), novice users cannot be bothered with OpenGL
commands, although developers plugging in new algorithms will most likely
want to use OpenGL.

Instead, the renderer subclasses PGraphicsOpenGL and overrides all methods
that do the software transformations and lighting, based on whether rendering
in NPR “mode”, e.g. during the execution of a NPRAlgorithmXXX.begin() and
NPRAlgorithmXXX.end() block. Sometimes, the renderer will do some addi-
tional work instead of just overriding, like normal calculation, a convenience
when the user has not explicitly specified normals.
In other cases, the renderer may need to extract information from a method call
before passing over to the superclass method, like the light position in world
space (needed for rendering shadow maps).

Instead of sending eye space coordinates to OpenGL with the OpenGL ma-
trix set to identity, the Processing Matrices have to be pushed onto the OpenGL
matrix stack when entering a begin()/end() block.

However, hybrid and image space NPR algorithms rely on multiple render-
ing passes. To accomplish this, a trick is used. Upon entering a begin()/end() block, depth sorting is enabled. This causes Processing to store geometry in its
internal data structures, instead of rendering them directly on the screen. The
triangles are never sorted, however. Instead, the internal data structures are
cleared when NPR rendering is finished.

Yet, keeping geometry in internal data structures is not sufficient. The same
has to be done with matrices. Therefore, the concept of a matrix history is
introduced. The matrix history stores each matrix along with a triangle index, or a display list, which was registered by the model loader when the model’s `draw()` method was called.

Note that not individual transformations are stored, but the modelview matrix as a whole. Therefore, the claim that all transformations are done in hardware is not exactly true. Yet, the matrix multiplication has to be done in software, otherwise the renderer would not be in a consistent state after leaving a `begin()/end()` block. The expensive part of multiplying each vertex by the modelview matrix is done in hardware, however.

Maintaining and traversing this data structure is a very small overhead, as only the last of multiple transformations executed in sequence is stored, and the total number of transformations is neglectible compared to the number of triangles rendered.

Storing modelview matrices is still not enough, as shadow mapping needs only the model part of the modelview when rendering from a light source’s view. Furthermore, it also requires the inverse camera matrix to be multiplied onto the texture matrix.

Fortunately, Processing maintains a separate camera matrix. The modelview is set to this matrix on `beginDraw` at the beginning of each frame. The renderer also maintains the respective inverse to camera and modelview matrices.

Maintaining a separate camera matrix gives the user intuitive control over camera movements. The code below results in a continuously rotating camera when used inside `draw()` (to result in a constant amount of rotation, the camera matrix has to be cleared before rotating, as the camera matrix is not reset on `beginDraw`):

```
beginCamera();
rotateY(Pi/8);
endCamera();
```

All transformations are multiplied onto a matrix called `forwardTransform`, and premultiplied onto `reverseTransform`. These two matrices initially point to the modelview and the modelviewInv matrix respectively, but they are swapped when in camera manipulation mode, meaning that the camera inverse matrix is assigned to `forwardTransform`, and the camera matrix is assigned to `reverseTransform`. Upon calling `endCamera()`, finally, the modelview is set to the camera matrix, meaning that all previous transformations are lost (the same is done for the inverse). That way camera transforms are automatically accumulated in inverse on the modelview matrix.

The existence of a separate camera/camera inverse matrix makes it possible to get the model part of the modelview back by multiplying it onto a copy of the camera inverse.

All these matrices are stored in the matrix history as float arrays, which may then be passed to a `glMultMatrixf()` call. The renderer’s `render` method,
Figure 2.1: Processing Coordinate System: The x axis is pointing to the right, the y axis is pointing downward, and the z axis is pointing towards the viewer which may be invoked by NPR algorithms takes a start and end index to the matrix history, and an array of targets each matrix should be pushed upon (modelview, texture matrices, etc.)

Before the matrices are stored, they have to be transposed, since OpenGL uses column-major layout, while Processing's layout is row-major. The renderer offers a transpose method, which should be used when developing algorithms and having to deal with Processing Matrices.

Processing differs from OpenGL in another important fact: It uses a left-handed coordinate system (see figure 2.1), when OpenGL uses a right-handed one. In Processing, the positive x axis is pointing to the right, the positive y axis is pointing downward, and the positive z axis is pointing towards the viewer.

As the y axis points downwards, to flip the image so it is displayed correctly with OpenGL, the PGraphicsOpenGL renderer premultiplies the modelview matrix with a scaling of -1 in the y direction (by pushing the appropriate scaling matrix onto the OpenGL matrix stack. In addition to this, the vertex order has to be set to clockwise, instead of OpenGL's standard counterclockwise. With the NPR renderer, the flipping still has to be done, however the vertices are already in the “correct” order, so vertex order has to be set to counterclockwise upon entering a begin()/end() block in order to make backface culling and lighting work correctly, and set back to clockwise when leaving it.

For details on how algorithms are plugged into this framework, see chapter 5.4.
Chapter 3

NPR Algorithms

3.1 Shading

Shading is implemented as a diffuse pixelshader, which can be modified using a lookup table. This lookup table may either be accessed programatically (for examples see chapter 1), or by means of a graphical frontend, allowing for interactive previewing of results.

The lookup table is divided into regions, which may either be filled with a solid color, or a gradient.

When the user is contented with the result, it can be saved to the data folder. A previously saved lookup table can be loaded from a file:

```java
lut = new LookUpTable("cel", 20, 20);
```

Lookup tables use a custom file format which allows to retrieve the boundaries of colour ranges or gradients for interactive modification. The file extension is .lut.

![Figure 3.1: There are three classes of NPR algorithms implemented: Shading, Shadows, and Line Drawings](image)
Figure 3.2: Class hierarchy of all algorithms implemented

3.1.1 Cel Shading
3.1.2 Using Look-up-Tables
3.1.3 Results

3.2 Shadows
3.2.1 Shadow Mapping
3.2.2 Stylization of Shadows
3.2.2.1 Results

3.3 Line Drawings
Line drawings For shape comprehension, line drawings are effectively used as an addition to or substitute for surface coloring and shading
    Sobel: first order differential operator

3.3.1 Image Space Algorithms
3.3.1.1 Sobel Contours
3.3.1.1.1 Results
Figure 3.3: Knot - rendered with unaltered diffuse shading

Figure 3.4: Knot - rendered with simulated ambient light

Figure 3.5: Torus - rendered with Cel Shading
Figure 3.6: Blade - rendered with sharp specular highlight

Figure 3.7: Knot - simulated metal
Figure 3.8: Cubehole - Image Space Sobel edge detection

Figure 3.9: Cubehole - Image Space Sobel edge detection and 3x3 dilatation filter with threshold 1
Figure 3.10: Hippo - Image Space Sobel edge detection

Figure 3.11: Hippo - Image Space Sobel edge detection with Smoothstep Interpolation
Figure 3.12: Hippo - Image Space Sobel edge detection with Smoothstep interpolation and 3x3 Gaussian smoothing. Source: ?

Figure 3.13: Hippo - Image Space Sobel edge detection with Smoothstep interpolation and 5x5 Gaussian smoothing. Source: ?
Figure 3.14: Elephant - Image Space Sobel edge detection

Figure 3.15: Elephant - Image Space Sobel edge detection with Smoothstep interpolation
Figure 3.16: Elephant - Image Space Sobel edge detection with Smoothstep interpolation and 3x3 Gaussian smoothing

Figure 3.17: Elephant - Image Space Sobel edge detection with smoothstep interpolation and 5x5 gaussian smoothing
Figure 3.18: Rockerarm - Image Space Sobel edge detection with Smoothstep interpolation

Figure 3.19: Rockerarm - Image Space Sobel edge detection with Smoothstep interpolation medium steep curve

Figure 3.20: Rockerarm - Image Space Sobel edge detection with smoothstep interpolation steep curve
3.3.1.2 Suggestive Contours

3.3.1.2.1 Results
Figure 3.23: Suggestive Contours - anticipation and extension
Figure 3.24: Armadillo - Object Space Suggestive Contours

Figure 3.25: Armadillo - Image Space Suggestive Contours with Smoothstep interpolation
Figure 3.26: Armadillo - Image Space Suggestive Contours without Smoothstep interpolation

Figure 3.27: Armadillo - Image Space Suggestive Contours with Median filter
Figure 3.28: Armadillo - Contours only
Figure 3.29: Lucy - Object Space Suggestive Contours

Figure 3.30: Lucy - Image Space Suggestive Contours with Smoothstep interpolation
Figure 3.31: Lucy - Image Space Suggestive Contours without Smoothstep interpolation

Figure 3.32: Lucy - Image Space Suggestive Contours with Median filter
Figure 3.33: Lucy - Contours only
Figure 3.34: Lion - Object Space Suggestive Contours

Figure 3.35: Lion - Image space algorithm with Smoothstep interpolation
Figure 3.36: Lion - Image Space Suggestive Contours without smoothstep interpolation

Figure 3.37: Lion - Image Space Suggestive Contours with Median filter
Figure 3.38: Lion - Contours only
Figure 3.39: Brain - Object Space Suggestive Contours

Figure 3.40: Brain - Image Space Suggestive Contours with Smoothstep interpolation
Figure 3.41: Brain - Image Space Suggestive Contours without Smoothstep interpolation

Figure 3.42: Brain - Image Space Suggestive Contours with Median filter
Figure 3.43: Brain - Contours only
Figure 3.44: Elephant - Object Space Suggestive Contours

Figure 3.45: Elephant - Image Space Suggestive Contours with Smoothstep interpolation
Figure 3.46: Elephant - Image Space Suggestive Contours without Smoothstep interpolation

Figure 3.47: Elephant - Image Space Suggestive Contours with Median filter
Figure 3.48: Elephant - Contours only
Figure 3.49: Hippo - Object Space Suggestive Contours

Figure 3.50: Hippo - Image Space Suggestive Contours with Smoothstep interpolation
Figure 3.51: Hippo - Image Space Suggestive Contours without Smoothstep interpolation

Figure 3.52: Hippo - Image Space Suggestive Contours with median filter
Figure 3.53: Hippo - Contours only
Figure 3.54: Lucy - Image Space Suggestive Contours with radius 1

Figure 3.55: Lucy - Image Space Suggestive Contours with radius 3
Figure 3.56: Lucy - Image Space Suggestive Contours with radius 5
Figure 3.57: Femur - Image Space Suggestive Contours with 3x3 filter kernel and Smoothstep interpolation

Figure 3.58: Femur - Image Space Sobel edge detection with 3x3 filter kernel and Smoothstep interpolation results in thicker lines than respective suggestive Contour algorithm with equal kernel size
Figure 3.59: Shell - Image Space Sobel edge detection

Figure 3.60: Shell - Image Space Suggestive Contours (radius matched to fit image 3.3.1.2.1)
3.3.2 Hybrid Algorithms

3.3.2.1 Wireframe Contours

3.3.2.1.1 Results
Figure 3.61: Flange - Image Space Suggestive Contours with Median filter combined with Image Space Sobel edge detection

Figure 3.62: Flange - Image Space Sobel edge detection drawn in white and Image Space Suggestive Contours drawn in black
Figure 3.63: Six Cubes - hybrid Wireframe Contours with small line width

Figure 3.64: Six Cubes - hybrid Wireframe Contours with large line width
Figure 3.65: Torus - hybrid Wireframe Contours at low model resolution

Figure 3.66: Torus - hybrid Wireframe Contours at high model resolution
Figure 3.67: Elephant - hybrid Wireframe Contours
3.3.2.2 Offset Contours
3.3.2.2.1 Results

3.3.3 Object Space Algorithms
3.3.3.1 Creases
3.3.3.1.1 Results
Figure 3.68: Foot Bones - Contours rendered with hybrid Polygon Offset Method

Figure 3.69: Two Box Cloth - Contours rendered with hybrid Polygon Offset Method
Chapter 4

Example Program
Chapter 5

API

Algorithm is the superclass for all algorithms and is not to be instantiated directly. Algorithm provides two methods:

public void begin()

Starts using this algorithm and registers the algorithm with the renderer. The first call to begin() puts the renderer in NPR mode, which means you can’t use hint(ENABLE_DEPTH_SORT), and you have to use NPR lighting. Processing lighting is disabled.

public void end()

Ends using this algorithm and deregisters the algorithm with the renderer. The last algorithm calling end() ends NPR mode.

5.1 Shading

Shader is the superclass for all shading algorithms and is not to be instantiated directly.

5.1.1 Look-up Table

Shading is based on look-up tables.

public LookUpTable(int xpos, int ypos)

Creates a look-up table with the specified position.

Parameters:

- xpos - horizontal position of the look-up table
- ypos - vertical position of the look-up table

public LookUpTable(int xpos, int ypos)
Creates a look-up table with the specified position from a file. The file extension is .lut.

**Parameters:**

- **file** - the file name the look-up table should be loaded from
- **xpos** - horizontal position of the look-up table
- **ypos** - vertical position of the look-up table

```java
public LookUpTable(String file, int xpos, int ypos,
                    int height, PFont font, int fontSize)
```

Creates a look-up table with the specified position from a file. The file extension is .lut. Uses specified font and font size for display.

**Parameters:**

- **file** - the file name the look-up table should be loaded from
- **xpos** - horizontal position of the look-up table
- **ypos** - vertical position of the look-up table
- **height** - the vertical size of the look-up table
- **font** - the font to be used
- **fontSize** - the font size to be used

```java
public void setRange(int start, int end, int value)
```

Sets a segment of the lookup table to a gray value.

**Parameters:**

- **start** - the start of the segment. Accepted values are 0 to 255.
- **end** - the end of the segment. Accepted values are 0 to 255.
- **value** - the gray value of the segment. Accepted values are 0 to 255.

```java
public void setGradient(int start, int end)
```

Sets a segment of the look-up table to a gradient. The start and end gray values of the segment are determined from the last gray value of the segment to the left and the first gray value of the segment to the right respectively.

**Parameters:**

- **start** - the start of the segment. Accepted values are 0 to 255.
- **end** - the end of the segment. Accepted values are 0 to 255.

```java
public void save(String file)
```
Saves a look-up table to a file. The file extension is .lut.

**Parameters:**
- *file* - the file name

**public void draw()**

Draws a look-up table to the screen for interactive use.

**public void disable()**

Disables a look-up table. A disabled look-up table will be drawn in gray and will not react to mouse input. Useful in conjunction with a GUI when shading is disabled in the GUI.

**public void enable()**

Enables a previously disabled lookup-table for interactive use.

### 5.1.2 Diffuse Shader

**public DiffuseShader()**

Creates a DiffuseShader object and initializes it.

**public void setLookUpTable(LookUpTable lut)**

Attaches a look-up table to the shader.

**Parameters:**
- *lut* - the look-up table to use with the shader.

### 5.2 Shadows

Shadow is the superclass for all shading algorithms and is not to be instantiated directly.

**public void setBrightness(float brightness)**

Sets the intensity of the shadow, that is, simulates the effect of an ambient light.

**Parameters:**
- *brightness* - intensity of the shadow. Accepted values are 0.0 to 1.0.
5.2.1 Dual Map Shadows

public DualMapShadow()

Creates a DualMapShadow object and initializes it.

public void setMapSize(int size)

Sets the shadow map size. As a rule of thumb, the shadow map size should be at least twice the size of your screen width/height. For best performance, use a power of two size.

Parameters:

size - the shadow map size.

public void setOffset(float offset)

Dual shadow maps use a constant offset to avoid self-shadowing artifacts. In most cases the default offset should be sufficient.

Parameters:

offset - a constant offset.

5.3 Line Drawings

Lines is the superclass for all line drawing algorithm. It provides for line width and color. Lines is not intended to be instantiated directly.

public void setWidth(float width)

Sets the line width.

Parameters:

width - line width.

public void setColor(int gray)

Sets the line color to a gray value. Alpha will be set to 255.

Parameters:

gray - the gray value between 0 and 255.

public void setColor(int gray, int alpha)

Sets the line color to a gray value and the alpha to specified alpha value.

Parameters:

gray - the gray value between 0 and 255.
alpha - the alpha value between 0 and 255.

public void setColor(int r, int g, int b)

Sets the line color to specified RGB values. Alpha will be set to 255.

Parameters:

  r - the red value between 0 and 255.
  g - the green value between 0 and 255.
  b - the blue value between 0 and 255.

public void setColor(int r, int g, int b, int a)

Sets the line color to specified RGBA value.

Parameters:

  r - the red value between 0 and 255.
  g - the green value between 0 and 255.
  b - the blue value between 0 and 255.
  a - the alpha value between 0 and 255.

public void setColor(float gray)

Sets the line color to a gray value. Alpha will be set to 1.0.

Parameters:

  gray - the gray value between 0.0 and 1.0.

public void setColor(float gray, float alpha)

Sets the line color to a gray value and the alpha to specified alpha value.

Parameters:

  gray - the gray value between 0.0 and 1.0.
  alpha - the alpha value between 0.0 and 1.0.

public void setColor(float gray, float alpha)

Sets the line color to specified RGB values. Alpha will be set to 1.0.

Parameters:

  r - the red value between 0.0 and 1.0.
  g - the green value between 0.0 and 1.0.
  b - the blue value between 0.0 and 1.0.
public void setColor(float r, float g, float b, float a)
Sets the line color to specified RGBA value.

Parameters:
- **r** - the red value between 0.0 and 1.0.
- **g** - the green value between 0.0 and 1.0.
- **b** - the blue value between 0.0 and 1.0.
- **a** - the alpha value between 0.0 and 1.0.

5.3.1 Sobel edge detection

public SobelLines()
Creates a SobelLines object and initializes it.

public void useGaussian(boolean useGaussian)
Specifies whether to run gaussian filtering before sobel edge detection. This suppresses weaker lines with noisy objects. It also results in a certain broadening of lines and has an antialiasing effect.

Parameters:
- **useGaussian** - true if gaussian is to be used, false if not.

public void setGaussianSize(int size)
Sets the filter kernel size for gaussian filtering.

Parameters:
- **size** - size of the filter kernel. Accepted values are 3 and 5.

public void useSmoothstep(boolean useSmoothstep)
Specifies whether to use smoothstep interpolation.

Parameters:
- **useSmoothstep** - true if smoothstep should be used, false if not.

public void setSmoothstepStart(float start)
Sets the start value for smoothstep interpolation.

Parameters:
- **start** - the start value for smoothstep interpolation.

public void setSmoothstepEnd(float end)
Sets the end value for smoothstep interpolation.

Parameters:
- **end** - the end value for smoothstep interpolation.
5.3.2 Suggestive Contours

public void setWidth(float width)
Sets the radius of the circle-shaped filter mask.

Parameters:
width - the radius. Will be casted to an integer.

public void useSmoothstep(boolean useSmoothstep)
Specifies whether to use smoothstep interpolation.

Parameters:
useSmoothstep - true if smoothstep should be used, false if not.

public void setSmoothstepStart(float start)
Sets the start value for smoothstep interpolation.

Parameters:
start - the start value for smoothstep interpolation.

public void setSmoothstepEnd(float end)
Sets the end value for smoothstep interpolation.

Parameters:
end - the end value for smoothstep interpolation.

public void useMedian(boolean useMedian)
Specifies whether a median filter should be run after detecting suggestive contours.

Parameters:
useMedian - true if median is to be used, false if not.

5.3.3 Wireframe Contours

public WireframeContours()
Creates a WireframeContours object and initializes it.

5.3.4 Offset Contours

public OffsetContours()
Creates a OffsetContours object and initializes it.

setWidth(float width)
Sets the polygon offset.

Parameters:
width - amount of the polygon offset.
5.3.5 Boxy Ridges

public BoxyRidges()

Creates a BoxyRidges object.

public void setThreshold(float threshold)

Sets the threshold for the dot product of angles between normals of adjacent faces. Controls how many lines are detected.

Parameters:

threshold - the threshold.

public void drawHiddenLines(boolean drawHiddenLines)

Specifies whether hidden lines should be drawn.

Parameters:

drawHiddenLines - true if hidden lines are to be drawn, false if not. The default is false.

public void stippleHiddenLines(boolean drawHiddenLines)

Specifies whether hidden lines should be stippled.

Parameters:

drawHiddenLines - true if hidden lines are to be stippled, false if not. The default is false.

5.3.6 Boxy Valleys

public BoxyValleys()

Creates a BoxyValleys object.

public void setThreshold(float threshold)

Sets the threshold for the dot product of angles between normals of adjacent faces. Controls how many lines are detected.

Parameters:

threshold - the threshold.

public void drawHiddenLines(boolean drawHiddenLines)

Specifies whether hidden lines should be drawn.

Parameters:
**drawHiddenLines** - true if hidden lines are to be drawn, false if not. The default is false.

```java
public void stippleHiddenLines(boolean drawHiddenLines)
```

Specifies whether hidden lines should be stippled.

**Parameters:**

- **drawHiddenLines** - true if hidden lines are to be stippled, false if not. The default is false.

5.4 Developing Custom Algorithms

5.4.1 GLSL API

```java
GLSL glsl;

void setup()
{
    // ... normal setup stuff ...
    glsl=new GLSL();
    glsl.loadVertexShader("myShader.vert");
    glsl.loadFragmentShader("myShader.frag");
    glsl.useShaders();
}

void draw()
{
    // might need a ((PGraphicsOpenGL)g).beginGL(); here
    glsl.startShader();
    // draw stuff
    glsl.endShader();
}
```

5.4.2 Rendering to Textures
Chapter 6

Performance
Chapter 7

Further Work

stylization of lines

Figure 7.1: ?
Chapter 8

Conclusion

The implementation presented in this thesis has shown that with efficient image space, hybrid, and precomputed object space algorithms, a broad range of stylistic effects can be achieved.

Yet, the contributions of this thesis are not only the implementation of NPR algorithms like cel shading, stylized shadows and various line rendering algorithms, making the renderer suitable for a wide range of applications including interactive technical, scientific, medical and narrative illustration, architectural visualizations and animated comic style rendering.

While the main focus was on NPR algorithms developed at Princeton University, where the thesis was written, the framework allows any NPR researcher to plug in their algorithms to make them available to a wider public. It also offers an extensive, easy-to-use API that makes it possible for programmers to develop their own glsl shaders, whereas shading support was very limited in the original PGraphicsOpenGL renderer.

Furthermore, the architecture of the renderer results in a considerable speedup compared to the PGraphicsOpenGL renderer, as it performs all transformations and lighting in hardware and uses display lists for geometry and framebuffer objects for off-screen rendering, avoiding unnecessary copying from the framebuffer to textures.

In fact, as a side effect of the project, the architecture lends itself even to fast, high-quality non-non-photorealistic rendering, including (soft) shadows, which was not possible with Processing up to date due to the limitations of the PGraphicsOpenGL renderer. (See chapter 2) For example, Phong Shading can now easily be implemented in hardware on top of the renderer presented in this paper. An excellent tutorial on how to approach this can be found at http://www.lighthouse3d.com/opengl/glsl/.

The hope is that this renderer will spark interest in the young field of NPR, and a synergy effect between researchers on the one side and artists and practitioners on the other side can be achieved.
Bibliography


Glossary

2D  two-dimensional (space)

3D  three-dimensional (space)

JavaDoc  Java API documentation tool

obj  text-based geometry definition file format developed by Wavefront Technologies

PGraphicsOpenGL  Processing OpenGL renderer, using the JOGL interface