Rendering Optimisations for Stylised Sketching

Holger Winnemöller*
University of Cape Town

Shaun Bangay†
Rhodes University

Abstract

We present work that specifically pertains to the rendering stage of stylised, non-photorealistic sketching. While a substantial body of work has been published on geometric optimisations, surface topologies, space-algorithms and natural media simulation, rendering-specific issues are rarely discussed in-depth even though they are often acknowledged. We investigate the most common stylised sketching approaches and identify possible rendering optimisations. In particular, we define uncertainty-functions, which are used to describe a human-error component, discuss how these pertain to geometric perturbation and textured silhouette sketching and explain how they can be cached to improve performance. Temporal coherence, which poses a problem for textured silhouette sketching, is addressed by means of an easily computed visibility-function. Lastly, we produce an effective yet surprisingly simple solution to seamless hatching, which commonly presents a large computational overhead, by using 3-D textures in a novel fashion. All our optimisations are cost-effective, easy to implement and work in conjunction with most existing algorithms.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling—Boundary representations; I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling—Curve,surface, solid, and object representations; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Colour, shading, shadowing, and texture

Keywords: 3D Texturing; Coal, Pencil, Chalk, Hatching, rendering; Edge-fading; Non-photorealistic rendering; Temporal coherence; Uncertainty-functions

1 Introduction

Amongst the established non-photorealistic (NPR) rendering styles (cartoon, sketching & painterly) the sketching style has by far attracted the most attention of researchers. As we show in Section 2, most published work addresses such issues as:

- Geometric Optimisations (edge traversal, mesh-simplification)
- Image-space / Object-space / Hybrid-space considerations
- Surface topologies
- Natural Media Simulation

As yet, very little work addresses rendering-specific problems for stylised sketching and we attempt to narrow this gap. In order to identify rendering-specific problem areas for stylised sketching, we recognise the following prevailing sketching styles:

- *Outlining* or Silhouette rendering – In this style, lines are drawn which usually approximate the silhouette of an object in order to define its shape while neglecting such properties as texture or shading.
- *Hatching* or Cross-hatching – In this style, the main focus is the simulation of shading information and/or surface topology through the use of hatching-strokes of varying densities.

Within the outlining style, there are two main approaches to rendering a silhouette in a stylised fashion:

- *Geometric Perturbations*: The geometry of an object is randomised in order to simulate realistic imperfections. Rendering is usually performed using line-primitives.
- *Stroke-textures*: The geometry of the object is left intact, but imperfections and/or natural media are simulated through textures, which are applied to bill-boarded (viewer-oriented) extensions of the original geometry. The drawing primitives are therefore textured quadrilaterals or triangle-strips.

Typical rendering-specific issues exist with each of these rendering styles. We identify the main problems as:

- Overhead in computing uncertainty-functions[1] on-line and applying them homogenously (General Outlining)
- Temporal incoherence for silhouette rendering (Textured Outlining)
- Smooth shading-boundaries for textured hatching (Hatching)

While discussing these problems, we produce highly optimised sample solutions to the most commonly used sketching approaches. Since our work focuses on the rendering side of sketching, it can seamlessly extend many of the optimisations and techniques of other authors.

*e-mail: hwinnemo@cs.uct.ac.za
†e-mail:S.Bangay@cs.ru.ac.za

[1] See Section 3 for a definition of *uncertainty-functions.*

2 Related Work

2.1 Geometric Perturbation and Texture Outlining

[Markosian et al. 1997] introduce a system which can render in a variety of outlining styles (exact – following object geometry, randomly perturbed and with texture attributes, hinting at pencil or chalk lines). Their main innovation is a very rapid, probabilistic edge detection algorithm.

[Northrup and Markosian 2000] present a hybrid algorithm that works in both object and image space. Firstly, the silhouette of an object is detected in object-space as described in earlier work by [Markosian et al. 1997]. Then, the silhouette is projected into image-space and a series of operations and decisions is performed to generate long, continuous curves from the polygonal fragments comprising the visible silhouette in image space.

Deussen et al show a rather specialised use of a Pencil-and-Ink technique in [Deussen and Strothotte 2000]. Their main goal is to render trees (but shrubs and bushes can also be produced) in a distinct NPR style.

[Raskar and Cohen 1999] use "*fattening of lines*" (extending the back-facing polygon of an edge outwards) and limited texturing to produce what they call "*Image Precision Silhouette Edges*". While their method could more efficiently be implemented using texturing and thick lines (such as can be produced with OpenGL) they take great care to avoid resolution problems associated with depth-buffers of limited bit-depth through processes called z-scaling and line-fattening. Raskar later developed an extension of this approach in [Raskar 2001], where he shows how a standard rendering pipeline can be modified to incorporate the generation of Silhouettes, Ridges, Valleys and Intersections.

2.2 Hatching

[Praun et al. 2001] use tonal art maps, which are intimately related to mip-maps (and are implemented through these), to place stylised strokes (dots, single lines or hatches) on the surface of 3D models. This is done in object-space (as opposed to the image-space approach followed by [Lake et al. 2000] and ourselves) to follow isoparameter curves on the surface of objects.

[Rössl and Kobbelt 2000] use a three-stage system, which works on triangulated meshes to produce line-art drawings in a technical style. In a multi-step semi-automatic process, hatches are placed onto surface geometry in order to aide manual conversion of 3-D models to line-art drawings.

Another rather technical approach by the same authors [Rössl et al. 2000] is mostly concerned with surface topologies and analysis of curvature gradients on these surfaces.

[Lake et al. 2000] use projective texturing with four different textures to implement their pencil stroke (hatching style) *inker*. As textures can only be applied to complete geometric primitives (as opposed to parts thereof), they have to implement a geometry sub-division algorithm, to align primitive boundaries with shading boundaries.

[Hall 1999] presents a technique which he calls "Comic-strip rendering", but which we categorise as a primarily hatching style. It involves a two-stage rendering scheme and procedural textures to generate hatches on surface geometry and is intended to be plugged into a ray-tracing application.

[Sousa and Buchanan 1999a], [Sousa and Buchanan 1999b] devise a realistic natural media simulation system. They model physically realistic pencils and are able to apply these to surface geometry.

3 Stylised Rendering and Uncertainty-functions

Amongst other criteria already mentioned, we can differentiate between *artistic/manual* and *technical* sketching. While technical sketching is probably the most commonly found in the real world (manuals, textbooks, illustrations, etc.) it is also more easily automated than artistic sketching. This is because both styles rely on the same basic drawing primitives (lines, edges, dots, etc.) and techniques (outlining, hatching, stippling, etc.), but the artistic sketching requires an additional artistic or human component.

This is usually achieved by introducing imperfections into the rendered image. While various possible imperfections are defined and discussed in the literature, we would like to formalise this discussion by introducing the concept of *uncertainty-functions*.

Definition 1 *An uncertainty-function is the functional representation of a human-error component. The input to an uncertainty-function is the original object, while the output of an uncertainty-function is the object with added human-error variations.*

Various basic uncertainty-functions can be concatenated in order to produce a combined uncertainty-function as in Figure 1.

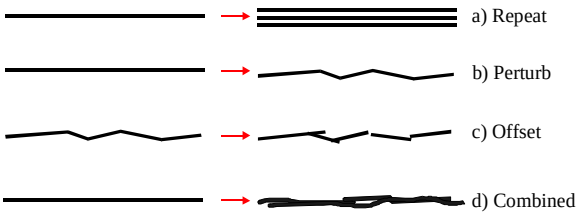

Figure 1: Individual and combined Uncertainty-functions

We can identify uncertainty-functions in all manual sketch renderers. They are most explicitly present in geometric perturbation sketching, where object geometry is usually displaced and perturbed (i.e. Figure 1b and c), before being rendered into the screen-buffer. More often, we find them pre-concatenated in the form of textures (e.g. a bitmap of Figure 1d), which allows for efficient caching of the uncertainty-functions. Since caching can considerably increase performance, we discuss in Section 3.1 how it can be achieved for the geometric perturbation approach. The consequences of applying cached and therefore static uncertainty-functions are discussed in 3.2.

3.1 Object segmentation

The texturing approach represents a form of caching of concatenated uncertainty functions, thus saving the cost of performing concatenations on-line. Next to other advantages like thick lines or natural media textures, there are also disadvantages to the texturing approach, foremost the need for bill-boarding, i.e. the screen-space orientation of textured elements towards the viewer.

We therefore discuss how caching of uncertainty-functions similar to the texturing approach is possible for purely geometric perturbation sketching.

Displaylists are available on all modern 3-D graphics cards and allow for fast retrieval of rendering information (vertices, colours, etc.) by storing relevant data on the card's internal memory. It seems obvious how displaylists could be used to cache the necessary uncertainty-functions for geometric perturbation sketching. Unfortunately, displaylists can only hold static data which would be equivalent to applying uncertainty-functions once-off upon loading

an object. This would result in a static representation of the object and look unconvincing under animation. What we want to achieve is a dynamically changing look similar to a rough stop-motion sequence.

Our solution is therefore to cache uncertainty-functions as applied to only one edge instead of the entire object. In this way all the required human-error imperfections can be stored in various displaylists and recalled when needed.

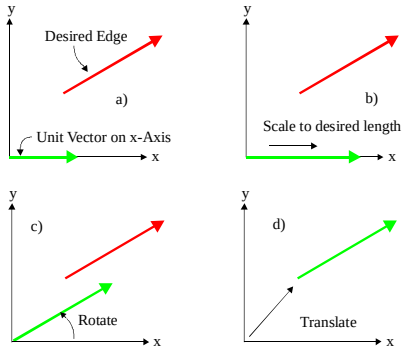

Figure 2: Preparing an Edge for use with Displaylists

The way we utilise these displaylists to render an object is as follows. First, we dissect the object into its relevant edges (for a closer discussion on relevant edges and importance-functions, see [Winnemöller 2002]) and then compute the transformations required to transform a unit-vector along an arbitrary axis (we have chosen the x-axis for simplicity) to each of these edges. These transformations, as shown in Figure 2, can all be expressed in matrix notation and multiplied together to form a single transformation matrix for each edge. Now, instead of rendering a unit-vector and transforming it, we render a given displaylist instead so that the transformation is applied to it as a whole. The very simple and efficient algorithm implementing this approach is shown in Listing 1.

3.2 Stretch Factors of Cached Uncertainty-Functions

Since uncertainty-functions cached in displaylists are just as static as those cached in textures, the following discussions pertain to both rendering solutions.

3.2.1 Longitudinal Stretch Factors

Whether scaling a displaylist as in Figure 2b) or selecting texture co-ordinates for the same purpose, we have to deal with the issue of applying the same uncertainty-functions to edges of different lengths. As becomes evident from Figure 3, the same uncertainty-function can look considerably different when stretched by different factors. To produce a homogenous look, we have to address this matter.

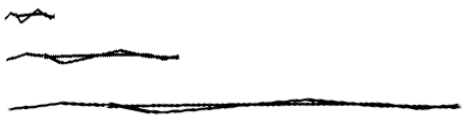

Figure 3: Different longitudinal Strech-factors for a Sketchy Line

For the texturing approach there exists a simple solution as we can use texture clamping and/or repetition in order to fix the uncertainty-density with respect to length.

The geometric perturbation method is a bit more difficult to deal with, because it lacks the inherent clamping and repetition capabilities of texturing. Our solution here is to increase the number of available uncertainty displaylists. We already have various displaylists to allow us to animate the uncertainty-function of a whole object, now we simply increase this number slightly, to allow for edges of different lengths. We do this by computing a histogram of the edge-lengths of an object and assigning several displaylists to each bin so that displaylists of different bins are related in their uncertainty-densities through their respective bin-sizes. Within each bin, edges are stretched to fit the bin-size. The homogeneity of this approach then depends on the number of bins allocated.

The measure we use to determine uncertainty-density is chosen relative to the object-space dimensions of an object, so that uncertainty-functions scale with the object.

3.2.2 Lateral Stretch Factors

By pure geometric definition an edge does not have a width associated with it, but in both the texturing approach and the geometric perturbation approach, we artificially widen edges in order to apply uncertainty-functions to them ([Raskar and Cohen 1999] call this *fattening of lines*). While it would be easy to keep this lateral stretch factor constant throughout the object, this can conceal finer object detail, as we show in Figure 4a). Object detail with high spatial curvature like the nose, eyes and horns consists of many short edges, which are over-emphasized when a constant lateral stretch factor is applied. Figure 4b) shows our approach in contrast. Widths are scaled by an amount proportional to the length of an edge. While this deals effectively with short edges, it produces a similar problem for long edges. We address this problem by clamping the width-factor to a maximum value if necessary.

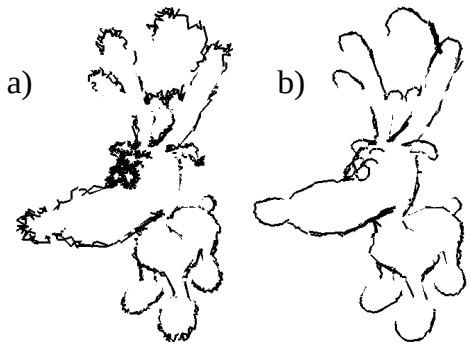

Figure 4: Different Stretch-width Approaches: a) Constant; b) Edge-length dependent

As in Section 3.2.1, both the width-scaling factor and the clamping value are chosen relative to the object-space dimensions of the object. For additional examples of objects rendered with our choice of longitudinal and lateral stretch-factors, see Figure 12a) and b).

4 Temporal Coherence

In renderers like the geometric perturbation sketcher discussed in Section 3, where a fair amount of visual noise is present, the effects of edge-popping are disguised by the constantly changing silhouette. Most texture sketch renderers, on the other hand, do not use animated uncertainty-functions and the sudden appearance and disappearance of edges becomes noticeable and distracting.

The root of the problem is the silhouette-condition, which determines whether a given edge lies on the silhouette of an object for the current view-point. Figure 5 shows the possible configurations. Configuration b) has the viewer looking onto two front-facing faces and the edge joining the two faces is not part of the silhouette. The same holds for configuration c), where both faces are back-facing with respect to the viewer. Only in configuration a), where an edge is part of a front-facing and a back-facing face, is the edge-condition fulfilled. The signs mark whether the dot-product of the normal of a face with the view-vector is positive or negative. We could therefore rephrase the silhouette-condition as being fulfilled if the product of the above-mentioned dot-products is negative.

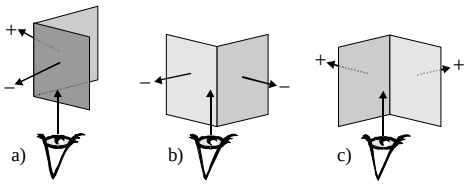

Figure 5: Possible Edge-configurations

It should be obvious that this condition is binary, i.e. it is either fulfilled or not and the changeover is instantaneous. This is the reason for edge-popping. Our solution is therefore to change the condition to a fuzzy one, which means that an edge can either be on the silhouette, not on the silhouette or *somewhat* on the silhouette. This allows us to fade edges in, which are about to become visible, and fade those out which are about to become invisible, but the problem then becomes *how can we predict what edges are about to do?* Our answer is as simple as it is effective, and in fact has already been mentioned above.

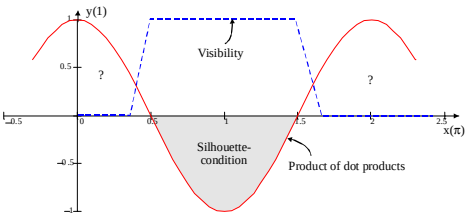

Figure 6: Visibility-function for Edges

The solid curve in Figure 6 shows a first approximation of the product of dot-products measure (the actual curve depends on the angle between faces and orientation towards the viewer). In the shaded region, where the curve is negative, the silhouette-condition is fulfilled, while in the other regions it is not. The dotted curve, representing edge-visibility, would normally exhibit discontinuities at the boundaries of the silhouette-condition region (i.e. shaded region), but we defined a slightly sloping visibility-function instead. Two points should be noted. Firstly, the shape of the solid curve is in general not identical to that in Figure 6 , but can be assumed to be symmetrical. Secondly, the shape of the visibility function is arbitrary and can easily be user-defined and customised. Its rising point determines how many edges are at least partially visible and therefore has an influence on the number of edges rendered. The width of the slope determines over how many degrees edges are faded in and out. The slope itself need not be a straight line at all and could be modified to implement accelerated fading. For the sake of performance and configurability, we implemented our visibility-function in form of a look-up table and achieve pleasing visual results for a fade-span of 5-8 degrees.

In summary, the product of dot-product measure in connection with a suitable visibility function can be used to eliminate edge-popping completely in an extremely simple and efficient fashion.

5 Smooth Hatching Boundaries

Hatching or cross-hatching is another popular NPR sketching style and commonly implemented by texturing the surface of an object with aligned stroke textures. The stroke density and placement of these textures is chosen so that textures with fewer strokes are subsets of those with more strokes as shown in Figure 7a). The local choice of texture is performed as a function of light-intensity (usually the diffuse component of the Phong reflection model) at a given vertex in order to represent shading through stroke-density.

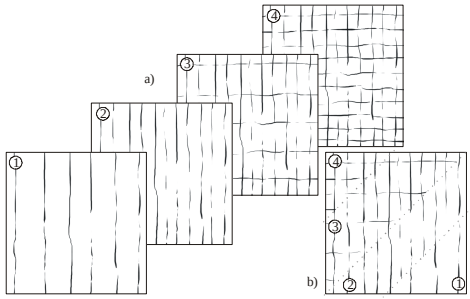

Figure 7: Hatching Textures: a) Various interdependent Densities; b) Combined

Various methods exist to place strokes onto the surface geometry of an object. Some authors affix textures in screen-space by applying projective texturing (e.g. [Lake et al. 2000]), others parameterise their surfaces in order to affix textures in object-space (e.g. [Praun et al. 2001]), but all face the same problem: Achieving smooth shading boundaries. This problem arises, because shading boundaries (the dotted diagonal lines in Figure 7b) do in most cases not align with geometric boundaries. To address this issue, several solutions exist. [Lake et al. 2000] sub-divide their surface geometry until shading boundaries do coincide with geometric boundaries. [Praun et al. 2001] use lapped textures and blend 6 textures in multiple rendering passes to solve the problem. Most other solutions also use multiple rendering passes, usually dependent on the number of discrete hatching textures. All of these solutions represent a considerable computational overhead as well as an increase in geometric information which has to pass through the graphics pipeline.

We present a novel solution, which works in a single rendering pass, independently on the number of hatching textures to be used. The keyword to our approach is 3-D textures. These have been around for a while, but so far not been recruited for NPR hatching.

Figure 8 shows how we form a 3-D texture by stacking the above-mentioned stroke-textures on top of each other. While doing so, we repeat textures several times, but with increasing transparency so that textures of lesser stroke-density will appear lighter and blend with background elements. Our method can be used with any previous approach, as we place no restrictions on our 3-D texture being fixed in screen-space or object-space. The lighting value of a vertex can be used directly to index into the added, third dimension and texture interpolation can be used to blend smoothly between adjacent textures.

Two disadvantages exist with our method. Firstly, a 3-D texture of sufficient spatial resolution requires a lot of texture memory

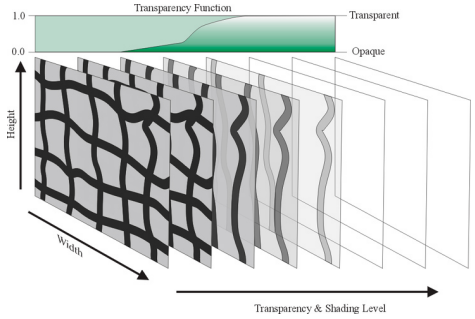

Figure 8: A stack of 2-D textures forms a 3-D texture

(typically about 2MB), but we argue that modern graphics easily offer this amount of storage capacity. In addition to that, conventional 2-D textures would also require the same order of magnitude of memory. Secondly, while 3-D textures have been part of the OpenGL specification since version 1.2, they are not yet implemented in hardware by many vendors. We believe this will change with new generations of graphics cards.

6 Results

Even though the optimisations discussed in this paper are intended to be used in connection with existing systems, and performance values are therefore application-dependent, we demonstrate the possible performance increase of our object-segmentation approach in Figure 9. The recursive algorithm trace shows the performance of a geometric perturbation sketcher, which calculates uncertainty-functions on-line. The object-segmentation trace represents the same renderer with caching of uncertainty-functions. Compared to the first trace, caching has in this case improved performance 26-133%. The texture-sketching trace is shown for comparison purposes. In absolute terms, we are able to render 800-15,000 edges at 25-270 frames per second on a PentiumII 500 with GeForce 256, without any geometric approximations (e.g. smart edge-traversal, etc.)

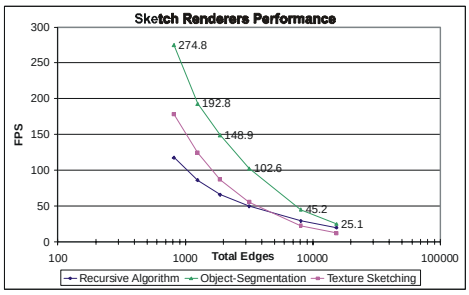

Figure 9: Performance Comparison

Examples of the increased temporal coherence achieved by our visibility-function can be seen in the Edge-fading animation accompanying this paper. Figure 10 illustrates a typical scenario. The edges marked by arrows are slowly faded out with our optimisation instead of suddenly disappearing as would normally be the case.

Another animation is provided, demonstrating the smooth continuous hatching-boundaries that can be produced in a single rendering pass using our proposed 3-D texture approach. In this animation a single quadrilateral is textured using a 3-D texture. Each

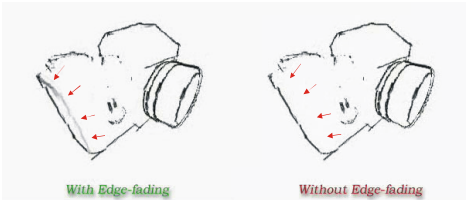

Figure 10: Screen-shot of Temporal Coherence Animation

vertex is assigned a different lighting value according to Figure 11 and the quadrilateral is rotated relative to the light-source so show the hatching-boundaries move over the surface.

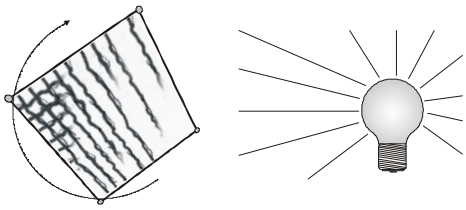

Figure 11: Illustrated screen-shot of 3-D texturing Animation

Rendering examples of a real-time system which uses our optimisations can be see in Figure 12.

7 Conclusion

We have categorised sketching styles from a rendering point-of-view into:

- Outlining styles
  - Geometric Perturbations
  - Stroke-Textures
- Hatching style

Key rendering problems associated with each of these styles were successfully addressed.

The computational overhead of perturbing object geometry on-line was investigated. We defined so-called *uncertainty-functions*, which allow for a formalised discussion of human-error components in artistic rendering. By examining uncertainty-functions in textured outlining, we developed an equivalent method of caching these functions for the geometric perturbation style. Applying cached uncertainty-functions in either style was solved by defining appropriate scaling-functions for longitudinal and lateral stretching.

We provided an efficient and powerful solution to the temporal coherence problem of textured outlining (edge-popping), by devising a scheme in which edges are automatically faded in and out of a scene as they are about to appear or disappear. Our solution uses a user-defined visibility-function, which can be set to adjust the number of edges rendered, the range over which edges are faded, as well as fading speed and acceleration.

We identified the key problem of hatch-rendering as that of producing smooth hatching boundaries. Previous solutions are computationally expensive and/or require multiple rendering passes.

```
Render
   For each Edge E do
      If silhouette(E)              // if edge is visible
         PushMatrixStack            // push current matrix so we can restore it
            MultMatrixOntoStack M(E) // Apply Matrix of current Edge
            call Sketchy_Line       // Draw displaylist (cached uncertainty-functions)
         PopMatrixStack             // restore previous matrix
      End If
   Next Edge
End Render
```

Listing 1: Pseudocode to sketch geometrically perturbed lines using cached display-lists

We show that 3-D textures are ideally suited for hatch-style rendering and how they can be used to produce smooth, geometry-independent hatching-boundaries in a single rendering pass for as many hatching-textures as desired.

In summary, we identify the main rendering problems of established stylised rendering styles and provide viable and efficient solutions, all of which can be used in connection with most existing sketching systems.

Acknowledgements

The authors would like to thank Rhodes University for its moral and financial support of this project.

All object models publicly and freely available from www.meta3d.com, www.its-ming.com, www.3dup.com or www.3dcafe.com.

References

DEUSSEN, O., AND STROTHOTTE, T. 2000. Computer-Generated Pen-and-Ink Illustration of Trees. In *Proceedings of SIGGRAPH 2000 (New Orleans, July 2000)*, K. Akeley, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 13–18.

HALL, P. 1999. Nonphotorealistic Rendering by Q-mapping. *Computer Graphics Forum 18*, 1 (Mar.), 27–39.

LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. 2000. Stylized Rendering Techniques for Scalable Real-Time 3DAnimation. In *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 2000)*, ACM, 13–20.

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. 1997. Real-Time Nonphotorealistic Rendering. In *Proceedings of SIGGRAPH'97 (Los Angeles, Aug. 1997)*, T. Whitted, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 415–420.

NORTHRUP, J. D., AND MARKOSIAN, L. 2000. Artistic Silhouettes: A Hybrid Approach. In *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 2000)*, ACM, 31–37.

PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. 2001. Real-Time Hatching. In *Proceedings of SIGGRAPH'2001 (Los Angeles, Aug. 2001)*, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 581–586.

RASKAR, R., AND COHEN, M. 1999. Image Precision Silhouette Edges. In *Proceedings of the Conference on the 1999 Symposium on interactive 3D Graphics*, ACM Press, New York, S. N. Spencer, Ed., 135–140.

RASKAR, R. 2001. Hardware Support for Non-photorealistic Rendering. In *Proceedings of SIGGRAPH'01, Graphics Hardware*, ACM Press.

RÖSSL, C., AND KOBBELT, L. 2000. Line Art Rendering of 3D-Models. In *Proceedings of Pacific Graphics 2000 (Hong Kong, Oct. 200)*, IEEE Computer Society Press, Los Alamitos, B. A. Barsky, Y. Shinagawa, and W. Wang, Eds., 87–96.

RÖSSL, C., KOBBELT, L., AND SEIDEL, H.-P. 2000. Line Art Rendering of Triangulated Surfaces Using Discrete Lines of Curvature. In *Proceedings of WSCG 2000 (Pilzeň, Feb. 2000)*, V. Skala, Ed., 168–175.

SOUSA, M. C., AND BUCHANAN, J. W. 1999. Computer-Generated Graphite Pencil Rendering of 3D Polygonal Models. In *Proceedings of EuroGraphics'99 (Milano, Italy, Sept.1999)*, NCC Blackwell Ltd, Oxford, P. Brunet and R. Scopigno, Eds., 195–207.

SOUSA, M. C., AND BUCHANAN, J. W. 1999. Computer-Generated Pencil Drawing. In *Proceedings of SKIGraph'99*.

WINNEMÖLLER, H. 2002. *Implementing Non-photorealistic Rendering Enhancements with Real-Time Performance*. Master's thesis, Rhodes University, South Africa.

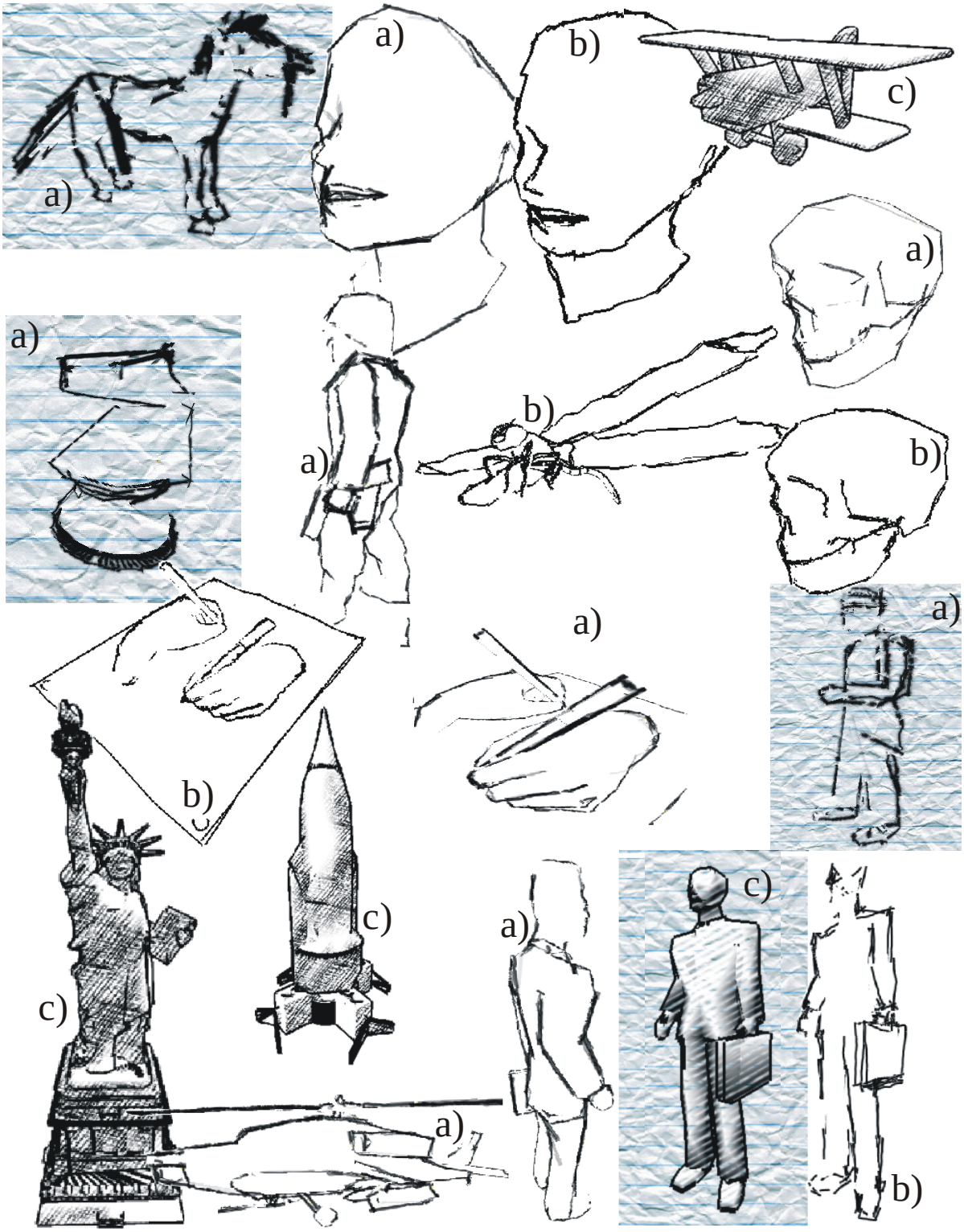

Figure 12: Sketching examples of a real-time system using our optimisations: all objects labelled a) use Textured Sketching; those labelled b) are rendered with Geometric perturbations; and those with label c) use Textured Hatching. Similar objects are grouped to allow for comparison.