

# Real-Time Painterly Rendering for MR Applications

Michael Haller \*

Upper Austria University of Applied Sciences  
Media Technology and Design, Austria

Daniel Sperl†

Upper Austria University of Applied Sciences  
Media Technology and Design, Austria

## Abstract

In this paper we describe a real-time system for AR/MR rendering applications in a painterly style. Impressionistic images are created using a large number of brush strokes, which are organized as 3d particles to achieve frame-to-frame coherence. Reference pictures are used to compute the properties of each stroke.

The presented technique is based on B. J. Meier's "Painterly Rendering for Animation". We modified the algorithm of Meier for real-time AR/MR environments by extensively using modern 3d hardware. Vertex and pixel shaders allow both the rendering of thousands of brush strokes per frame and the correct application of their properties. Direct rendering to textures allows rapid generation of reference pictures and assures great flexibility, since arbitrary rendering systems can be combined (e. g. painterly rendering of toon shaded objects, etc.).

**CR Categories:** I.3.5 [Non Photorealistic Rendering]: Hardware Accelerated Rendering—Painterly Rendering; H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities;

**Keywords:** Non-Photorealistic Rendering, Painterly Rendering, Mixed Reality, Augmented Reality

## 1 Introduction

The main focus in computer graphics and consequently in MR/AR applications has always been the rendering of photorealistic imagery. While a tremendous amount of research has been done in this area, non-photorealistic rendering (NPR) is a relatively young field of research. In many areas, a NPR approach can be more efficient than photorealism, since it can focus on the information to be conveyed. Cartoon and painterly rendering have become very important not only in game design [Ken-ichi Anjyo and Katsuaki Hiramitsu 2003], but also in architecture [Freudenberg et al. 2001] and even in medical applications [Feng Dong et al. 2003].

The user is not amazed if an object has a different (non realistic) behavior - the artificial world has to be at least believable and it must be more convincing. In addition, users expect a realistic behavior if the world is rendered photo-realistic. When talking of NPR applications, we think of more artistic environments. Non-photorealistic pictures can be more effective at conveying informa-

tion, more expressive or more beautiful. Especially in AR, where the superimposed objects are not part of the real world, the virtual objects should be more convincing than realistic.

Durand describes in [Durand 2002] that the border between photorealism and non-photorealism can be fuzzy and the notion of realism itself is very complex. Another interesting conclusion of his paper is that the virtual world has to be interpreted more convincing rather than realistic. In other words: it is enough if the virtual, augmented, not existing world, *superficially* looks real. It should be a *believable* world. The augmented objects should be expressive, clear, and look aesthetically perfect.



(a)



(b)

Figure 1: The traditional visitor has no interaction with the painting. In contrast, the second figure shows an AR scenario of Van Gogh's bedroom.

What's the main goal of an Augmented Reality application?

While superimposing virtual objects onto the real picture, the application wants to get the users' attention. Especially when we experiment with AR in everyday settings, the usability part of an AR applications becomes very essential. How can we focus the attention of the user to the superimposed objects? How can we underline the augmented objects?

In paintings, a scene represents an artist's view of the world. All the information he/she wants to convey with the painting has to be assembled by strokes of a brush. The attributes of each stroke can affect various characteristics of the painting. The size of a stroke determines the maximal detail of an object, direction and color describe the quality of the surface. By choosing different brush strokes, a painter can emphasize the parts of the scene he/she considers most important and create a certain mood and/or atmosphere.

Many algorithms have been developed in the last decade that create images which resemble art made by humans [Gooch et al. 2002; Hertzmann and Perlin 2000]. Different art techniques and styles can be simulated, such as pen and ink [Winkenbach and Salesin 1994; Salisbury et al. 1994], hatching [Praun et al. 2001], water color [Curtis et al. 1997] and impressionism [Haeberli 1990; Meier 1996]. The fast progress of graphics hardware affordable for consumers is allowing more and more of these algorithms to be processed in real-time (e. g. [Kaplan et al. 2000; Lander 2000; Markosian et al. 1997; Majumder and Gopi 2002]).

The system described in this paper allows the real-time creation of impressionistic imagery in an AR environment using ARToolKit [Kato et al. 1999]. One of our goals was to allow the real-time rendering of a 3d scene in a painterly style. Imagine looking at the Van Gogh's bedroom painting including full interaction possibilities. A painting where the user can choose the personal view and where he/she can interact in real-time with the different 3d objects. Figures 1 and 2 depict a future scenario of a possible museum's application, where the visitors don't have a flat 2d impression of the painting. In contrast, they get the possibility to *immerse into* the painting. Moreover, in our application we support different NPR rendering techniques, such as toon-rendering and painterly rendering.

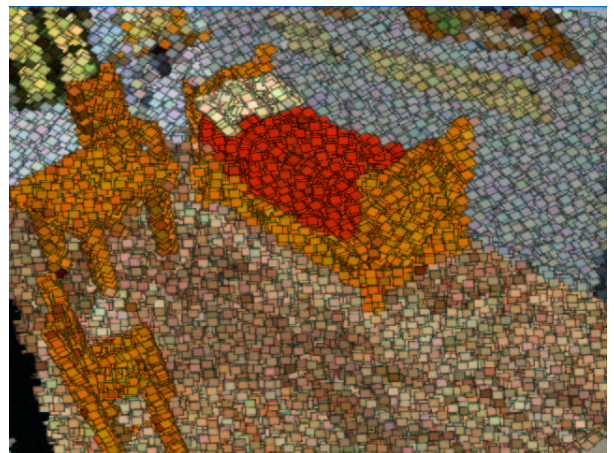
In our application, each image is composed of thousands of brush strokes with different attributes such as color, size, and orientation. The system was designed with inter-frame coherence in mind, since this is a basic requirement of animation. Extensive use of modern graphics hardware—in combination with several optimization steps—allows a rendering speed suitable for real-time applications.

The main contributions of this paper are the following:

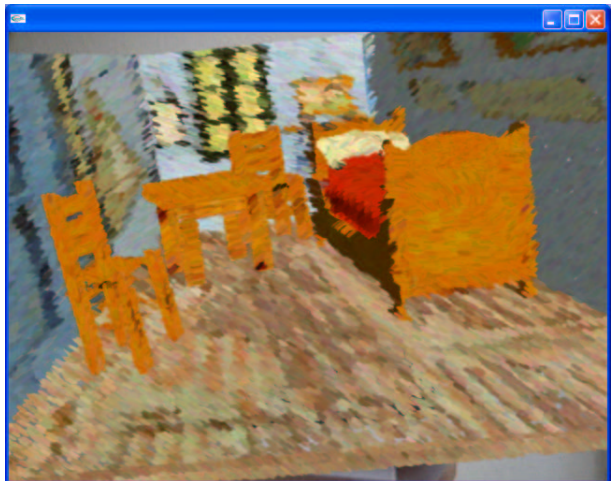
- **Flexible creation of reference pictures:** We created a generic framework in which different renderers are available and can be chosen at run-time. The flexibility of the framework allows a fast exchange of the renderer. Currently, we support standard-, cartoon- and painterly rendering algorithms. Those and any other arbitrary renderers can be used for the reference pictures. If the algorithm renders shadows or reflections, they will appear in the impressionistic image as well.
- **Brush strokes are computed on graphics hardware:** The attributes of each brush stroke are applied directly by the programmable GPU (including reference picture look-up). This reduces the impact on the CPU and allows thousands of strokes to be rendered in each frame.
- **Manual depth test eliminates need for depth-sorting the particles:** The depth buffer of the reference picture is used to determine which particles are visible and need to be drawn. This allows the strokes to be blended without a need for depth-sorting.



(a)



(b)



(c)

Figure 2: Close-up of Van Gogh's bedroom model rendered with the painterly rendering algorithm using different strokes.

The vertex and pixel shaders are programmed using the high level shading language *Cg* [Mark et al. 2003] [Fernando and Kilgard 2003]. Basically, a vertex shader (also known as vertex program) is a program that is executed for each vertex of an object, and can perform advanced lighting calculations or vertex transformations (e.g. the movement of waves in water). A pixel shader (also known as fragment program) can be used to determine the color of a pixel, since it is executed for each pixel of a polygon's surface. A common application for a fragment program is per-pixel-lighting (e.g. *dot3 bump mapping*).

The results of the presented system are very promising. Although not all aspects of the original (non real-time) painterly algorithm were included, the rendered images provide an idea of what is possible using this technique. With a few more optimization steps it should be possible to utilize the presented technique in more complex environments.

In the following section, we describe current painterly algorithms and discuss related work. Our modified painterly rendering algorithm is described in Section 3. In Section 4 we show the results including images rendered in real-time. Finally, in Section 5 we conclude with a summary of the presented techniques and future work.

## 2 Related work

The presented system was inspired mostly by [Haeberli 1990] and [Meier 1996]. Haeberli described a system for creating painterly images by using brush strokes which obtain their attributes (position, color, size, etc.) from reference pictures containing photographic or rendered images. The use of reference pictures simplifies the creation of artistic images and allows fast and direct control of the process.

The brush strokes of Haeberli's system are placed in screen space, which is sufficient for still images but leads to a "shower door" effect in animated sequences. In [Meier 1996], this problem is addressed by using particles to control the placement of brush strokes. The particles are rendered using 2D textures which are first sorted by the distance to the user and then blended with each other.

In photorealistic rendering, particle systems can be used to render complex natural phenomena like water or fire using simple geometric structures. In [Reeves and Blau 1985], particle systems are used to create complex objects like trees. [Kaplan et al. 2000] utilizes particle systems in a similar way to Meier, but uses so-called *geograftals* instead of bitmaps to create brush strokes.

Meier's system was already converted into a real-time system by Drone et al. in [Drone et al. 2000]. This system is designed to run even on older hardware without using vertex or pixel shaders. Since most of the operations and the calculations are executed on the CPU, the number of strokes is limited. In addition, the strokes are depth sorted in each frame. Although this results in an excellent image quality, it has additional impact on the rendering speed.

Our system extends [Meier 1996] and introduces several optimizations to achieve a performance suitable for real-time applications. The amount of particles is dynamically changed to achieve an optimal trade-off between screen coverage and rendering speed. In addition, a manual depth test carried out by the graphics hardware allows the brush strokes to be blended on the screen without depth-sorting the particles. The transfer of a great deal of calculations from the CPU to the GPU results in a higher speed than is accomplished in [Drone et al. 2000], while our user-defined creation of reference pictures allows more flexibility.

## 3 The real-time painterly rendering algorithm

Before we start in this section with the real-time painterly rendering algorithm, we present Meier's algorithm as shown as follows:

```

create particles to represent geometry
for each frame of animation
    create reference pictures using geometry, surface
    attributes, and lighting
    transform particles based on animation parameters
    sort particles by distance from viewpoint
    for each particle, starting with furthest from camera
        transform particle to screen space
        determine brush stroke attributes from reference
        pictures or particles and randomly
        perturb them based on user-selected parameters
        composite brush stroke into paint buffer
    end (for each particle)
end (for each frame)

```

Our modified painterly algorithm can be divided into three steps:

1. Before we can start to render the scene, we need a **preprocessing** step. The polygon meshes are converted into 3d particles. The attributes of brush strokes that remain constant in each frame can be computed simultaneously.
2. Next, the rendering can start and it takes at least two passes. During the **first pass**, two reference images that contain color and depth information are rendered and stored into textures. An arbitrary rendering algorithm can be used.
3. In the **second pass**, an optimal amount of brush strokes is rendered using billboards. Reference pictures from the first pass are used to define color and visibility, while the surface normal determines the orientation of each stroke. This is the most expensive part of the rendering process, since a large number of strokes needs to be processed.

The graphics-pipeline of the described system is shown in figure 3. It is an extended version of the pipeline presented in [Meier 1996]. The main differences are the optimization steps, namely the visibility and depth tests, as well as the calculation of the optimal number of particles.

In the following sections, we begin with a discussion of the preprocessing step, followed by a description of the two rendering passes.

### 3.1 Preprocessing

In the preprocessing step, the polygonal objects of the 3d scene are used as a basis for creating particles. These particles will be used later to position the brush strokes that compose the image.

All particles are placed within the polygons of the mesh. Depending on the area of the polygon and a user-defined value for density, a certain number of particles is placed within each polygon. In the described approach, we implemented a simple algorithm for distributing the particles randomly onto the surface (cf. figure 4).

The number of particles represents the maximum that can be rendered in the second pass. However, only a fraction of them will be needed in an average frame. The actual number depends on the distance of the camera to each surface and the size of the brush strokes. As we will see later, we can reduce the number of needed particles immensely by blending the billboards with the original scene. This means that we can significantly accelerate the preprocessing step by carefully determining the density of the brush strokes. If the density is too high, a tremendous amount of particles is created, which has negative effects on performance and memory usage.



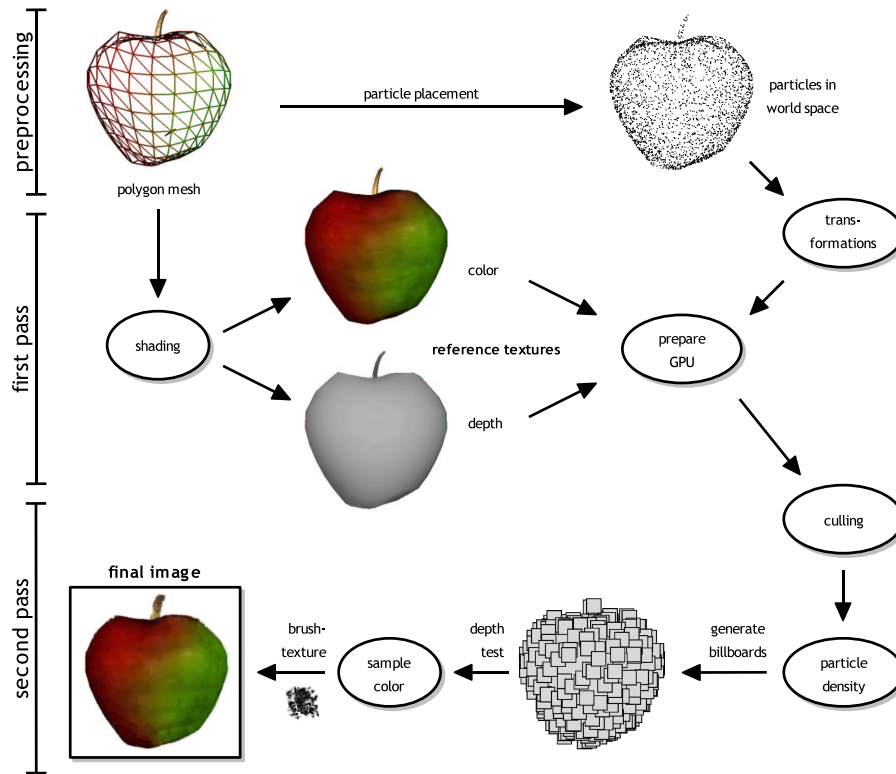


Figure 3: The pipeline of the painterly renderer.



Figure 4: The density of the particles is a key issue for the speed-up.

### 3.2 First pass

The polygon-meshes of the objects are not only needed for particle placement, but also for the rendering of reference pictures. Two images are created in the described system: one providing information on color, the other about depth (examples can be seen in figure 3). The images have to be created as efficiently as possible and should be stored within textures to allow fast access during the second pass. As described in [Kirk 2003], the use of textures allows a very efficient solution for real-time rendering.

There are two possibilities to create these textures. One would be to render the geometry into the frame- and depth-buffer (using common OpenGL-functions) and copy the content of these buffers into textures. However, copying data is not very efficient. Therefore, we used OpenGL extension functions that allow direct rendering to textures (including `ARB_render_texture` and `WGL_ARB_pbuffer`). These extensions allow the creation of additional rendering con-

texts, and proved to be useful.

It would also be possible to determine the attributes of the brush strokes directly during the rendering process (e. g. in the vertex program). The advantage of using reference pictures, however, is the high flexibility. In theory, any arbitrary shading technique could be used in the first rendering pass. It does not matter whether normal Gouraud shading or a Cartoon shading approach is used. Even shadows and reflections could be used and would consequently be transferred directly to the brush strokes.

### 3.3 Second pass

The second rendering pass is the most expensive one. Thousands of brush strokes have to be rendered simultaneously, which can be accomplished by programming the graphics card directly (using the vertex and fragment shaders), and by several other optimizations.

The final image is composed of a large amount of billboards which represent the different strokes. The particles are grouped within the polygons on which they were created. This turns out to be helpful in several steps of this rendering pass.

#### 3.3.1 Backface culling and clipping

Before the particles are rendered, typical optimization steps, such as backface culling and clipping, should be performed. This can save a lot of processing time. Normally, we could use OpenGL's `glCullFace()` function to achieve backface culling, but due to the fact that only billboards are used, it would not have any effect in this case. As described before, each billboard is attached to a (invisible) polygon, and we know the position and orientation of this polygon. Consequently, we can determine if a billboard has to be rendered or not.

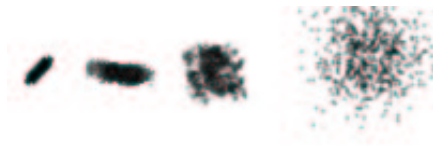


Figure 5: Different brush strokes used to render the paintings in this paper.

Another speed increase could be achieved by clipping the billboards outside the view frustum, but this has not been implemented in the described prototype.

If all optimizations are executed with the polygons of the original geometry, and only the particles of the polygons assumed to be visible are drawn, the processing time can be reduced to a fraction of the time without performance tuning.

### 3.3.2 Variation of particle density

In a real painting, small objects are drawn with few brush strokes. The maximum level of detail is predetermined by the size of the brush. The painterly renderer should resemble this behavior.

When using polygon meshes, automatic generation and modification of the levels of detail can become a complex task. In contrast, when using the painterly renderer this task becomes comparatively easy. If we want to decrease the detail, we have to reduce the number of rendered strokes.

The required number of strokes per polygon is dependent on three things:

- the size of the brush stroke,
- the area of the polygon in screen space, and
- the desired stroke-density.

If these values are known, the optimal number of strokes per polygon can be calculated and used when the strokes are drawn.

While the size of each brush stroke and the desired density are defined through the user's input, the polygon's area in the output image has to be approximated. This can be accomplished by using the actual size of the polygon (in world space), its distance to the viewer, and the angle between its normal-vector and the view-vector.

The density is calculated in each frame and for each polygon. Therefore, when the camera moves through the scene, the number of particles smoothly changes to the appropriate amount.

### 3.3.3 Creation of billboards

Now, the vertex shader comes into play. Each brush stroke (cf. figure 5) is represented by a textured billboard. The coordinates of the billboard's vertices could be created on the CPU and sent to the graphics card, but it is faster to create the billboards directly on the GPU. In order to accomplish this, the coordinate of the center of the billboard (= the coordinate of the particle) is sent to the vertex program four times, accompanied by the correct texture coordinates needed for the brush texture. These two coordinates can be used—in combination—to create each corner of the billboard. Since vertex programs need the coordinate of the vertex in clip-space, the two sets of coordinates have to be added together (in addition to adding an offset and multiplying with the desired size).

Each billboard should always have the same size in screen space, independent from the distance it is being viewed from. Using a perspective view, an object that is further away from the viewer

is normally displayed smaller than a closer one. Internally, this is accomplished by a perspective division. In this process, the  $x$ - and  $y$ -coordinates of a vertex (in clip space) are divided by their distance from the viewer. To compensate for this process, which cannot be avoided presently, the affected coordinates are simply multiplied with the same value in the vertex program, thus reversing the division.

However, the process described above can lead to holes in the surface of the objects if the camera is very close to a surface, because there may not have been enough particles created during the preprocessing step. We compensated for this effect by re-enabling the perspective division at exactly the distance when there are no longer enough particles available anymore (cf. figure 6).

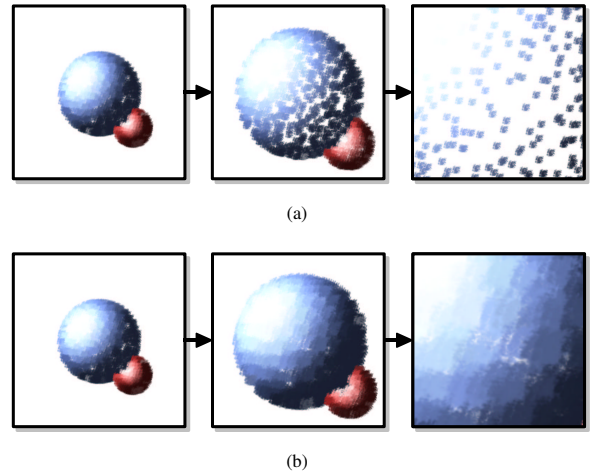


Figure 6: When there are too few particles available for a certain perspective (a), the strokes are scaled proportionally (b).

### 3.3.4 Accessing reference textures

The color of each brush stroke is computed by accessing the color reference picture. Figure 7 illustrates the process. For this purpose, the screen space position of the particle is needed that describes the coordinates ( $x/y$ ) which are used to access the reference pictures. The screen coordinates are calculated in the vertex program and subsequently scaled and biased to be in the range that is needed for texture access. The color of the reference image at that position is then used as color for the billboard (and consequently for the brush stroke, after multiplying with the brush texture).

If a slight color variation was stored when creating the particles, it can also be added. This results in images that are more natural, since the colors of a real painting are rarely mixed perfectly.

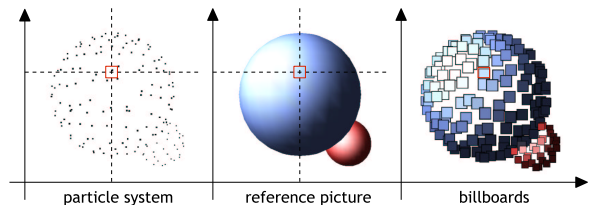


Figure 7: A reference texture is used to compute the color of each brush stroke.

### 3.3.5 Depth test

Brush strokes generally are images with an alpha channel. Therefore, the particles have to be rendered into the framebuffer using a blending function. The problem is that in order to get a correct image, blending needs the brush strokes to be drawn depth sorted, beginning with the one that is furthest away from the viewer. Since several thousand strokes have to be drawn, it is not advisable to do that in real-time. Therefore, we had to find an alternative approach.

The solution is to draw only those brush strokes situated on surfaces that are visible in the present frame (even though the surfaces themselves are not rendered). Obscured strokes should not be drawn at all. The visible particles can be determined by using the depth reference image stored in the first rendering pass (the content of which is the depth-buffer of the first pass). The depth value found in the reference texture is compared with the depth value of the present billboard. If the billboard's depth is smaller or equal to the one found in the reference texture, it is drawn (as shown in figure 8). Due to the fact that access to this texture is only granted in the fragment program, this test has to be executed for every fragment.

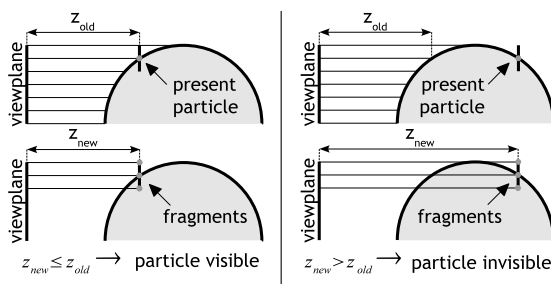


Figure 8: The depth test compares the  $z$ -values from the first pass ( $z_{old}$ ) with the  $z$ -values of each billboard ( $z_{new}$ ).

The results of this depth test are not as accurate as those of depth sorting, but if density and size of the strokes are well balanced, this can hardly be noticed in the final image.

## 4 Results and Discussions

Figure 10 shows a still life that was rendered in real-time. In this case the tests have been based in a non AR environment, because the performance tests shouldn't be influenced by the ARToolKit library. Approximately 50,000 brush strokes compose this image, while in the preprocessing step, nearly 500,000 particles have been created. This means that the camera can be placed very closely to the objects before the brush strokes start to scale.

Table 1 shows three different scenes rendered in real-time. All scenes are rendered on an ordinary PC with nVIDIA's GeForce4 graphics card. Naturally, the framerate decreases with the number of particles in the scene. The tests included three different scenarios. We show three different quality levels with a minimum, a medium and a maximum quality - depending on the amount of resulting billboards.

- The first scenario shows the Venus model with 5,500 polygons and 825,912 possible particles.
- The second and third scenarios show a still life model with 4,400 polygons polygons and 547,860 possible particles.

Of course, the best performance results are achieved by using fewer billboards (approximately 10,000 billboards are no problem

at all). Notice that by using a Pentium 4 with 2.4 GHz and a GeForce4 Ti 4600, a scene with about 58,000 particles results in an interactive rate of 36.6 frames per second, while even very complex scenes with about 100,000 particles were still rendered with more than 8 frames per second.

The number of required particles can be reduced tremendously if the reference color image (rendered in the first pass) is drawn as background image behind the brush strokes. In this case, the particle density can be lower than it normally is and the background shines through without being noticed.

Figure 10 and 11 depict the still life and other models using different brush strokes.

One problem of our algorithm can be recognized at the edges of an object and/or when an object is animated. In this case, some particles are popping on and of as they become visible and invisible. A possible solution that the particles shouldn't be created and eliminated quickly but they could be faded in/out more smoothly.

Besides the performance tests, our painterly algorithm got good feedback from the users, because they liked the idea of combining a painterly renderer with AR. NPR makes sense in AR, because the overlapped objects are indirectly emphasized. Of course, sometimes the non-realistic approach and the non perfect rendering style shouldn't influence the overall picture of the scene, but in general, NPR makes good pictures in an AR environment.

## 5 Future Work and Conclusion

We have presented a technique for rendering real-time AR images in a painterly style (cf. figure 9).

Through extensive use of modern graphics hardware, the process is fast enough to be applicable in real-time environments. The system provides automatic variation of particle density and eliminates the need of depth-sorting the particles by accessing the depth buffer of the first rendering pass. By allowing arbitrary rendering algorithms that are used for the creation of a reference image, a variety of different painterly styles can be achieved.

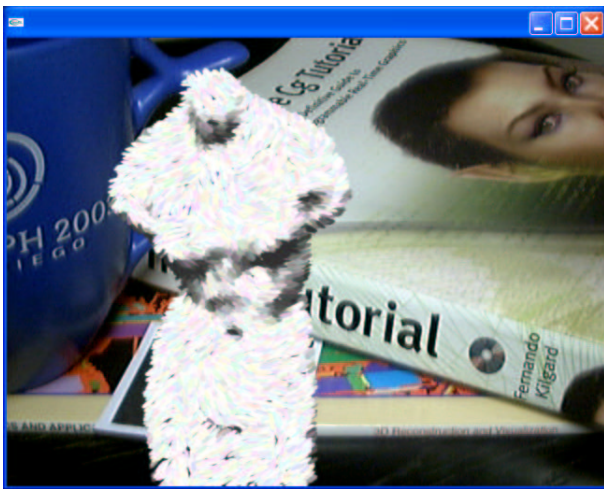
A problem can be observed when the brush strokes are relatively large. Since the brush strokes are not depth-sorted, it can happen that small details of an object are obscured by adjacent strokes. This can be avoided, however, if the brush size and particle density are chosen carefully. For a maximum of artistic freedom, the user should be allowed to choose some basic brush attributes per object or even per polygon.

The biggest challenge is to reduce the amount of memory used by the algorithm. If the painterly renderer, as described above, is used for very complex scenes (e.g. the extensive environments of modern computer games), the number of particles that are created in the preprocessing step is far too high. A solution for this problem would be to create only those particles during preprocessing which are needed for the first frame of animation. All other particles could be created in real-time during the rendering (while those no longer needed could be deleted simultaneously). If it is assured that the camera only moves in small steps, i.e. direct jumps from one point to another are not possible, the number of new particles per frame would be small enough to allow on-the-fly creation.

Non-photorealistic rendering techniques (as depicted in figure 9) can be used for achieving artistic effects. Of course, it depends on the type of the applications and each AR application has its domain focus - consequently, we have different end users and different goals that require different rendering techniques. Finally, we have seen that in AR, the user should immediately be confronted with the abstract model - in fact, with the superimposed, augmented object. Therefore, a painterly rendered object could be a good solution.



(a)



(b)

Figure 9: A sculpture in an AR environment using a cartoon render and the painterly render.

## References

- CURTIS, C. J., ANDERSON, S. E., SEIMS, J. E., FLEISCHER, K. W., AND SALESIN, D. H. 1997. Computer-generated watercolor. In *Proceedings of SIGGRAPH 97*, 421–430.
- DRONE, S., KRISHNAMACHARI, P., LAPAN, D., MICHAELS, J., SCHMIDT, D., AND SMITH, P. 2000. Project aember: Real-time painterly rendering of 3d scenery. Tech. rep., UIUC ACM SIGGRAPH, August.
- DURAND, F. 2002. An invitation to discuss computer depiction. In *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, ACM Press, 111–124.
- FENG DONG, GORDON J. CLAPWORTHY, HAI LIN, AND MELEAGROS A. KROKOS. 2003. Nonphotorealistic Rendering of Medical Volume Data. *IEEE Computer Graphics and Applications* 23, 4 (July/August), 44–52.
- FERNANDO, R., AND KILGARD, M. J. 2003. *Cg Tutorial, The: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley.
- FREUDENBERG, B., MASUCH, M., ROEBER, N., AND STROTHOTTE, T. 2001. The Computer-Visualistik-Raum: Veritable and Inexpensive Presentation of a Virtual Reconstruction. In *In Proceedings VAST 2001: Virtual Reality, Archaeology, and Cultural Heritage*, 97–102.
- GOOCH, B., COOMBE, G., AND SHIRLEY, P. 2002. Artistic vision: painterly rendering using computer vision techniques. In *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, ACM Press, 83–ff.
- HAEBERLI, P. E. 1990. Paint by numbers: Abstract image representations. In *Proceedings of SIGGRAPH 90*, 207–214.
- HERTZMANN, A., AND PERLIN, K. 2000. Painterly rendering for video and interaction. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, ACM Press, 7–12.
- KAPLAN, M., GOOCH, B., AND COHEN, E. 2000. Interactive artistic rendering. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, ACM Press, 67–74.
- KATO, H., BILLINGHURST, M., BLANDING, B., AND MAY, R. 1999. ARTToolkit.
- KEN-ICHI ANJYO, AND KATSUAKI HIRAMITSU. 2003. Stylized Highlights for Cartoon Rendering and Animation. *IEEE Computer Graphics and Applications* 23, 4 (July/August), 54–61.
- KIRK, D. 2003. *Cg Toolkit - User's Manual, Release 1.1*, 1 ed. nVIDIA Corporation, March.
- LANDER, J. 2000. Shades of disney: Opaquing a 3d world. *Game Developers Magazine* (March).
- MAJUMDER, A., AND GOPI, M. 2002. Hardware accelerated real time charcoal rendering. In *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, ACM Press, 59–66.
- MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a c-like language. In *Proceedings of SIGGRAPH 2003*, 896–907.
- 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*, 415–420.
- MEIER, B. J. 1996. Painterly rendering for animation. In *Proceedings of SIGGRAPH 96*, 477–484.
- PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. 2001. Real-time hatching. In *Proceedings of SIGGRAPH 2001*, 581.
- REEVES, W. T., AND BLAU, R. 1985. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH 85 Proceedings*.
- SALISBURY, M. P., ANDERSON, S. E., BARZEL, R., AND SALESIN, D. H. 1994. Interactive pen-and-ink illustration. In *Proceedings of SIGGRAPH 94*, 101–108.
- WINKENBACH, G., AND SALESIN, D. H. 1994. Computer-generated pen-and-ink illustration. In *Proceedings of SIGGRAPH 94*, 91–100.





(a) The still life with the different reference images: the upper left figure shows the brush texture, the bottom left figure shows the normal rendered scene, and finally the bottom right figure shows the depth map of the scene.



(b) Stroke brushes without the background image.



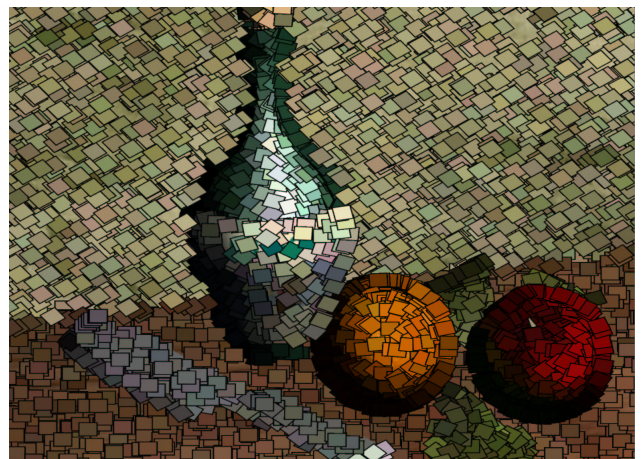
(c) Different brush textures yield in different artistic styles.



(d) Orienting the brush strokes in dependence of the surface normal can lead to a realistic painting effect.






(e) This brush stroke shows in a very impressive way the usage of the particles.



(f) A close up of the scene.

Figure 10: The still life using different brush strokes rendered in real-time.



Scene	A	B	C
			
Particles (Preprocessing)	825 912	547 860	547 860
Particles (Quality: Minimum)	2 610	5 657	24 978
Particles (Quality: Medium)	4 784	19 922	99 799
Particles (Quality: Maximum)	14 563	58 797	220 441

	Scene A			Scene B			Scene C		
	Min.	Med.	Max.	Min.	Med.	Max.	Min.	Med.	Max.
AMD Athlon 550, 448 MB SD-RAM, GeForce4 Ti 4200	47,2	36,1	17,2	36,5	13,3	5,3	11,7	3,2	1,5
Intel P4 1,8 GHz, 786 MB DDR-RAM, GeForce4 Ti 4600	>85,0	55,7	28,5	52,6	25,1	9,7	21,3	6,1	2,8
Intel P4 2,5 GHz, 1 GB DDR-RAM, GeForce4 Ti 4600	>75,0	>75,0	37,6	>75,0	36,6	14,5	25,1	8,5	4,2

Table 1: The performance of the painterly prototype (in fps) on different systems.

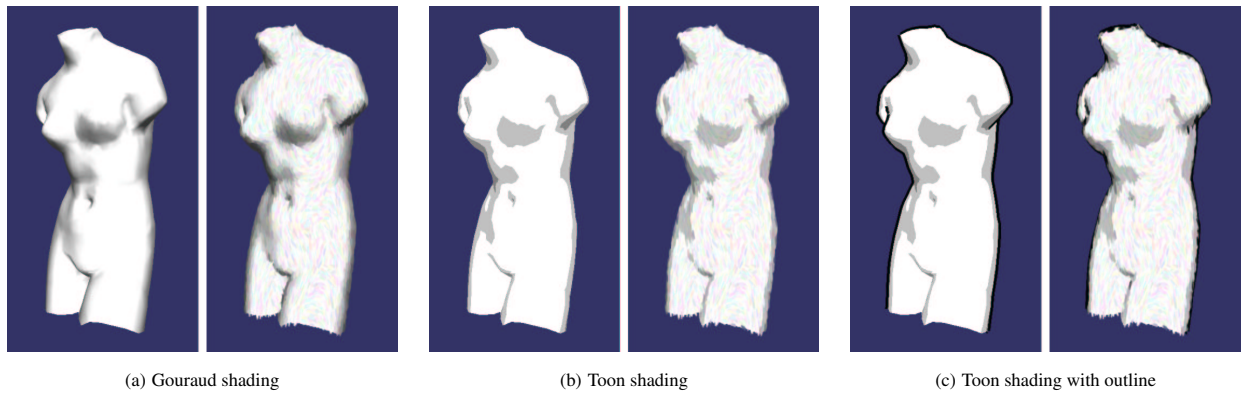


Figure 11: The rendering style of the reference pictures (left) has direct influence on the resulting images (right).