EUROGRAPHICS 2001 / A. Chalmers and T.-M. Rhyne
(Guest Editors)

Volume 20 (*2001*), *Number 3*

Walk-Through Illustrations: Frame-Coherent Pen-and-Ink Style in a Game Engine

Bert Freudenberg, Maic Masuch, Thomas Strothotte

Institut für Simulation und Graphik, Otto-von-Guericke-Universität Magdeburg, Magdeburg, Germany

Abstract

In this paper we show how a game engine designed to generate photorealistic images can be extended to produce non-photorealistic and hybrid renditions. We introduce new hardware-based methods to accomplish pen-and-ink illustrations. The combination of the highly optimized processing of a game engine and the use of hardware for NPR algorithms yields real-time animation of pen-and-ink illustrations.

The particular advance of this method is that it yields the first real-time, frame-coherent pen-and-ink animations which maintain both tone and texture.

1. Introduction

In recent years 3D game engines have made a tremendous advance in the graphical presentation of interactive worlds. A number of optimization and specialization techniques for the graphics pipeline and the representation of the scene geometry allow the depiction of a graphically rich world in varying detail. This development makes 3D game engines interesting also for non-gaming applications, like for instance virtual reality visualizations. These allow users to be immersed in another world, as the computer simulates the complete spatial environment which can be visited with inexpensive equipment in 3D.

While the main scope of current game graphics still lies in generating photorealistic images, other fields of application favor the use of non-photorealistic images, either for scientific visualization or for artistic expression. Lately, computer graphics research has turned to the field of non-photorealistic rendering and there are a number of solutions for the generation of still images. If, however, we consider non-photorealistic animation or even real-time rendering, we encounter two challenging problems: The first is the maintenance of frame-to-frame coherence and the second is the ability of a 3D engine to render non-photorealistic images of a given scene in real-time.

We present an approach to non-photorealistic visualization of a 3D scene that adopts a 3D game engine. The engine facilitates a hybrid rendering pipeline that allows specification of the rendering style on a per-object basis.

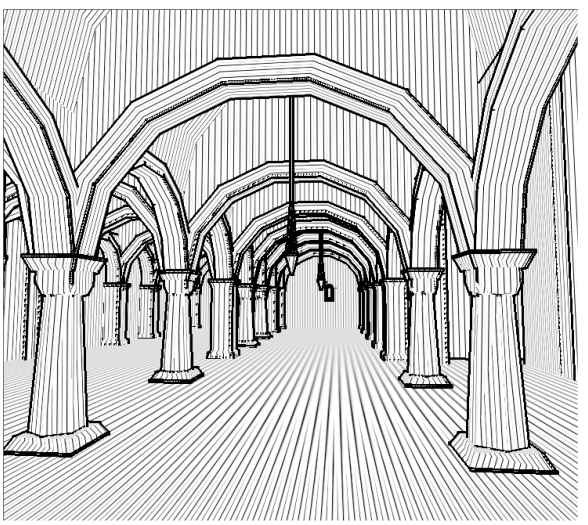

Figure 1: *A medieval hall rendered in pen-and-ink style.*

The paper is organized as follows: First, we give a brief overview of the field of non-photorealistic imaging in computer games, then we discuss the benefits of non-photorealistic visualizations in combination with a game engine. In Section 3 we describe the architecture and key elements of a non-photorealistic game engine. We introduce new techniques for real-time pen-and-ink-style rendering. Section 4 covers the results accompanied by a number of sample images of various scenes rendered with the 3D en-

gine. Finally, in Section 5 we summarize our results and point out areas of future work.

2. Computer Games and Non-Photorealistic Rendering

Inspecting current computer games, we have to conclude that no game achieves a true *photorealistic* look at all. Thus, games in general could be called *non-photorealistic* to some degree. However, in a broader sense, the rendering style can be called *photorealistic* with respect to realistic shapes, spatial relationships and surface details. The development of current 3D engines aims at more and more realism, mainly through an extensive use of elaborated textures. There are only very few computer games that incorporate non-photorealistic style elements. First, there are games using a traditional cartoon rendering style as in MONKEY ISLAND III or BROKEN SWORD, but as these are based solely on 2D data, we do not take them into further account.

The most original looking game PENCIL WHIPPED adopts hand-drawn texture maps in a conventional 3D game engine to achieve a sketchy look[1]. However, it is clearly visible that the hand drawn textures are simply mapped onto the scene geometry, more resembling graffiti on a wall than an animated drawing. The NPRQUAKE-Project carried out by GLEICHER et al. pursues a different approach[2]. They developed a method to change the appearance of a 3D application by intercepting the application's calls to the graphics library and replacing them with different drawing primitives that depict a sketch style, a blue-print and a brush-stroke style. Their approach uses the core GLQUAKE engine and maintains the original game-play while achieving a sketchy, hand-drawn look. The non-photorealistic style elements do not preserve frame coherence, as the outlines of objects vary from frame to frame. This results in a lively, but also unsteady and disturbing experience.

The preservance of frame coherence is one of the most intriguing challenges of NPR animation. Although there are an increasing number of non-photorealistic rendering systems aiming at the generation of various different styles (impressionistic paintings, technical illustrations, pen-and-ink etc.), there are few systems dealing with the goal of frame-to-frame coherence. For instance, the system presented by SALESIN et al. creates pen-and-ink illustrations[3, 4], a style similar to what we are aiming at. It is based on *stroke textures*, collections of strokes arranged in different patterns, to generate texture and tone. However, like other comparable approaches, stochastic methods are used to model the line deviations of a hand drawing, which makes them unsuitable for the generation of animation.

MEIER et al. presented an offline rendering system for painterly animation. In this system particles are placed on the 3D model and transformed to strokes in the image[5]. This approach and related work suggest that in order to maintain frame coherence there has to be some sort of connection between the model in object space and the artistic strokes in screen-space[6, 7]. First advances in real-time NPR rendering were made by MARKOSIAN[8], LAKE[9] and others. The latter introduced real-time techniques for cartoon and pencil sketch rendering. Their system also generates stylized silhouette edges, but encounters also another problem of NPR animation, the "shower-door" effect. This effect describes the visual discontinuity when particles stick to the screen and not, as they should, to the object, and the object seems to move independently from the strokes it is composed of.

A first step in NPR virtual reality was DISNEY'S ALADDIN[10], a magic carpet ride through a cartoon-shaded world, using carefully adapted hand-drawn texture maps. Recently, a system for the creation of non-photorealistic walk-throughs in real-time was introduced by KLEIN et al.[11]. It is based on image-based rendering methods, using preprocessed images which are captured in advance (from reality or a 3D model) and processed with a stroke-oriented filter. These so called "art-maps" are mapped onto the 3D scene and rendered in real-time.

2.1. Non-Photorealistic Visualization

Non-photorealistic imaging allows the user to simplify an image by omitting details to control the viewer's attention in order to emphasize and deemphasize certain elements of a depicted scene[12]. Furthermore, an artistic non-photorealistic rendition can be more compelling than a synthetic image. We aim for a visualization style with the qualities of pen-and-ink illustrations: Silhouettes and creases on the surface should be clearly drawn, while surface detail is added through hatching. Since only black and white is used for this, color can be used to present additional information.

Using a hybrid rendering pipeline, we can specify the rendition style for each single object. Thus, we gain an additional degree of freedom for the visualization.

- *photorealistic images* use normal photorealistic texture mapping and shading
- *non-photorealistic images* consist of a modified shading, an outline (possibly deviating from the exact shape) and non-photorealistic textures (i.e. ink-maps or hatch-maps)
- *hybrid images* combine photorealistic and non-photorealistic images

Furthermore, by changing the presentation style of an object over time, we gain a completely new style of animation. All these features are integrated in a game engine, as explained in the next sections.

2.2. Benefits of a Game Engine

The aim of a game engine, in general, is to supply an interactive virtual environment for the interaction of one or several players with objects and characters of the virtual world. Although there are numerous different game engines, and even more 3D engines, we can characterize a game engine by its

main components (which, of course, may differ from engine to engine):

- a special purpose *3D engine*, which is optimized for real-time rendering and ensures the generation of a constant frame rate for fluent animation
- a set of rules defining the *behavior* of the virtual world, like kinematic control, dynamic simulation, collision detection and AI for the behavior of non-player characters
- a *network* component supporting multi-player interactions

As discussed in the beginning of this section, 3D engines of current computer games focus almost exclusively on realistic rendering. If we want to use non-photorealistic visualization elements in a 3D environment, we somehow have to combine NPR and game engines.

Instead of adding 3D real-time capabilities to a non-photorealistic renderer, we decided to extend an existing game engine by adding techniques for frame-coherent non-photorealistic rendering. Here, we found the FLY3D-engine developed by WATT and POLICARPO a flexible open-source platform to adopt methods for real-time NPR, yet gaining additional features at low additional costs[13].

3. Non-photorealistic elements in a 3D game engine

The development of 3D game engines aims at providing the player with a highly immersive experience. This involves enhancing speed, realism, and image quality. Techniques employed in 3D engines to achieve speed-up mainly involve preprocessing as much data as possible and let the graphics hardware handle much of the per-frame load. For example, *binary space partitioning* trees are used to rapidly cull geometry outside the view frustum, *potentially visible sets* reduce the number of polygons drawn for each viewpoint, and precalculated *lightmaps* capture the static lighting in the scene. To improve realism, *texture mapping* is used for providing detail and to make realistic surfaces. Not only the surface color is modified, but also *environment mapping* and *bump mapping* is becoming common. To reduce the number of rendering passes required for adding all those layers of realism, *multi-texturing* and flexible blending is supported by the graphics hardware, as well as special texture coordinate generation modes suited for environment mapping or *per-pixel lighting*. The hardware also supports *mip-mapping* and *full scene anti-aliasing* to enhance image quality.

Non-photorealistic rendering, however, presents quite a challenge to a game engine designer. The most notable difference to photorealism is that non-photorealism explicitly uses image-space elements. That is, the pixels generated on the screen are not necessarily a strict projection of the 3D scene objects. The only method commonly provided by graphics hardware and low-level graphics libraries like OpenGL is line drawing. A triangle outline rendered as polyline has a constant width independent of the actual distance of the triangle to the viewer.

For hybrid illustrations, we want both, photorealistic rendering and non-photorealistic drawing styles. How to do the photorealistic part in a game engine is well understood. The interesting question is how to make use of the existing graphics hardware features to achieve a non-photorealistic look. We describe the features of our engine in the following subsections.

3.1. Outlining

Lines are the central element of pen-and-ink-style illustrations. *Silhouettes* are drawn to enhance the visual separation of objects from each other and from the background. Additional lines are used to mark *discontinuities* of the surface depicting the inner structure of an object.

3.1.1. Discontinuities

Finding discontinuities on the model's surface can be done in a preprocessing step. Usually, the angle between two neighboring faces is compared to a fixed threshold to determine if the shared edge represents discontinuity or if it is just an artifact of the polygonal approximation process. This works well for highly tessellated models. But in a low polygon count environment typical for games it is preferable to explicitly define "artistic" edges in the modeling stage that are always drawn[14].

3.1.2. Silhouettes

Silhouettes are view-dependent and therefore silhouette edges need to be determined in every frame. For our intended illustration style we do not need to distinguish between discontinuities and silhouettes, so we can safely apply an optimization: We only need to consider strictly convex, smooth edges. Non-smooth edges are drawn anyway, and smooth edges in concavities cannot be silhouettes. This can significantly reduce the number of edge candidates. For example, in the eight-sided prism used to approximate a cylinder shown in Figure 2, there are 42 edges, but only 8 smooth edges are silhouette candidates when applying the "smooth-and-convex" rule. If an object only contains sharp and non-convex edges, no silhouette detection needs to be performed at runtime at all.

Convexity of an edge is determined in a preprocessing step. For each edge, the unshared vertex of one adjacent

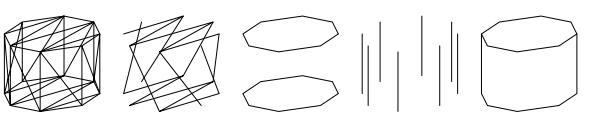

Figure 2: *Silhouette determination: all edges (left), smooth non-convex edges are never drawn (second), sharp edges are always drawn (third), smooth convex edges are silhouette candidates (fourth), result with hidden lines removed (right).*

triangle is checked if it lies in front of the plane of the other triangle. In Figure 3 this is demonstrated with the shared edge $\overline{AC}$ of the triangles *ABC* and *ACD*. To determine if *D* is in front of *ABC*'s plane we can take the dot product of the *ABC*'s normal N_{ABC} and the vector $\overrightarrow{AD}$:

$$\overline{AC} \text{ is strictly convex} \iff (D-A) \cdot N_{ABC} < 0$$

In practice, we compare to a small ε instead of 0 to ensure that co-planar triangles do not become silhouette candidates.

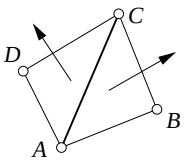

Figure 3: *Determining convexity of an edge*

3.1.3. Drawing edges

In each frame, when the edges to draw were selected as described above, they are rendered as wide line segments using `GL_LINES`. This takes advantage of the graphics hardware depth buffer for hidden line removal. To keep lines from being overdrawn by adjacent faces, `glPolygonOffset` is used. The gaps between adjacent edges are filled using `GL_POINTS` placed at the end of each segment.

3.2. Surface detail and Hatching

Surface detail in traditional pen-and-ink illustrations is added by drawing more complex structures between the outlines (for example, bricks in a wall). Surface curvature can be indicated by hatching, which produces a set of aligned strokes. Both methods have in common that the line width is roughly constant over the image plane, it does not diminish with distance.

3.2.1. Hatch maps

Surface detail in 3D computer games is added via texture mapping. Structures painted into a texture are intended to become smaller with increasing distance. In the case of a pen-and-ink hatching texture, the small structures cause serious moirée patterns. In motion, this produces even worse visual artifacts than in a still image, which prevents frame-coherence.

Figure 4 shows a single tilted rectangle covered with a repeated black-and-white pattern choosing the nearest texel's color for each pixel. The texture scale is adjusted so that in the front, the texture is drawn at its original size, while in the back, it is minified due to perspective foreshortening.

The usual antidote for such problems is mip-mapping[15].

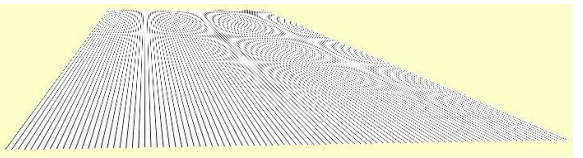

Figure 4: *Unfiltered hatching textures cause moirée patterns.*

Mip-maps consist of a series of textures decreasing in size. The graphics hardware automatically chooses a mip-map level so no texture minification occurs, on a per-pixel basis. The mip-map levels are constructed by scaling the original texture down in a preprocessing step, applying more or less sophisticated filtering techniques. Unfortunately, the normal filtering process applied to the large white areas and relatively thin black lines of a hatching texture causes the lines to vanish into gray soon (Figure 5). All visible surface structure is lost.

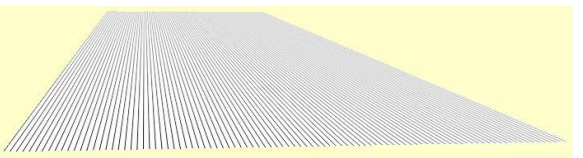

Figure 5: *Normal mip-map filtering fades hatch lines into light gray.*

So, at a first glance, textures seem to be unsuited for implementing pen-and-ink detail. But fortunately, the graphics hardware allows us to set each mip-map level independently. It does not care whether the individual mip-map levels really are "nicely" filtered-down versions of the original image. We would need to construct the mip-map levels in a way that maintains the desired gray-level by spatially distributing black ink on the white background instead of locally raising the gray-level of the texels.

This is indeed possible. The key idea is that mip-mapping essentially displays textures at constant size in screen-space. That means, a one-texel-wide line drawn into each mip-map level will be displayed roughly as a one-pixel-wide line on the screen, independently of the actual object size.

Figure 6 demonstrates the process of constructing one-dimensional *hatch-map* levels that are used as hatching primitive. On the left, the normal mip-map filtering scheme is demonstrated. As can be seen, the lines are black and well-separated in the top level, but they become lighter and closer to each other in the lower levels. On the right, the hatch-map construction scheme is shown. It uses equally-spaced black hatch lines for all mip-map levels. Only in the very lowest levels that are smaller than the desired on-screen line spacing, gray values are used to maintain the tone.

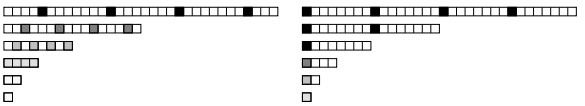

Figure 6: *Normal mip-map (left), hatch-map (right).*

Texture coordinates for game objects are usually assigned interactively in a modeling tool. Hatch-map coordinates are in no way different, except that only one coordinate actually gets used. An object needs to be entirely covered by a repeating texture for hatching. In practice, we use a checkerboard pattern in the modeling program, while at run-time it is substituted by the hatch-map.

For hatch-maps to work, the texture must be scaled so that it is always minified, because the mip-mapping hardware only allows to specify minification textures. On the other hand, to cover a wide depth range, we need many mip-map levels. We adjust the texture scale for an object while modeling, so that the texture appears at original size when the object is at its closest distance to the viewer. The result of applying the constructed hatch-map is shown in Figure 7.

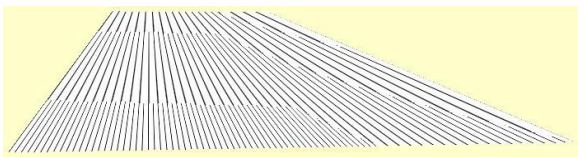

Figure 7: *Hatch-maps in bi-linear texture filtering mode. Four hatch-map levels can be distinguished.*

It can be seen that our hatch-map approach works as expected. Less lines are used in the distance, each line is clearly distinguishable. However, though the over-all tone is maintained, it is not continuous. There is an abrupt switch of hatch-map levels, causing discontinuities in line-width and hatching density.

To overcome this limitations we use tri-linear texture filtering. This is a special texture hardware mode that smoothly blends texels from two adjacent hatch-map levels. Since hatch-map levels are constructed such that lines in one level, when displayed, are spaced exactly twice as wide as in the previous level, the lines fit exactly in-between. This creates the highly desirable effect of smoothly thinning-out hatch lines while maintaining the overall brightness of the image (Figure 8). This is the key for the frame-coherence inherent to our approach.

To give an example, assuming the desired hatch line spacing on screen is 8 pixels, the lower 3 mip-map levels of widths 1, 2, and 4 can not be used. Therefore, a 2048×1 texture gives an usable depth range of $1 : 2^{11-3} = 1 : 256$. The texture could be even larger than that, because, fortunately,

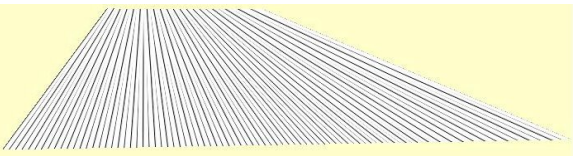

Figure 8: *Hatch-maps in tri-linear texture filtering mode. Strokes are faded out smoothly.*

one-dimensional textures do not use much texture memory. Also, to extend the depth range, the texture scale could be dynamically varied depending on the object's distance to the viewer.

In the case of a very steep viewing angle, anisotropic texture filtering (enabled via the texture_filter_anisotropic OpenGL extension) noticeably improves the image quality. Another factor for better image quality is enabling full scene anti-aliasing. This increases the virtual screen resolution, which causes the wrong mip-map level to be chosen. We compensate this by biasing the level computation using the texture_lod_bias OpenGL extension.

3.2.2. Ink maps

While maintaining a constant stroke width and spacing for hatching is a rather mechanical process, adding details to surfaces is artistically challenging. We looked into constructing *ink-maps* that utilize the same hardware-mip-mapping technique like the one-dimensional hatch-maps, but this time in two dimensions.

The desired surface structure was hand-drawn into a texture map. When we put an object textured with it in our illustration engine, similar problems to the hatch-map problems mentioned above arose. The moirée effect was not quite as disturbing because of the larger structure size, but again, worse in motion. Of course, the drawing rapidly blended into the white background when mip-mapping was applied.

Automatically adjusting the line spacing as we did for the hatch-maps was impossible, because the lines were not evenly spaced. We settled for maintaining the stroke width only, without considering the change in tone. Unfortunately, our initial attempts at automatic filtering gave unsatisfying results (see Figure 9).

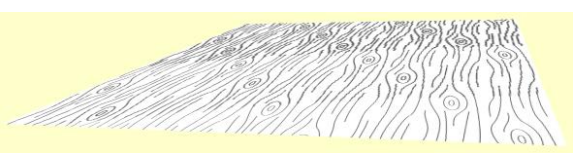

Figure 9: *Ink-map levels automatically constructed by minimum filtering.*

We used a filter that selects the darkest texel out of the four parent texels. While the stroke width was indeed roughly maintained, the lines became fuzzy. Although more sophisticated filtering methods might provide better results, we did not investigate this further.

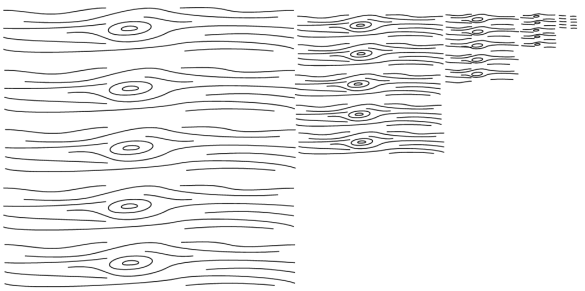

Figure 10: *Ink-map example.*

Instead, we chose to manually construct ink-maps. This also allows an artist more expressiveness than any automatic method can provide, because the artist can control exactly how an ink-map should look like in the distance. The ink-map was drawn again, this time in a vector graphics application. Down-scaling the drawing by 50% maintained the line width. In the lower scale levels, lines were gradually removed until no lines remained. An example of an ink-map constructed like this is shown in Figure 10, while Figures 11 and 12 show renderings of a rectangle with the ink-map applied.

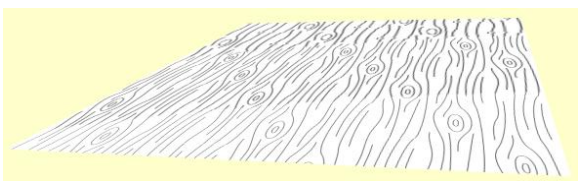

Figure 11: *Rendered levels in a manually constructed ink-map.*

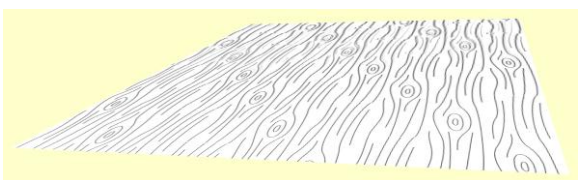

Figure 12: *Tri-linear filtered ink-map. Note how the tone is preserved.*

With carefully crafted ink-maps, a smooth animation can be achieved. Frame-coherence is maintained by using tri-linear filtering which blends different ink-map levels into each other. First acceptance tests indicated that the gradual appearance of detail as the viewer approaches an object is a valuable feature of our illustration style.

3.3. Coloring

The non-photorealistic image elements presented so far (outlining and hatch-maps/ink-maps) did not make use of color. This opens up the possibility to use color as free presentation variable. An example is to lighten the color from the ground upwards to indicate rising uncertainty, like the middle column in Figure 13.

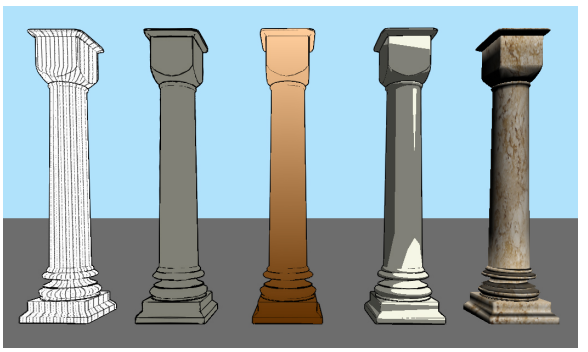

Figure 13: *Different rendering styles can be mixed in a scene.*

What is missing in the plain pen-and-ink style renderings is some sort of shading, which is needed to convey a stronger sense of form. Assigning textures with different degrees of darkness[9] is an approach that we discarded because of image quality considerations. We rather employed the conventional lighting techniques provided by the game engine to add shading by blending with the texture (Figure 14).

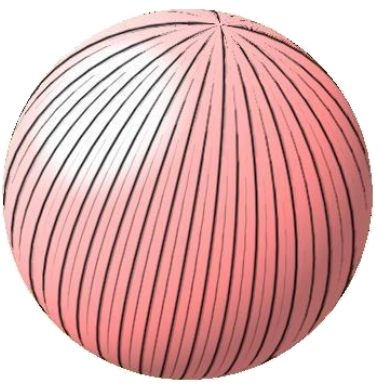

Figure 14: *Shading in combination with hatching conveys a strong sense of form.*

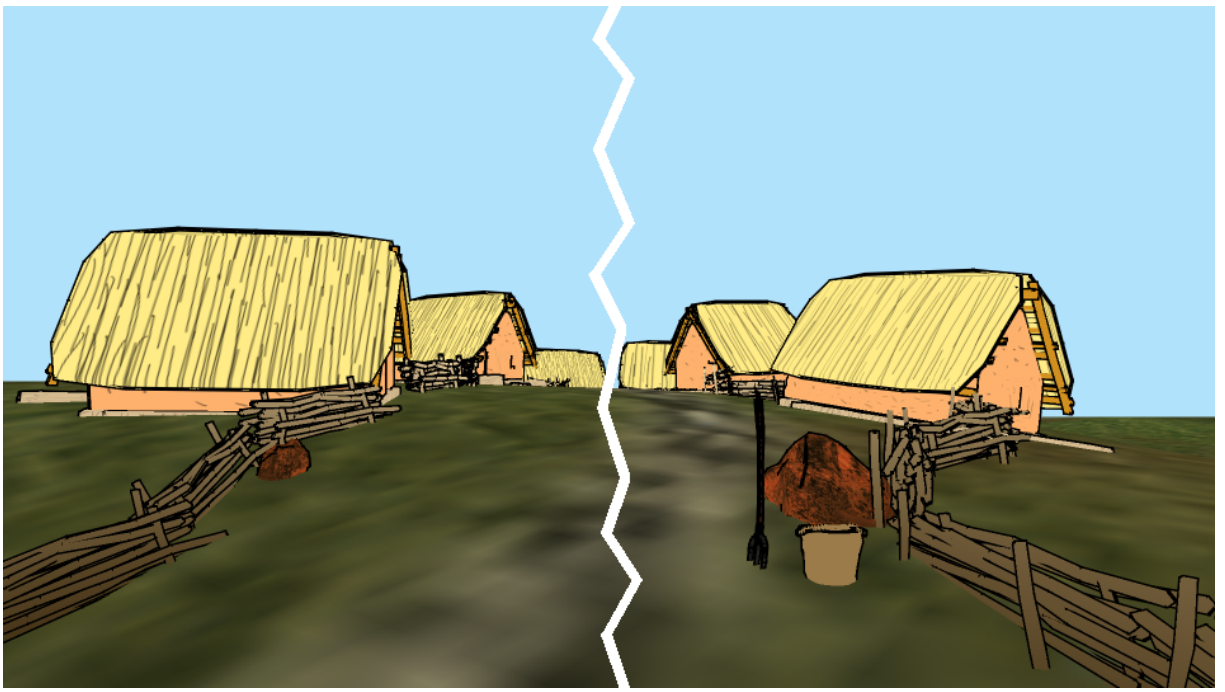

Figure 15: *Conventional mip-mapping (left part) and ink-maps (right). The advantage of our method can be seen at the roofs.*

4. Application and results

The game engine for illustrative walk-throughs is part of a research project to visualize the virtual reconstruction of a medieval palace. In contrast to the VRND project, which visualizes the still existing cathedral of Notre-Dame using the UNREAL-engine[16], our medieval palace no longer exists. This poses the serious problem that viewers cannot distinguish between what is scientifically proven and what has simply been made up to fill in gaps in the image[17]. Therefore, our visualization encodes the level of certainty by the style of presentation: A photorealistic style indicates actual excavated artifacts, an illustrative style points out uncertainty about the reconstruction. Figures 1 and 15 show frames from exemplary visualizations of the medieval palace and its surroundings.

The 3D engine runs with a 750 MHz CPU and a GeForce2 Ultra graphics card. Even with unoptimized models (10,000–20,000 polygons) we always run at interactive frame-rates, i.e. at least 25 fps. The FLY3D engine offers a flexible plug-in API, so all extensions were implemented in C++, like the engine itself.

5. Conclusions

In this paper we have presented a method to extend and reuse a game engine for visualization and presentation purposes. In particular, we show how to extend a game engine to produce non-photorealistic renditions.

The main problem encountered when programming for NPR is that graphical elements are often drawn in screen space rather than being simple projections of artifacts in object space. Indeed, graphics hardware is tuned to calculate such projections from object space; it is initially unclear how to handle, at the same time, adding objects in screen space making use of the same hardware operations. We solve this problem for the case of textures while showing how to maintain frame-to-frame coherence necessary for animation, concentrating on techniques for pen-and-ink-like renditions using hatch-maps and ink-maps. Hybrid renderings combine photorealistic and non-photorealistic picture elements into new forms of visualization.

Our illustration engine is still work in progress and opens up a number of areas for future work: A most promising extension would be to represent the actual shading of an object through hatching as it is carried out in artistic pen-and-ink illustrations. This would vastly enhance the aesthetic quality of the visual appearance. The preservation of the appearance of objects very far away from the viewpoint could be improved, as in these situations the lines tend to get darker than desired. Here, advanced level-of-detail schemes should be employed. Finally, we would like to implement more non-photorealistic styles and enhance the ability of the engine to change and blend these styles interactively.

Acknowledgments

The authors wish to thank Niklas Röber for his work on the FLY3D engine and Thomas Fuchs for modeling. Many thanks to Nick Halper for proof reading the manuscript. We also wish to gratefully acknowledge the excellent cooperation with the Magdeburg Museum of Cultural History, in particular Sebastian Kreiker, who spent many hours with the authors discussing the impact of uncertainty and modeling decisions in the visualization of ancient architecture.

References

1. L. Flickinger. Pencil Whipped, February 2001. `http://www.chiselhead.com/`.
2. A. Mohr and M. Gleicher. Non-invasive, interactive, stylized rendering. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*, 2001. to appear.
3. M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin. Interactive Pen-and-Ink-Illustration. In *Proceedings of SIGGRAPH 94*, pages 101–108, 1994.
4. G. Winkenbach and D. H. Salesin. Computer-generated pen-and-ink illustration. *Proceedings of SIGGRAPH 94*, pages 91–100, 1994.
5. B. J. Meier. Painterly rendering for animation. *Proceedings of SIGGRAPH 96*, pages 477–484, 1996.
6. M. Masuch, L. Schumann, and S. Schlechtweg. Rendering Frame-to-Frame-Coherent Linedrawings for Illustrative Purposes. In *Simulation und Visualisierung 98*, pages 101–112, 1997.
7. C. J. Curtis. Loose and Sketchy Animation. In *SIGGRAPH 98 Visual Proceedings*, page 317, 1998.
8. L. Markosian, M. A. Kowalski, S. J. Trychin, L. D. Bourdev, D. Goldstein, and J. F. Hughes. Real-time nonphotorealistic rendering. *Proceedings of SIGGRAPH 97*, pages 415–420, 1997.
9. A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3d animation. *Proceedings NPAR 2000: First International Symposium on Non Photorealistic Animation and Rendering*, pages 13–20, 2000.
10. R. Pausch, J. Snoddy, E. Hazeltine, R. Taylor, and S. Watson. Disney's Aladdin: First steps toward storytelling in virtual reality. *Proceedings of SIGGRAPH 96*, pages 193–204, 1996.
11. A. W. Klein, W. Li, M. Kazhdan, W. T. Corrêa, A. Finkelstein, and T. A. Funkhouser. Non-Photorealistic Virtual Environments. In *Proceedings of SIGGRAPH 2000*, pages 477–484, 2000.
12. T. Strothotte et al. *Computational Visualization. Graphics, Abstraction and Interactivity*. Springer Verlag, 1998.
13. A. Watt and F. Policarpo. *3D Games: Real-time Rendering and Software Technology*. Addison Wesley, 2001.
14. J. W. Buchanan and M. C. Sousa. The edge buffer: A data structure for easy silhouette rendering. *NPAR 2000: First International Symposium on Non Photorealistic Animation and Rendering*, pages 39–42, 2000.
15. L. Williams. Pyramidal parametrics. *Proceedings of SIGGRAPH 83*, pages 1–11, 1983.
16. VRND: A Real-Time Virtual Reconstruction of the Notre Dame Cathedral. `http://www.vrndproject.com/`, February 2001.
17. M. Masuch, B. Freudenberg, B. Ludowici, S. Kreiker, and T. Strothotte. Virtual reconstruction of medieval architecture. In *Proceedings of EUROGRAPHICS Short Paper*, pages 87–90, 1999.