Realtime Painterly Rendering for Animation

Daniel Sperl
Hagenberg College of Information Technology, Austria

Abstract

This paper describes a realtime system for rendering animations in a painterly style. Impressionistic images are created using a large quantity of brush strokes, organized in particle systems to achieve frame-to-frame coherence. Reference pictures are used to compute the properties of each stroke.

The basis of the presented technique is [Meier, B. J.: *Painterly Rendering for Animation.* In: *Proceedings of SIGGRAPH 96*, p. 477–484, 1996], which describes a system for offline-rendering. The system presented there was modified for being applicable in realtime environments by extensive use of modern 3D hardware. The possibility to send instructions to the graphics card and execute them either per vertex or per pixel—using vertex and pixel shaders—allows both the rendering of thousands of brush strokes per frame, and the correct application of their properties. Programming the graphics hardware is done with the high level shading language *Cg*. Fast access to reference pictures is achieved by using *OpenGL* extensions that allow direct rendering to textures.

The results are very promising. Although not all aspects of the original 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 complex environments.

Figure 1: This image was rendered in realtime using a prototype of the described system.

1 Introduction

The main application of computer graphics has always been the rendering of photorealistic imagery. A tremendous amount of research has been done in this area, and the majority of problems has been solved by now. In contrast to this, non-photorealistic rendering (NPR) is a relatively young field of research. Nevertheless, many algorithms have been developed in the last decade that create images, which resemble art made by humans. The fast progress of graphics hardware affordable for consumers allows more and more of these algorithms to be processed in realtime (e. g. [2, 4, 5, 6, 8]).

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

sphere.

The most important elements of this procedure can be emulated in a computer, some of them even in realtime applications. However, some advanced rendering technology has to be used, e. g. vertex and pixel shaders. Even then, several tradeoffs between image quality and rendering speed have to be made.

The system described here uses programmable graphics hardware. This is accomplished using the high level shading language *Cg*. To understand the principles of this language, a basic knowledge of vertex and pixel shaders is required.[1] Basically, a vertex shader (also known as vertex program) is a program that is executed for each vertex of an object, and can compute advanced lighting calculations or surface 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 fragment[2], since it is executed for each pixel of a polygon's surface. A common application for a fragment program is per-pixel-lightning (e. g. *dot3 bump mapping*).

2 Related work

The presented system was inspired mostly by [3] and [7]. In [3], Haeberli describes 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 [7], this problem is addressed by using particle systems 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.

3 The algorithm

To convert the system of Barbara J. Meier [7] to realtime applications, the rendering of the brush strokes has to be accelerated. This is achieved mainly by using Cg, *nVIDIA*'s high-level shading language [1]. Creation of the reference pictures is achieved by using several *OpenGL*-extensions that allow direct rendering to textures [10, 9].

The graphics-pipeline of the described system is shown in figure 2. It is an extended version of the pipeline presented in [7]. A prototype of the described system was implemented using *OpenGL*. It can be downloaded from the following website: `http://www.incognitek.com/redge/painterly`.

The rendering process consists of three steps:

1. Before the actual rendering steps, some preprocessing is needed. The polygon meshes are converted into particle systems. Those attributes of the brush strokes that remain constant in each frame can be computed simultaneously.

2. The rendering itself takes at least of two passes. During the first pass, two reference pictures, which contain color and depth information, are rendered.

3. In the second pass, the brush strokes are rendered. Therefor, each particle is represented by a billboard. This is the most expensive part of the rendering process, since a large number of strokes needs to be processed.

3.1 Preprocessing

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

The 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 implementation, a simple algorithm for distributing the particles randomly on the surface is used.

The number of particles created here represents the maximum that can be rendered later. 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.

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 pictures are created in the described system: one providing information on color, the other about depth (example images can be seen in figure 2). The pictures have

[1] Sufficient information on Cg can be found in [1].

[2] Before a pixel is written into the framebuffer, it is called *fragment*

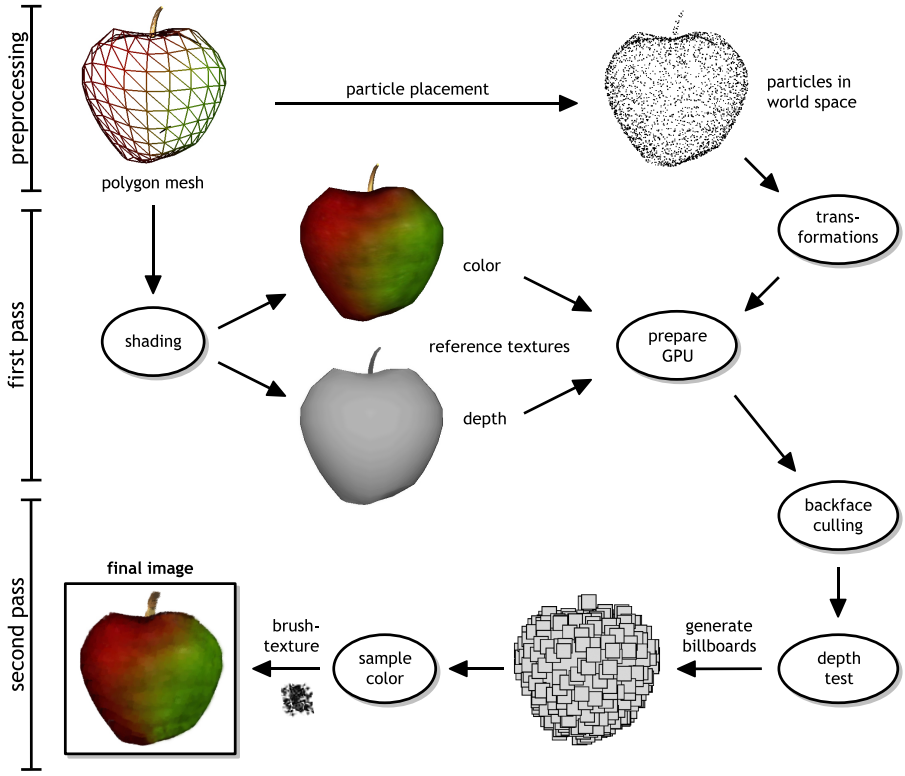


Figure 2: The pipeline of the painterly renderer.

to be created as efficiently as possible and should be stored within textures to allow fast access during the second pass.

There are two possibilities to create these textures. One would be to render the geometry into the frame- and z-buffer (using common OpenGL-functions) and copy the content of these buffers into textures. However, copying the data is not very efficient. For this reason, another approach is used. There are some OpenGL extensions which allow direct rendering to textures (including `ARB_render_texture` and `WGL_ARB_pbuffer`). These extensions allow the creation of additional rendering contexts.[3] In the presented implementation, they proved to be useful.

Unfortunately, some of the used extensions only work on graphics hardware from *nVIDIA*. It should be possible, however, to overcome this restriction.

It would also be possible to determine the attributes of the brush strokes directly during rendering (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 pass. It does not matter whether normal Gouraud shading or complex shadow- and reflection-algorithms are used—all the provided information will be assigned directly to the brush strokes.

[3]This is normally used for effects like realtime reflections, etc.

3.3 Second pass

The second rendering pass is the most expensive one. Thousands of brush strokes are rendered simultaneously, which can be accomplished by programming the graphics card directly, and by several optimizations.

The final image is composed of nothing but the billboards which represent the strokes. Nevertheless, the particles are still grouped within the polygon on which they were created. This turns out to be helpful in several steps of this rendering pass.

3.3.1 Culling

Before the particles are rendered, the typical optimization-steps should be performed, such as backface culling and clipping. This can save a lot of processing time and has be done manually. The reason for having to do this is that some optimizations,

which are normally done by the graphics card, do not work in the painterly renderer. The scene is made up exclusively by billboards, which are—by definition—front facing. So backface-culling, for example, cannot be performed by the graphics card. However, the polygon-meshes from which the particles were created do have backfacing polygons, and these can be used to cull all particles placed within them.

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 otherwise necessary.

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 modification of the level of detail is a complex process. Using the painterly renderer, on the contrary, makes this comparatively easy. To decrease the detail, it is only necessary to reduce the number of rendered strokes.

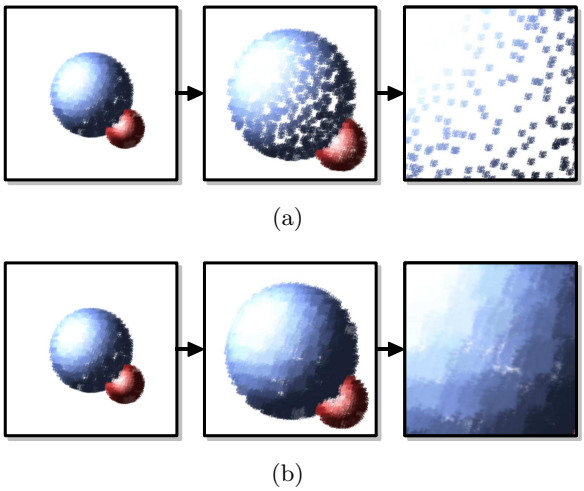
The required number of strokes per polygon is dependent on three things: the size of the brush stroke, the area of the polygon in screen space and the desired stroke-density. If these values are known, the optimal number of strokes per polygon can be calculated and used when the strokes are drawn.

While the size of each brush stroke and the desired density are defined through user input, the polygon's area in the output image has to be approximated. This can be accomplished 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. Therefor, when the camera moves through the scene, the number of particles smoothly changes to the appropriate amount.

3.3.3 Creation of billboards

Each brush stroke 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 since vertex programs are needed anyway, it is faster to create the billboard 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 just have to be added together (in addition to adding an offset and multiplying with the desired size).



(a)

(b)

Figure 3: When there are too few particles available for a certain perspective (a), the strokes are scaled proportionally (b).

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 through 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 created enough particles during the preprocessing step. This can be compensated for by re-enabling the perspective division at exactly the distance when too few particles are available (figure 3).

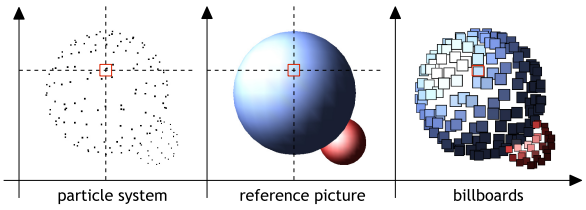


Figure 4: A reference texture is used to compute the color of each brush stroke.

3.3.4 Accessing reference textures

The color of each brush stroke is computed by accessing the color reference picture. For this purpose, the screen space position of the particle is needed, for it describes the coordinates that 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. In the fragment program, these coordinates are used for looking up the according color of the stroke. Figure 4 illustrates the process.

The final color of the billboard is then computed by multiplying the brush texture with the reference color. If a slight color variation was stored when creating the particles, it can also be added. It can be used to create more natural images, since the colors of a real painting are rarely mixed perfectly.

3.3.5 Depth test

Brush textures generally use an alpha channel. Therefor, the particles have to be rendered into the framebuffer using blending. 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 viewpoint. Since several thousand strokes have to be drawn, this is impossible to do in realtime. That is why another approach had to be found.

The solution is to draw only those brush strokes which are situated on polygons that are visible in the present frame (even though the polygons themselves are not rendered). This can be accomplished using the depth reference picture saved in the first rendering pass (the content of which is the z-buffer of the first pass). The depth value found in the reference texture is compared to the depth value of the present billboard, and only if the billboard's depth is smaller or equal to the one found in the reference texture, it is drawn (as shown in figure 5). Due to the fact that access to this texture is only granted in the fragment program, this test has to be repeated for every fragment.

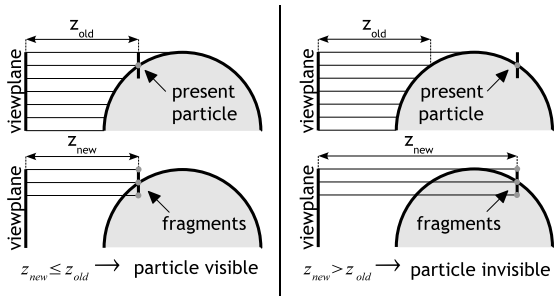


Figure 5: The manual depth test compares the z-values from the first pass (z_{old}) with the z-values of each billboard (z_{new}).

Figure 6: Two sculptures, rendered in realtime.

The results of this manual 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.

3.4 Results

Figure 1 shows a still life which was rendered in realtime. Approximately 50.000 brush strokes assemble this image, while in preprocessing, nearly 500.000 were created. This means that the camera can be placed very near to the objects before the brush strokes start to scale. Figures 6 and 7 display some other painterly rendered objects.

On an average computer, up to about 30.000 particles are rendered very smoothly (with about 30 frames per second). The framerate decreases with the number of particles in the scene. Using a Pentium 4 with 2.4 GHz and a GeForce4 Ti 4600, a

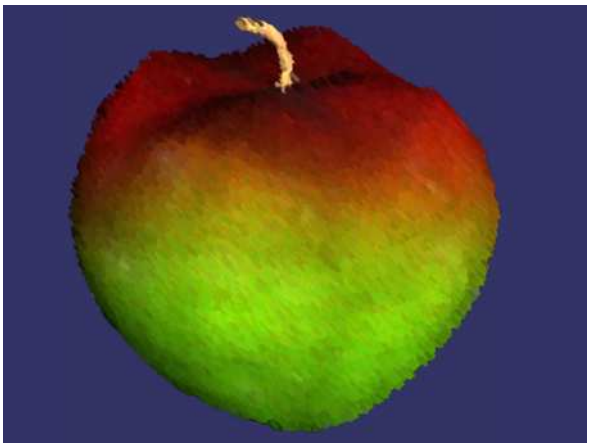


Figure 7: Example for a rendered image showing an apple.

scene with about 100.000 particles was still rendered with more than 8 fps.

The number of required particles can be reduced a lot if the color reference picture (rendered in the first pass) is drawn as a background behind the brush strokes. In that case, the particle density can be lower. The background shines through without being noticed.

However, the algorithm allows several more optimizations. In the prototype, there was, for example, no culling implemented, which would speed up the process immensely, especially when an object of the scene is observed very closely.

Even though the results are very promising, there are still some improvements to be made. First of all, the current prototype only works on nVIDIA graphics cards (starting with a GeForce3, apart from cheaper MX-variants) because of some of the extensions chosen. However, the rendering process should also work on hardware from ATI if some minor changes are made.

Another problem can be observed when the size of the brush strokes is relatively big. 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.

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 huge environments of modern computer games), the number of particles that are created in the preprocessing step is far too large. 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 realtime during the rendering (while those not 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.

References

[1] *Cg Toolkit—User's Manual, Release 1.1.* nVIDIA Corporation, first edition, March 2003.

[2] B. Gooch, P.-P. J. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999.

[3] P. E. Haeberli. Paint by numbers: Abstract image representations. In *Proceedings of SIGGRAPH 90*, pages 207–214, 1990.

[4] M. Kaplan, B. Gooch, and E. Cohen. Interactive artistic rendering. In *Proceedings of the first international symposium on Nonphotorealistic animation and rendering*, pages 67–74. ACM Press, 2000.

[5] J. Lander. Shades of disney: Opaquing a 3d world. *Game Developers Magazine*, March 2000.

[6] L. Markosian, M. A. Kowalski, S. J. Trychin, L. D. Bourdev, D. Goldstein, and J. F. Hughes. Real-time nonphotorealistic rendering. In *Proceedings of SIGGRAPH 97*, pages 415–420, 1997.

[7] B. J. Meier. Painterly rendering for animation. In *Proceedings of SIGGRAPH 96*, pages 477–484, 1996.

[8] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-time hatching. In *Proceedings of SIGGRAPH 2001*, page 581, 2001.

[9] C. Wynn. Using p-buffers for off-screen rendering in opengl. Technical report, nVIDIA Corporation, August 2001.

[10] C. Wynn. Opengl render-to-texture. Technical report, nVIDIA Corporation, March 2002.