The G^2-Buffer Framework

Mike Eißele* Daniel Weiskopf* Thomas Ertl*

Institute of Visualization and Interactive Systems
University of Stuttgart

Abstract

The geometric buffer (G-buffer) is a well-known approach to implement image-based rendering algorithms. We propose a framework that maps the G-buffer concept and associated image-space operations to the graphics processing unit (GPU). This GPU-G-buffer (G^2-buffer) framework consists of two major components: first, a texture-based representation of the G-buffer attributes; second, an implementation of the image-space operations by fragment programs. The G-buffer setup and the actual rendering algorithm are described in a plain text file, whose syntax is an extended version of the Effect Files in DirectX or CgFX. Our approach provides a fast and easy method to implement G-buffer algorithms that automatically exploit the GPU with its high memory bandwidth and processing power. We demonstrate the usage of the G^2-buffer for a couple of applications in non-photorealistic rendering.

1 Introduction

Image-space rendering techniques have always been an active area of research, since all renderings finally end up in image space. Advances in the field of non-photorealistic rendering (NPR) also show the usefulness of image-space approaches. An important class of NPR rendering styles operates directly in image space, such as halftoning, screening, stippling, or hatching methods.

Saito and Takahashi propose a general framework for image-space rendering algorithms called G-buffer [ST90]. This buffer forms an enriched image space, also referred to as 2.5D image space, whereby each pixel of the image space holds arbitrary additional information, such as normal, depth, or texture coordinates. In this paper we refer to this extended pixel as fragment. It is obvious that the memory requirements and the needed processing power of this approach are not beneficial for interactive applications. However, real-time rendering is possible on modern GPUs by exploiting their high memory bandwidth (up to 30.4GB/sec) and processing power. Recent advances in the programmability of graphics hardware allows for a very fast and flexible implementation of the G-buffer. Our proposed framework combines the G-buffer with the functionality of programmable GPUs to a novel generic G^2-buffer (GPU-G-buffer) concept.

* {eissele | weiskopf | ertl}@vis.uni-stuttgart.de

2 Previous Work

Non-photorealistic rendering is an area of application where image-space techniques are widely used. An overview of NPR techniques can be found in the textbooks by Gooch and Gooch [GG01], Strothotte and Schlechtweg [SS02], and in the SIGGRAPH Course on NPR [GSS+99]. Saito and Takahasi show how to use the G-buffer to implement different rendering styles [ST90]. Recently, Mitchell et al. presented an algorithm operating in image space to detect object outlines [MBC02]. Their work is also strongly based on the G-buffer concept, although they use application-specific implementations and not a generic framework. As another example NPR halftoning rendering style by Freudenberg et al. [FMS02] can readily be converted to image-space operations by using normal, intensity, and texture coordinates as G-buffer attributes. Also the hatching approach by Praun et al. [PHWF01] can be described via a G-buffer algorithm. Implementing the pen-and-ink style rendering method by Freudenberg et al. [FMS01] with the G-buffer framework offers new possibilities to improve the results. For example, silhouettes could be detected and emphasized in image space. Finally, Secord [Sec02] presents a method to generate stipple drawings; he also mentions an interactive technique that can be converted to a G-buffer description.
Often image-space techniques are used to post process renderings of 3D scenes and to further enhance the visual appearance. This recent trend can be observed in many GPU demonstration programs and also in new important video game titles. With this correspondence of post processing algorithms and image-space techniques, the proposed G^2-buffer framework could be seen as the logical add-on to Microsoft's Effect Edit tool included in the DirectX 9.0 SDK [Mic02].

3 The G-Buffer Concept

The G-buffer (geometric buffer) approach by Saito and Takahashi separates rendering in two parts. In a first step, the G-buffer attributes are generated on a 2D domain. There are many attributes that can be used in a G-buffer. The simplest ones come from the standard graphics pipeline itself; for example, G-buffer attributes can store color, alpha, depth, texture coordinates, or the normal per fragment of the image space. Other typical G-buffer attributes are object ID, world-space position, screen-space position, or parameterization of higher-order surfaces. Additionally, further attributes might be implemented such as the screen-space velocity for motion blur effects or light-space depth values for shadow calculations. Most of these attributes are computed in object space and stored for each fragment of the image plane. The subsequent rendering passes, referred to as G-buffer operations, receive this information and have therefore access to more than only simple 2D color information. Thus, the G-buffer could be thought of as a 2.5D enriched image space, since the data it holds is no longer restricted to the 2D image plane.
Arbitrary calculations upon this data structure are applied to evaluate the final result per fragment. This approach makes it possible to implement some classes of algorithms that cannot be realized by object-space techniques. One of them are filter operations, which are also widely used in computer vision and image post processing. Depending on the filter kernel, different computations are performed, for example edge detection by the Sobel

operator or bluring by Gaussian filtering. Also morphological operations like dilate or erode could be realized. There are far more filter operations described in the computer vision [Dav96] and image processing [Rus02] literature, and most of them could be implemented in a G-buffer technique.
Non-photorealistic rendering is another typical application for the G-buffer. Many algorithms for artistic drawing styles work in image space, just as a real artist would do. Hatching or stippling illustrations generated in image space do not suffer from perspective distortions, introduced by most object-space techniques. More important the rendering algorithms are not restricted by object borders and so, for example, hatching lines can cross object boundaries to achieve a more natural drawing style. Some NPR algorithms are exclusive image-space operations which can automatically be implemented via the G^2-buffer framework.
Deferred shading is a third field of application for the G-buffer. The basic idea is to separate shading and lighting from the other parts of rendering. This has several advantages: If the lighting conditions of a scene are changed but the geometry remains static, only the lighting has to be recomputed. Moreover, if a scene suffers from massive overdraw, a conventional object-space algorithm will compute the lighting for each fragment regardless if it is overdrawn by other, closer fragments. In contrast, deferred shading evaluates the illumination only for the visible pixels.

4 Effect Files

In this section we give a brief overview of Effect Files and their usage. The functionality of GPUs has advanced greatly over the last two years and so the programming of today's GPUs has become quite complex. States of the graphics pipeline have to be set up, vertex shader and pixel shader programs need to be specified. Traditionally, many of these operations are spread over different parts of the program source code (for example, the C++ code). Therefore, Effect Files (FX Files) were introduced in DirectX 8 and further enhanced in DirectX 9 [Mic02] to simplify the programming of the GPU. They encapsulate the whole setup and programming of the graphics pipeline within a description of a so-called *Effect*. An Effect is described by a character string or an external text file and thus this concept is mainly referred to as *Effect File*. The source code responsible for the GPU programming is decoupled from the main program code, which deals with the CPU programming. In this way, a better maintainability and readability of the source code for both parts is achieved. Additionally, the description of an Effect is platform-independent; thus, it could be reused on other platforms, such as CgFX [Fer03].
Effect Files provide support to define named parameters within an Effect description. The initial content is set up within the description; additionally, the values could be modified by the outside framework to influence the behavior of an Effect. Further, a mechanism is supported to set up all states of the graphics pipeline (such as culling mode, blending mode, or z-test) for each rendering pass defined within an Effect File. The programming of vertex shader and pixel shader unit of actual GPUs is supported in two different languages: High-level shading language (HLSL) to describe shader programs in a C-like manner and

a specific assembly language for machine-level programming. Within an Effect, each pass may contain its own set of state settings, vertex shader definition, and pixel shader definition. In terms of Effect Files, a collection of one or more passes is combined to a technique description. Since different GPUs have different features, it is also possible to define alternative techniques within one Effect. Out of these, DirectX can automatically choose the best available technique for the currently used GPU when the Effect is bound to the graphics hardware.

5 Extended Effect Files for G-Buffers

The concept of Effect Files is mainly designed to describe object-space rendering techniques. In contrast, the G-buffer is a pure image-space approach and therefore additional functionality is needed to make Effect Files useful in this context. The following two essential aspects of the G-buffer cannot be configured in the original Effect Files: first, the G-buffer attribute storage; second, the G-buffer operations and their associated data flow.
Multiple 2D render-target textures are used to model the G^2-buffer storage, one for each attribute. Effect Files do not provide a mechanism to define render targets, therefore the attributes are defined as a 2D texture within the description of the Effect. Our framework recognizes these definitions and updates the attribute texture for every frame. The algorithms to generate the attribute values are typically object-space methods that are also implemented via Effect Files.
A G-buffer operation is specified by a technique consisting of one or more rendering passes. The computations are performed per-fragment, thus the G-buffer operation is described via a set of pixel shader programs. Often one pass needs information generated by a previous pass. Therefore, the possibility to model the data flow, i.e., to say which pass result is used as input for another pass, must be introduced to the Effect description. Figure 1 shows all possible data flow configurations that should be configurable within an extended Effect File.
Textures can be used as data input to pixel shader programs, as it is already possible in standard Effect Files. To store the result of a rendering pass into a texture that can afterwards be used as input for subsequent pixel shader programs, render-target textures, referred to as *pass-result textures*, are used. As mentioned before, Effect Files do not support render-target settings, thus pass-result textures are defined as ordinary textures within the Effect. The G^2-buffer framework detects them and allocates corresponding render-target textures in which the result of the rendering pass is kept. During rendering the framework automatically activates the according render target for each pass of the G-buffer operation. The inputs of the single passes that are labeled with *Buf In* refer to an indirect input of the data already contained in the frame buffer via alpha test, alpha blending, z test, or stencil test.
Through this mechanism it is possible to have fragment programs of arbitrary length, as subsequent passes can continue the calculation of prior passes. A similar idea is proposed by William R. Mark and Kekoa Proudfoot [MP01]. More importantly, subsequent passes can access results of previous calculations of neighboring fragments in image space. Only in this way filter operations can be implemented.

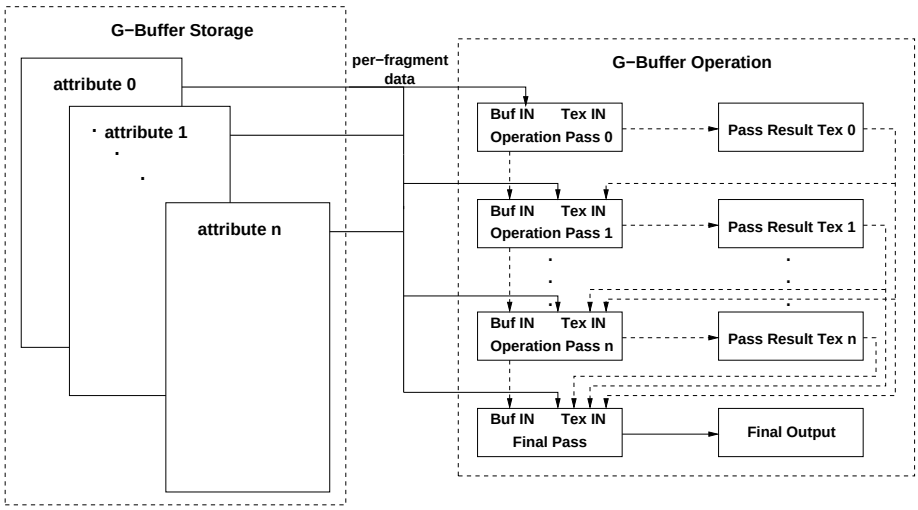

Figure 1: Data flow diagram for extended Effect Files.

To add the above mentioned enhancements, Effect descriptions provide named parameters that could be processed by the framework. Thus, our proposed framework follows the design of Microsoft's Effect Edit tool [Mic02] which also enriches the functionality of Effect Files through this mechanism. As there are many G-buffer attributes available and the final rendering possibly needs only a few of them, the setup should only contain the attributes that are finally used in the G-buffer operations. The description shown in Figure 2 defines a G-buffer attribute setup and operation that performs an edge detection and dilation to broaden them. The result of this algorithm is illustrated in the Figure 3.

6 Implementation

The implementation of the G^2-Buffer framework is based on DirectX 9.0 and its Effect File feature. DirectX provides the functionality to read and parse an Effect File. The extensions are encoded by the naming of the textures and attributes within the Effect File definition. The G^2-buffer framework examines the description and arranges the setup of the attributes and pass-result textures. The actual storage of the G-buffer attributes and intermediate processing results is realized via render-target textures. Therefore, any data transfer of the per-pixel information between the GPU and the CPU is avoided.
For pass-result and attribute textures, the number of components, the precision per element, and the relative scale factor for the size of the whole texture are specified in the Effect. These parameters are set up by modifiers within the name of the declared texture.
Since the allocated texture size, dimension, and precision can be configured to match exactly the requirements, any memory overhead is avoided. For example, it is possible to setup the normal attribute with three components in 16 bit fixed-point precision while the

```
// declaration of a G-buffer "intensity"
Texture GBAintensity_A8R8G8B8;
// define pass-result texture for pass 0
Texture PASS0_A8R8G8B8;
// texel size; set by the framework
float4 TexelSize;
// constants
float edgeThreshold = 0.09f;
float dilateThreshold = 9.0f;
// filter vertex shader output
struct VS_EDGE_OUT
{
  float4 Pos  : POSITION;
  float2 Tex0 : TEXCOORD0;
  float2 Tex1 : TEXCOORD1;
  float2 Tex2 : TEXCOORD2;
};
// edge vertex shader
VS_EDGE_OUT VSEdgeFilter(
  float3 Pos  : POSITION,
  float2 Tex0 : TEXCOORD0,
  float2 Tex1 : TEXCOORD1)
{
  VS_EDGE_OUT Out;
  Out.Pos.xyz  = Pos;
  Out.Pos.w = 1.0f;
  Out.Tex0 = Tex0;
  float2 lowerTexels = Tex0 - TexelSize.wy;
  Out.Tex1 = lowerTexels - TexelSize.xw;
  Out.Tex2 = lowerTexels + TexelSize.xw;
  return Out;
}
// dilate vertex shader output
struct VS_DILATE_OUT
{
  float4 Pos  : POSITION;
  float2 Tex0 : TEXCOORD0;
  float4 Tex1 : TEXCOORD1;
  float4 Tex2 : TEXCOORD2;
  float4 Tex3 : TEXCOORD3;
  float4 Tex4 : TEXCOORD4;
};
// dilate vertex shader
VS_DILATE_OUT VSDilate(
  float3 Pos  : POSITION,
  float2 Tex0 : TEXCOORD0,
  float2 Tex1 : TEXCOORD1)
{
  VS_DILATE_OUT Out;
  Out.Pos.xyz  = Pos;
  Out.Pos.w = 1.0f;
  Out.Tex0 = Tex0;
  Out.Tex1.xy = Tex0 + TexelSize.xw;
  Out.Tex1.zw = Tex0 - TexelSize.xw;
  Out.Tex2.xy = Tex0 + TexelSize.wy;
  Out.Tex2.zw = Tex0 - TexelSize.wy;
  Out.Tex3.xy = Tex0 + TexelSize.xy;
  Out.Tex3.zw = Tex0 - TexelSize.xy;
  Out.Tex4.xy = Tex0 - TexelSize.xw
    + TexelSize.wy;
  Out.Tex4.zw = Tex0 + TexelSize.xw
    - TexelSize.wy;
  return Out;
}
// edge pixel shader sampler
sampler objInt = sampler_state
{
  texture = <GBAintensity_A8R8G8B8>;
  AddressU  = WRAP;
  AddressV  = WRAP;
  MIPFILTER = POINT;
  MINFILTER = POINT;
};
// edge pixel shader
float4 PSEdgeFilter(VS_EDGE_OUT In) : COLOR
{
  float4 centerTexel = tex2D(objInt,In.Tex0);
  float4 lowLeftTexel = tex2D(objInt,In.Tex1);
  float4 lowRightTexel = tex2D(objInt,In.Tex2);
  float4 objColor;
  // calc roberts cross
  float edgeWeight =
    abs(centerTexel.x - lowLeftTexel.x)
    + abs(centerTexel.x - lowRightTexel.x);
  float4 edgeColor = 1.0f;
  if (edgeWeight > edgeThreshold)
    edgeColor = 0.0f;
  return edgeColor;
}
// dilate pixel shader sampler
sampler edge = sampler_state
{
  texture = <PASS0_A8R8G8B8>;
  AddressU  = CLAMP;
  AddressV  = CLAMP;
  MIPFILTER = POINT;
  MINFILTER = POINT;
};
// dilate pixel shader
float4 PSDilate(VS_DILATE_OUT In) : COLOR
{
  // read 3x3 texel block
  float4 centerTexel = tex2D(edge,In.Tex0);
  float4 leftTexel = tex2D(edge,In.Tex1.xy);
  float4 rightTexel = tex2D(edge,In.Tex1.zw);
  float4 upperTexel = tex2D(edge,In.Tex2.xy);
  float4 lowerTexel = tex2D(edge,In.Tex2.zw);
  float4 upperLeftTexel = tex2D(edge,In.Tex3.xy);
  float4 upperRightTexel = tex2D(edge,In.Tex3.zw);
  float4 lowerLeftTexel = tex2D(edge,In.Tex4.xy);
  float4 lowerRightTexel = tex2D(edge,In.Tex4.zw);
  // evaluate dilation
  float dilateWeight = centerTexel + leftTexel
    + rightTexel + upperTexel + lowerTexel
    + upperLeftTexel + upperRightTexel
    + lowerLeftTexel + lowerRightTexel;
  float dilateEdgeColor = 1.0f;
  if (dilateWeight < dilateThreshold)
    dilateEdgeColor = 0.0f;
  return dilateEdgeColor;
}
// definition of techniques
technique T0
{
  pass P0
  {
      VertexShader = compile vs_1_1 VSEdgeFilter();
      PixelShader  = compile ps_2_0 PSEdgeFilter();
  }
  pass P1
  {
      VertexShader = compile vs_1_1 VSDilate();
      PixelShader  = compile ps_2_0 PSDilate();
  }
}
```

Figure 2: Implementation of an edge detector followed by a dilate operation.

color attribute is generated with four components of 8 bit precision. The choice of the precision and number of components per element is restricted by the available formats for render-target textures. For example, formats with 8 and 16 bit fixed-point precision and

16 and 32 bit floating-point precision are supported on an ATI Radeon 9700. The size of the textures is not restricted to powers of two, i.e., non-power-of-two textures are utilized. This feature can be disabled for pixel shaders that use dependent texture fetches, since most GPUs do not support texture-coordinate calculations for such textures within a pixel shader program.
The first rendering step is to update the G-buffer attributes that are required during the subsequent execution of the G-buffer operations. Afterwards, the framework processes the G-buffer operation by rendering the corresponding passes step by step. Before a pass is rendered the framework sets the appropriate render target, which is either the framebuffer or a pass-result texture. Then a quadrilateral that matches exactly the size of the image plane is drawn, where the texture coordinates of the vertices are set up to represent the fragment positions in image space. During the fragment processing these texture coordinates are used to access the G-buffer attributes or results of prior passes. Since the assigned texture coordinates must exactly address the corresponding texel, special care has to be taken to ensure a one-to-one pixel-to-texel mapping.
The size of the texel must be known by the Effect to access adjacent texels. Since the size of the pass-result or attribute textures depend on the framebuffer dimension, the texel size changes whenever the framebuffer is resized. To overcome this problem, the texel size is defined as a vector within the effect description and changed by the framework after each viewport resize.
In general the output of the final pass of a G-buffer operation is stored in the framebuffer. Additionally, our framework allows the user to save the final result either to an image file or combine a sequence of images into a video file. On the other hand, a special G-buffer attribute is provided that decompresses a still image of a video stream each time the G-buffer operation is executed. Through these two features it is possible to realize programmable video processing utilizing the GPU.

7 Results

The performance of the implementation depends on various factors. The costs of the first part where the attributes are computed and stored in the G-buffer are affected by the number and complexity of the used attributes. Many operations on G-buffers use standard graphics pipeline properties, as color or normal, for whose computation the GPUs are optimized. However, for arbitrary attributes the computational cost might differ significantly, depending on the operations performed to evaluate the attribute.
As a G-buffer operation is an image-space technique, its performance is linearly dependent on the number of fragments on the image plane. Other parameters that affect the performance are the complexity of the fragment processing, number of passes in the G-buffer operation, and the precision and dimension of the intermediate results. Stated differently, the main part of the implementation is rasterization bound: Since each pass of a G-buffer operation just renders a screen-size quadrilateral, vertex processing is no restriction here.
The performance impact of a fast CPU is negligible as all computations are handled within the GPU. Because there are many factors that influence the performance of the framework,

the presented framerates are only valid for the shown specific example cases. All illustrations are rendered on a Windows XP machine with an Intel P4 2.8GHz CPU and an ATI Radeon 9700 PRO GPU.
Figure 3 shows the result of the afore mentioned Effect code in Figure 2. The G-buffer operation performs a Robert's cross edge detection and a subsequent dilation. This example is rendered at 64 fps on a 1024×1024 viewport.
A comparison of two edge detection filters is illustrated in Figure 4. On the left side the edges are detected via a Robert's cross filter, while on the right side a Sobel filter is used. Both images show the same scene with equal camera parameters. The Robert's cross G-buffer operation runs at 185 fps on a 1024×1024 viewport, while the Sobel filter results in 80 fps with same viewport dimensions. Only minor differences are noticeable in the visual appearance while the framerates differ significantly. With such short examples the

Figure 3: Dilated edges, implemented as a G-buffer operation.

Figure 4: Comparison of edge detection methods: Robert's cross filter (left) and Sobel filter (right).

G^2-buffer framework can be used to check if more sophisticated computations are required or not.
Figure 5 compares the result of a Gaussian blur (right hand side) versus the original image (left hand side). The G-buffer attributes are rendered at three times the dimension of the actual framebuffer. Afterwards the Gaussian filter is applied to average the value of 3×3 pixel blocks to a single pixel. This self-programmed supersampling renders the example scene at 145 fps on a 512×512 output framebuffer, while the rendering without filtering runs at 560 fps.
Figure 6 shows the last example using a NPR screening technique. Several G-buffer operations are performed to calculate the final result. The contrast of the scene is enhanced, the edges are detected and dilated, and finally the screening technique is applied. The example rendered at 78 fps on a 1024×768 viewport.

8 Conclusion

We have proposed the G^2-buffer framework, which offers a flexible, generic, and abstract way to implement G-buffer operations on graphics hardware. The design of the framework avoids unnecessary data traffic between GPU and CPU. Therefore, the implementation of image-space methods benefit from the hugh memory bandwidth and processing power of modern graphics hardware. Our framework allows all typical G-buffer approaches to be implemented, such as deferred shading, NPR, or image filtering. Processing of video streams is also possible, whereby all frames of a source video are filtered by a G-buffer operation on the GPU and afterwards recompressed to a new video file. We have integrated the G^2-buffer into Effect Files framework; since only minor extensions to the original syntax

Figure 5: Gaussian blur operation. The left image shows the original rendering, the right picture illustrates the Gauss filtered rendering.

are introduced, our system can be easily adopted by developers and existing code can be reused.
In future work, further features of GPUs could be considered to accelerate the G-buffer framework: Multiple render targets could be adopted to generate multiple G-buffer attributes or multiple intermediate results within one rendering pass [MBC02]; multiple element textures (MET) could improve the utilization of texture memory; masking of empty regions in image space by the early z-test. Finally, the framework could be reimplemented for other platforms by using CgFX [Fer03] or OpenGL 2.0 [3dl02].

Acknowledgements

The second author thanks the “Landesstiftung Baden-Württemberg” for support.

References

[3dl02] *OpenGL 2.0 Overview*. http://www.3dlabs.com/support/developer, 2002. 3Dlabs Inc. Ltd.

[Dav96] E. R. Davies. *Machine Vision: Theory, Algorithms, Practicalities.* Academic Press, 2nd. edition, 1996.

[Fer03] Randima Fernando. *CgFX Overview*. http://developer.nvidia.com/Cg, 2003. Nvidia Corporation.

[FMS01] Bert Freudenberg, Maic Masuch und Thomas Strothotte. *Walk-Through Illustrations: Frame-Coherent Pen-and-Ink Style in a Game Engine*. In: Computer Graphics Forum: Proceedings Eurographics 2001, volume 20, p. 184–191, 2001.

[FMS02] Bert Freudenberg, Maic Masuch und Thomas Strothotte. *Real-Time Halftoning: A Primitive For Non-Photorealistic Shading*. In: Proceedings of the 13th Eurographics Workshop on Rendering Techniques, p. 227–231, 2002.

[GG01] Bruce Gooch und Amy Gooch. *Non-Photorealistic Rendering*. AK Peters Ltd, 2001.

[GSS+99] Stuart Green, David Salesin, Simon Schofield, Aaron Hertzmann, Peter Litwinowicz, Amy Gooch, Cassidy Curtis und Bruce Gooch. *Non-Photorealistic Rendering*. SIGGRAPH 99 Course Notes, 1999.

[MBC02] Jason L. Mitchell, Chris Brennan und Drew Card. *Real-Time Image-Space Outlining for Non-Photorealistic Rendering*. SIGGRAPH 2002 Sketch, 2002. ATI Technologies Inc.

[Mic02] Microsoft. *DirectX 9.0 Programmer's Reference*. http://www.microsoft.com/downloads, 2002. Microsoft Corporation.

[MP01] William R. Mark und Kekoa Proudfoot. *The F-buffer: A Rasterization-Order FIFO Buffer for Multi-Pass Rendering*. In: ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, p. 57–64, 2001.

[PHWF01] Emil Praun, Hugues Hoppe, Matthew Webb und Adam Finkelstein. *Real-Time Hatching*. In: Proceedings of ACM SIGGRAPH 2001, p. 579–584, 2001.

[Rus02] John C. Russ. *The Image Processing Handbook*. CRC Press, 4th. edition, 2002.

[Sec02] Adrian Secord. *Weighted Voronoi Stippling*. In: NPAR 2002: Second International Symposium on Non-Photorealistic Rendering, p. 27–43, 2002.

[SS02] Thomas Strothotte und Stefan Schlechtweg. *Non-Photorealistic Computer Graphics – Modeling, Rendering, and Animation*. Morgan Kaufmann, 2002.

[ST90] Takafumi Saito und Tokiichiro Takahashi. *Comprehensible Rendering of 3-D Shapes*. In: Computer Graphics (Proceedings of ACM SIGGRAPH 90), volume 24, p. 197–206, 1990.

Figure 6: NPR screening technique with dilated edges.