Cartoon Style Rendering

Simon M. Danner[*]
TU Wien
0526996

Christoph J. Winklhofer[†]
TU Wien
0426461

Abstract

The objective of Cartoon Style Rendering is to produce a cartoon looking rendering out of a 3D scene. Instead of rendering photorealistic images the focus lies on artistic and stylistic characteristics. In this paper we give an overview of the common methods to generate cartoon looking scenes. We describe the basic techniques for the silhouette edge detection and the shading of the interior. Then we show how to enhance these techniques to mimic various cartoon styles and methods for special effects. We present methods for silhouette edge detection that operate in image-space as well as in object-space. For the interior shading we describe cel-shading, self-shadowing, diffuse-lighting and specular highlights. Then several approaches for enhanced cartoon style rendering follow, namely Chinese Painting, Pencil Drawings, and different styles of comic artists. Afterwards we show some possibilities for special effects in cartoon style renderings, including simulation of liquids and smoke as well as methods for stylistic shadows.

CR Categories: I.3.7 [Computer Graphics]: Three dimensional graphics and realism—;

Keywords: cartoon style rendering, rendering, non-photorealistc rendering, artistic

1 Introduction

Cartoon style rendering is a part of the non-photorealistic rendering (NPR) techniques. The field of non-photorealistic rendering aims at simulating handmade illustrations with computer algorithms. In other words, non-photorealistic rendering imitates non-photographic techniques, such as pen and ink drawings, paintings or comic style drawings [Gooch et al. 1998; Spindler et al. 2006].

NPR techniques are often useful when we want to emphasize on a particular aspect of the picture we want to create, for example in a technical illustration many details are left out to show the most important features, which are the only ones we are concerned of. Its the same with cartoons and comic, studies have shown that we can perceive the emotions of cartoon drawn faces much more easily than of photorealistic ones. So we see that the abstraction, which cartoon style renderings and all the other NPR styles use, help us to emphasize on the important features we want to show. Different applications need a different level of abstraction and need to concentrate on different aspects of the content.

[*]e-mail: simon.danner@student.tuwien.ac.at
[†]e-mail: christoph.winklhofer@student.tuwien.ac.at

This paper starts with a short summary of several non-photorealistic rendering applications and describes their underlying approach and their usage.

One application area of NPR are **technical illustrations**. This term summarizes illustration conventions used in technical manuals, illustrated textbooks, encyclopedias and similar forms. The communication of geometry information is their primary objective. A normal photograph would confuse the observer by the details. Therefore the object is described in a more abstract form. Leaving out unnecessary information in the final image improves the perception of the objects main characteristics [Gooch et al. 1998].

Another approach of NPR is the emulation of various artistic styles. There are many different styles of artistic drawings, depending on the country and the preferences of the artist. Some combine different styles or even create there own ones. Therefore we only mention a selection of styles, where methods for NPR rendering already exist.

Watercolor paintings are created with a water soluble color, composed of pigments. Several different colored layers are applied with a brush to the painting background. The background color shimmers through, due to the transparent colors. In chinese and japanese painting the watercolor technique is the dominant form and is often drawn in monochromatic black or brown tones. This aesthetic form of NPR was used in the computer game Okami *(see Figure 1)*, which was released in the year 2006. A similar NPR technique that simulate a chinese painting style will be discussed later.

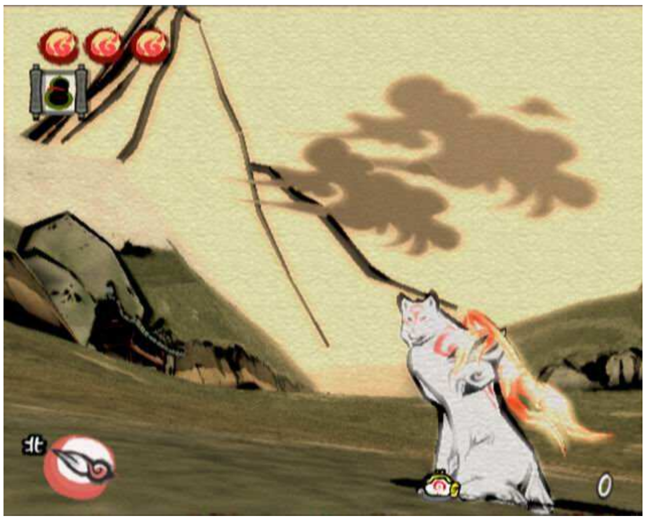


Figure 1: *Still from the computer game Okami, which uses a NPR chinese painting technique in realtime.*

Another common drawing style is **pencil rendering**. It is used for sketches but has also established as an autonomous art style. The basic element in a pencil drawing is the line. Outlines represent the main features of the object and are drawn usually with a thicker line. Interior regions are filled with hatching and simulate spatial effects or different tones. Normal hatching uses parallel lines, whereas in cross hatching, as the name already explains, uses crossed lines. An NPR technique for pencil rendering in real time is also discussed in the paper *(see Figure 2)*.

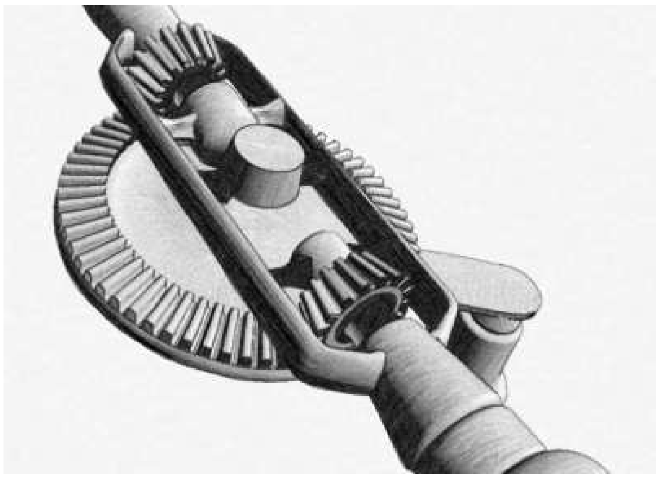


Figure 2: *Drill. Generated with an NPR pencil rendering method in realtime.*

The next NPR method is **cartoon style rendering**. As the name suggests, cartoon style rendering has its origin in comic books and cartoon movies. Essential of the traditional cartoon stylized look are the black outline and the solid color of the interior regions *(see Figure 3)*. The aim is to produce a flat looking 2D comic style image out of a 3D model. This basic cartoon style rendering technique is called cel shading, also know as toon shading. It is used in a few animation movies, for example the recent movie ”Simpsons the movie”, but is also required in real time applications, like computer games (Okami, Zelda Windmaker, etc).

Another field of application is the rapid prototyping of cartoon movies. Before an artist starts with the time consuming painting and inking process, an animation environment can be used to test animation, coloring and shot angles [Lake et al. 2000].

Besides the traditional cel shading more sophisticated cartoon rendering methods have been developed. Instead of only replicating the material of an object, NPR methods for enhanced carton style rendering have been found, which mimic a particular drawing style of comic artists, like the Sin-City style of Frank Miller or the graphic style of Mike Mignola in the hellboy comic series.

It is difficult to model a realistic looking human character. Paradoxically they look strange to the viewer, even when the textures come from photographs of real people. It is the same with the animation of characters. People are able to appreciate a life like motion for an animated character in cartoon style, but for a photorealistic human model a similar animation is annoying [Lander 2000]. Cartoon style rendering is capable of broadening our ability to communicate thoughts, emotions, and feelings through computers. More people can identify themselves with the characters and the story, if the scene and included objects are in an abstract style [Lake et al. 2000].

Silhouette edges are an important factor in cartoon drawings. They express shapes and the visual style of an cartoon character *(see Figure 4)*. In short, a 3D computer model used in computer games or animation movies is defined by vertices, which are connected by edges. The faces of the object normally are triangles, built up by the edges. In NPR a silhouette edge is an edge we want to stroke out in the final rendering. Silhouette edges are distinguished in contour

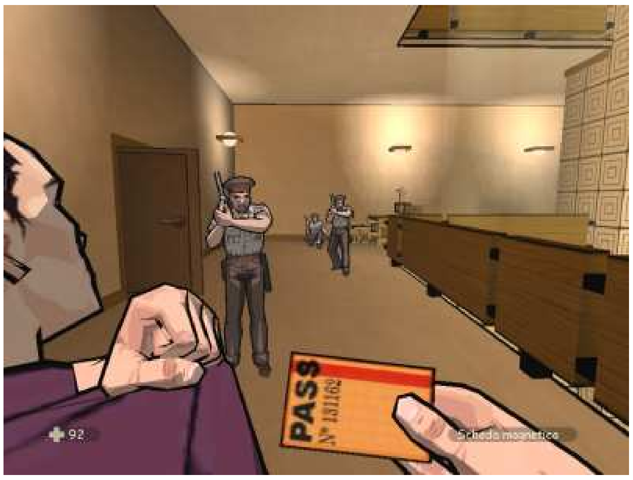


Figure 4: *Still from the computer game XIII. Black silhouette edges simulate an aesthetic comic look.*

edges, crease edges and marked edges. The geometric classification of an edge is done by some algebraic vector calculation of the surface normal vectors adjacent to an edge. If the edge is shared by a front and back a face, it is classified as a contour edge, also called a silhouette. Crease edges are defined by the angle of the faces normal vector and marked edges which are defined by the modeling artist. Back and front faces can be found by the dot product of the face normal vector and the vector of the viewing direction *(see Figure 5)*. The dot product of a front face is negative and for a back face it is positive or zero. The dot product of two vectors is the cosine of the angle between them and is used to classify the silhouette edges.

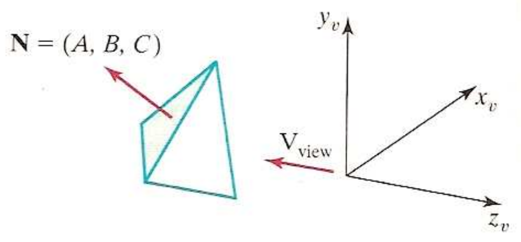


Figure 5: *Back face polygon. N is the surface normal vector and V is the viewing vector.*

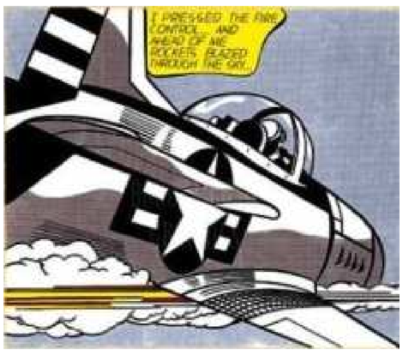
In the traditional shading model *(see equation 1)* the color of a surface is not calculated physically correct. A lighting model - which is a synonym for shading model - approximates the physical law to reduce the computation complexity. The color of a surface is the sum of an ambient, diffuse and specular term. The realation of the vectors used in the traditional shading model from *equation 1* are specified in *Figure 6*.

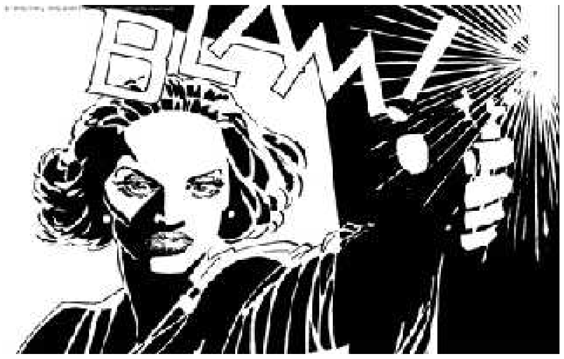
$$I = I_a * k_a + I_d * k_d * (L \cdot V) + I_s * k_s * (V \cdot R)^n s \qquad (1)$$

The ambient term $I_a * k_a$ approximates the background lighting and the global diffuse reflections. I_a is the ambient light and k_a the ambient material coefficient. The ambient term produces a flat light.

Diffuse reflection assumes that each incident light is scattered with equal intensity in all directions. The reflecting surfaces are also

(a) Zelda Windwaker.


I PRESSED THE FIRE CONTROL... AND AHEAD OF ME ROCKETS BLAZED THROUGH THE SKY...
(b) Pop-art image by Roy Lichtenstein.



(c) Sin-city by Frank Miller.

Figure 3: *Several examples for a cartoon stylized look. Cartoon style rendering used in the Computer game Zelda Windwaker (a). Pop-Art image by Roy Lichtenstein with smoke particles (b). An enhanced cartoon style look, like the double contour lines, is used by Frank Miller in his Sin-City comic series (c).*

called diffuse reflectors and are approximated by the Lambert's cosine law $I_d * k_d * (L \cdot V)$ *(see Figure 7)*. I_d is the diffuse light and k_d is the diffuse material coefficient. The number of light rays intersecting a surface is proportional to the dot product of the light direction vector L and the surface normal N.

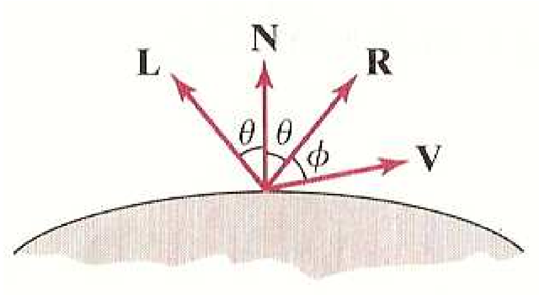


Figure 6: *Vectors for the traditional shading model. L is vector to the light direction. N is the surface normal. V is the vector to the viewing direction. R is the reflection vector.*

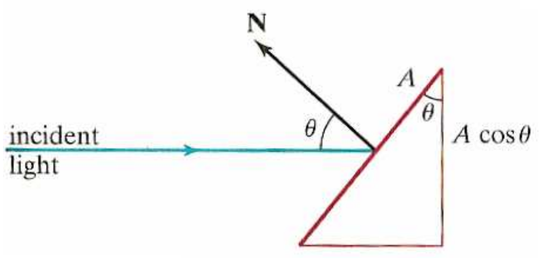


Figure 7: *Lambert's cosine law. The amount of reflected light is proportional to the angle between the incident light and the surface normal N.*

The specular term simulates the shininess of a surface and depends on the material properties. Dull surfaces have a wider reflection range than shiny surfaces. The empirical calculation mode uses an exponential value of the dot product from the viewing and the reflection vector. A shiny surface has high exponent. I_s is specular light, k_s is specular material coefficient, V is viewing vector, R is reflection vector, and ns is shininess [Hearn and Baker 2003].

On todays consumer graphics hardware the integrated standard shading model for surfaces is goraud shading. In goraud shading the color for each vertex is calculated with the traditional shading model and interpolated linearly over the surface. This results in an smooth surface color.

The aforementioned traditional shading technique is also the foundation of the interior shading in cartoon style rendering and essential to understand NPR processes. The interior shading of a cartoon looking scene is done with discrete color values. This means instead of a smooth surface color, as it is calculated by the goraud shading method a solid shaded surface, or in other words a more stylized shaded surface is desired. This is the reason why the result of the traditional shading technique can not be used without any changes to color the interior regions. Several NPR methods, which utilize this value are introduced in the rest of this paper.

We describe the basic techniques for creating cartoon looking scenes, which consists of the silhouette edge detection and the interior shading (see section 3). Furthermore we present methods extending the basic technique to generate pencil drawings (see section 4.1) and chinese paintings (see section 4.2). Afterwards we present several methods to render cartoon styles of different comic artists (see section 4.2.1) and methods to process effects, like liquid surfaces, smoke and shadows (see section 4.3).

2 Previous Work

A variety of work has been done in comic- and non-photorealistic rendering. In this paragraph we review these previously published works.

Decaudin built a non realtime rendering system for a 3D scene with a cartoon looking output, using OpenGL [Decaudin 1996]. They use a multi-pass rendering approach to detect edges. Discontinuities in the z-buffer and normal-buffer are detected and used for drawing the edges. The z-buffer from the light source is used to calculate shadow maps and the composition of all rendering passes produce a cel shaded picture. The output looks like a basic comic with shadows.

Lake et al. show techniques for emulating cartoon style graphics in real time. Various different styles were achieved [Lake et al. 2000]. Using textures for contours and for the interior of the cel (a cel in cartoons is usually an area surrounded by an inked line). Real time pencil sketching can be emulated. New methods for detecting silhouettes and crease edges using the normal of the faces are also proposed. This rendering system scales to the hardware used and renders cartoons or sketch style graphics in real time.

Silhouettes are of artistic importance for cartoon style graphics and play a very important role in NPR in general. Wang et al. as well as Hertzmann introduce important techniques and algorithms for silhouette detection [Wang et al. 2004; Hertzmann 1999]. Algortihms are analyzed and classified according to their use.

Gooch et al. use non-photorealistic rendering methods to create technical illustrations [Gooch et al. 1998] . These are normally hand drawn, but with the right degree of abstraction and coloring and adjusting the hue and the luminance, they show a technique for creating technical illustrations based on a 3D scene.

Silhouette, edge and crease detection algorithms often perform per-edge and per-face mesh computations using global adjacency information, which is unsuitable for an implementation on modern hardware using pixel- or vertex-shaders, where only local information is accessible. McGuire and Hughes describe a technique to pack global adjacency information into a vertex data structure [McGuire and Hughes 2004] . Using this algorithm they manage to compute feature-edges entirely on the hardware. Furthermore they use it to draw thick contours and describe methods for mapping stroke textures on the thick contours.

The aforementioned resources shows only techniques for rendering cartoon style imagery of a static scene and characters, but not effects like liquid animations or smoke. Eden et al. publish a method to render a liquid surface obtained from a three-dimensional physically based liquid simulation system in cartoon style inspired by cartoons such as Futurama and The Little Mermaid [Eden et al. 2007]. They use bold outlines to show depth discontinues, patches of constant color to highlight near-silhouettes and areas of shininess. With these approaches they achieved cartoon style water animations.

Diepstraten and Ertl extend the hard shading algorithm for cel shading from Lake et al. to render transmissive and reflective surfaces in cartoon style [Diepstraten and Ertl 2004; Lake et al. 2000]. They use new methods because techniques for photorealistic rendering can not be applied.

Selle et al. propose a technique for producing artistic and physically correct smoke renderings [Selle et al. 2004] . They use the output of a smoke simulator to obtain physically correct behavior and create an abstract picture using simple color and line strokes only.

McGuire and Fein use a similar approach for rendering smoke [McGuire and Fein 2006]. They use a particle system to render animated smoke in real time. For self-shadowing effects they use nailboard shadow volumes that create complex shadows from few polygons, only three polygons are drawn per particle. For smoke simulation they use a simulator that is artistically controllable. This method is very usable for application where performance and expressiveness is needed, for example games and rapid development systems for animations.

Petrovic et al. invented a system to support comic artists with drawing shadows in an existing, hand drawn, scene [Petrovic et al. 2000]. The method is semi-automatic and the input from the user is required. The user has to specify the depth of various objects in the scene. The method employs a scheme for ”inflating” a 3D figure based on the hand-drawn art. This method is especially useful when shadows are cast by a complex object and/or fall over interesting shapes.

Cartoon drawing is an art, and contrary to photorealistic rendering cartoon style rendering can be used to produce many different styles. If we think of cartoons or comics we see that hardly any cartoon looks like another, each one has a distinct style.

For example traditional chinese painting is famous for its freehand brushwork an lingering spirits. It is primarily done with a brush pen dipped in pine soot made ink. Cartoons made by this technique are called ink-and-wash cartoons. Yuan et al. propose a method for rendering and animating this style of comic [Yuan et al. 2007]. They use a GPU-based real-time approach that automatically converts animated 3D models into Chinese-style cartoon. The rendering process uses interior shading, silhouette extracting and background shading. This method can be used to create games and movies and greatly simplifies the process of creating animated chinese cartoons.

Spindler et al. enhance the cel rendering techniques to achieve rendering styles inspired by Frank Miler’s ”Sin City” and McFarlane’s ”Spawn” [Spindler et al. 2006]. They implement four new cartoon-like rendering styles, namely: stylistic shadows, double contour lines, soft cel shading and pseudo edges. All of them are applicable for real-time rendering and can be used in stylistic games.

Lee et al. present a real-time technique for rendering 3D scenes in pencil drawing style [Lee et al. 2006]. They incorporate the characteristics of pencil drawing into the rendering process, which runs on the GPU. For contours, a multiple contour drawing technique is used that imitates trial-and-errors of humans in contour drawing. A simple approach for mapping oriented textures onto surfaces was presented for interior shading. The quality of pencil rendering was improved compared with previous methods, textures that reflect the properties of graphite pencils and paper were generated and mapped.

3 Techniques

3.1 Silhouette edge detection

Silhouettes define the basic shape of an object. They are view based dependent that is they have to be recalculated when the viewpoint (eye point) changes. In a realtime or interactive system this calculation is typically updated every frame. The silhouette edge detection (SED) algorithms can be classified in image space and object space methods.

A recapitulation of the silhouette definition from the introduction paragraph is given before this two SED algorithms are discussed in more detail. The outline or silhouette is an edge which connects a back facing polygon to a front facing polygon. A back facing polygon is invisible to the viewer and a front facing polygon is potentially visible. In *Figure 8* the silhouette edge highlights the object from the background and describes the objects profile. Internal discontinuities are also represented by this type of silhouettes.

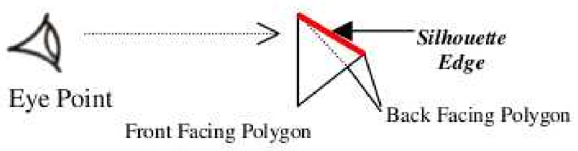


Figure 8: *Silhouette edge connecting a front face and a back face polygon.*

The next silhouette type is the crease edge. They can be classified in a ridge edge and a valley edge. Both depend on the dihedral angle of the adjacent polygons. For ridge edges the value of the angle is less than a threshold and for valley edges it is greater than a specified threshold. Material edges are defined by the artists during the modeling process. They are computed in advance and must not be recalculated during the animation phase every frame. Another silhouette type, called boundary for non solid objects, for example a sheet of paper, exists. It has a marginal role in cartoon style rendering, where mostly closed objects are used. *Figure 9* shows the various types of silhouette edges, where *B* is a boundary edge, *C* is a crease edge, *M* is a material edge and *S* is a silhouette.

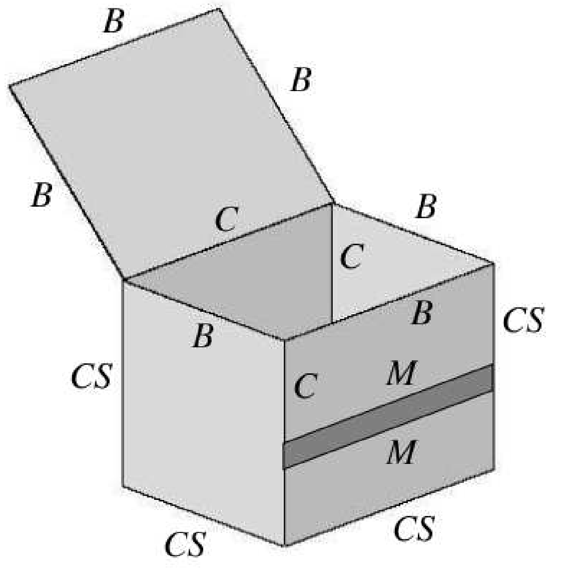


Figure 9: *Several types of silhouette edges. B is a boundary edge. C is a crease edge. M is a material edge. S is a silhouette.*

Image SED algorithms detect edges with image processing methods. They do not need the 3D geometry information of an object, because the algorithm works with the final rendered 2D image of the object or scene. Furthermore the computational complexity corresponds to the image resolution and is constant if the image resolution remains constant. The amount of geometry data has no influence on the calculation time and effort required, so image based algorithms scale well.

A simple approach is to use the color buffer (which contains the color value for every pixel) to detect silhouettes with image space SED algorithms. However the result is not convincing, because it will not find an edge for overlapping objects with the same color. Moreover objects with detailed textures will produce edges with no relevance to the object.

Another approach is to use geometry characteristics, like the depth buffer and a normal map. On todays hardware the visibility detection is done with the depth buffer. It is an image space method and contains the depth value for every visible pixel. The scene is rendered and the depth values are extracted. A edge detection algorithm is applied to the depth values stored in an image or a texture. Depth values between different objects are large, whereas depth values for the same object are small. With the depth buffer only C0 surface discontinuities can be detected, that means crease edges are not found.

The normal map contains the components of the normal vector for every pixel. Each value in the color-buffer, depth-buffer and normal buffer is associated to a corresponding 3D point. The values in the normal map can be produced by rendering the scene two times. The material coefficients of each object are set to a pure diffuse light and three directional lights with the colors red, green and blue are positioned on the coordinate axis. In the second rendering step the light positions are inverted on the coordinate axis and finally the normal components are computed by subtracting the two results of the light intensities [Decaudin 1996].

Another method to fill the normal map is to use a shader. A shader makes it possible to alter the fixed-functional pipeline of the graphic processing unit with a user defined program. On todays graphic hardware three stages of the pipeline are represented, namely the vertex-, fragment- and geometry-stage. The latter is only presented on modern hardware and is not needed for the normal map calculation. In the vertex shader the normal is sent to the fragment shader and automatically interpolated over the polygon surface. Afterwards in the pixel shader - because of the interpolation - the normal can be accessed per pixel. An image processing method for edge detection is applied to the normal map. C1 discontinuities - detection of crease edges - can be found by the normal map method. Afterwards the two images are combined to get the silhouettes *(see Figure 10)*.

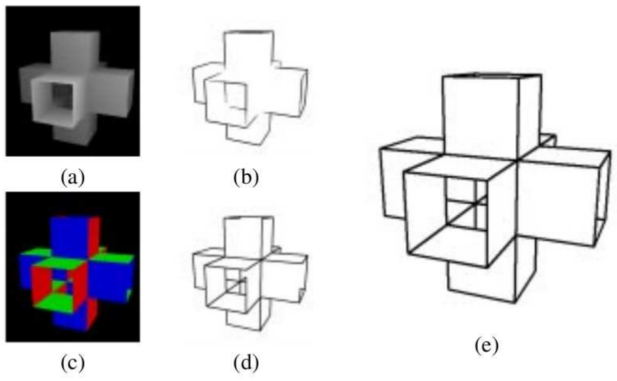


Figure 10: *(a) Depth buffer. (b) Detected silhouette edges using the depth buffer. (c) Normal map. (d) Detected silhouette edges using the normal map. (e) The Final image is a combination of the edges detected by the depth buffer and the normal map.*

A drawback of image space SED algorithms is that the 3D information is no more available and the generation of stylished edges is more complex than to object space methods. Object space methods exists for polygonal meshes and free-from surfaces.

An easy method to detect object space methods for polygons is to check all edges. Two normal vectors of the adjacent polygon faces are stored for every edge. Afterwards the result of the dot product is evaluated. For non-interactive animation this simple method is sufficient. For real-time applications it is too time consuming, because the silhouette edge have to be recomputed every frame.

The idea is to take advantage of coherence information to speed up this calculation and test only a few edges. The first coherence information to improve the performance is that a silhouette edge never exists in isolation. There is always an neighboring silhouette edge at the two points defining an edge. Small changes in the viewing parameters is the requirement for the second coherence information. The possibility is very high that a given chain of silhouette edges contains edges that were also detected as silhouette edges in the previous frame. Markosian describes an algorithm to randomly select edges for the testing process [Markosian et al. 1997]. The algorithm is about 5 times faster than the brute force method, but there is no guarantee to detect all edges.

The silhouette edge detection for a free form surfaces is different to objects represented by polygons. The most common representation of free form surfaces are NURBS (Non-Uniform Rational B Splines) and subdivision surfaces. The silhouette for smooth surfaces is the set of points x_i where the surface normal n_i is perpendicular to the viewing vector *(see equation 2).*

$$n_i * (x_i - C) = 0, \tag{2}$$

where C is the position of the camera center *(see Figure 11).*

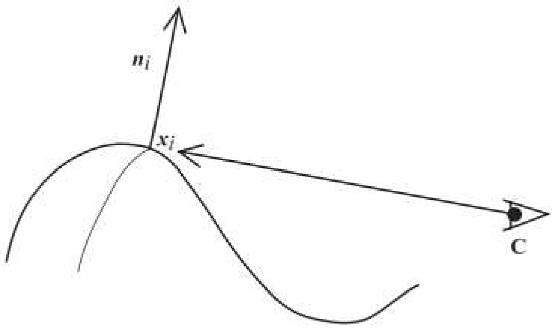


Figure 11: *The silhouette of a smooth surface is the set of points where the surface normal is perpendicular to the viewing vector.*

Hertzmann describes an algorithm to find the silhouette edges of free form surfaces approximated by a triangular mesh [Hertzmann 1999]. In the first step they calculate the dot product of the surface normal and the view vector for each vertex. Two perpendicular vectors have a dot product of 0, so the goal is to find these. Because the value of the dot product varies smoothly over the surface only the sign of the dot product is interesting. If the sign of the two edge vertices is different than there must be a silhouette point on this edge. The position on the point is computed by an linear interpolation along the specified edge. Finally a silhouette is generated between the determined silhouette points of the triangle mesh. This process is repeated for every triangle *(see Figure 12).* For large triangles the generated silhouette edges are very coarse and not every silhouette edge can be detected.The solution for this problem is to produce smaller triangles and refine the mesh.

Typically the silhouette edges are rendered with a simple black line. However more artistic line styles can be generated with the additional geometry information available within the object space SED algorithms. Various textures can be applied to the silhouette edges to achieve more sophisticated line styles.

Lake present a method that a texture follows the curvature of the edge [Lake et al. 2000]. Three different textures corresponding to the curvature of the line strokes are used *(see Figure 13).* The chosen edge texture depends on the angle of the edge E_1 with it's successor edge E_2. The angle can be determined by the dot product of this two vectors. If the angle of the two corresponding edges is smaller than a specified value d a texture with a leftward stroke is applied. If the angle is greater than the value d a rightward stroke is used. Otherwise a straight stroke is mapped onto the quadrilateral representing the edge *(see equation 3).*

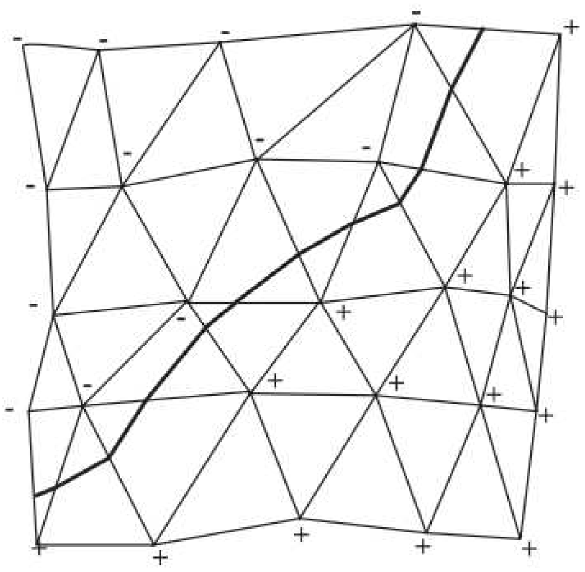


Figure 12: *Silhouette points lies on edges with a different sign of the dot product for the vertices. The position is determined by a linear interpolation. Afterwards all resulting point are connected to form the silhouette edge.*

$$E_1 \cdot E_2 = \begin{cases} \leq -cos(d) & \text{use leftward texture} \\ -cos(d) < 0 > cos(d) & \text{use straightward texture} \\ \geq cos(d) & \text{use rightward texture} \end{cases} \tag{3}$$

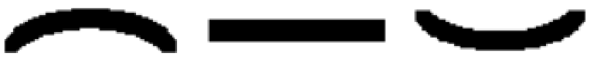


Figure 13: *Different textures for generation of stylized strokes (from left to right): Rightward stroke, Straight stroke and Leftward stroke*

A simplified 2D scene shows the difference between a drawing using only straight strokes and a more stylized version with silhouette edges depending on the curvature *(see Figure 14).* Other artistic effects can be generated by varying the width of the quadrilaterals. However this algorithm produces new problems while rendering the stylized silhouette edge. The effect can be observed easily by the viewer in regions with high curvature. Moreover the textured quadrilaterals can overlap with the polygons of the texture and gaps in the silhouette edge can result for quads with sharp angles.

3.2 The Painter

The aim of the Painter is to produce a flat looking image out of the 3D models. With this method a variety of styles can be produced by varying the shadow an highlight parameters and weighting factors of the shading calculations.

By creating animations and cartoons the artists deliberately abstract from the real world and reduce the visual detail in order to emphasize the emotions and humor in the story. In contrary to shading the character in 3D, the cartoon artists use a solid color that does not change in the over cel expect for maybe a hard edge between the

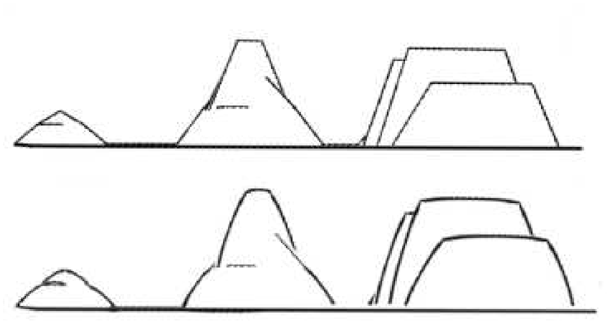


Figure 14: *Scene with straight silhouette edges and a scene with stylished strokes depending on the curvature of the edge.*

shadowed and illuminated surface. In the shadow part of the object an artist will often use a darkened color of the main material color. This helps to add lighting to the scene and adds cues to the shape and context of the character in a scene. There is mostly a boundary between the light and the dark color, which is a hard edge, determined by the properties and shape of the character or object. This technique is called hard shading.

This hard shading technique proposed by Lake et al. relies on texture mapping and the mathematics of traditional diffuse lighting [Lake et al. 2000]. It tries to find a transition boundary and shades on each side of the boundary with a solid color, rather then interpolating smoothly across the model. *equation 4* is used to calculate the lightning for non cartoon styles with gradient color transitions:

$$C_i = a_g \times a_m + a_l \times a_m + Max(\overline{L} * \overline{n}, 0) \times d_l \times d_m \qquad (4)$$

C_i is the vertex color, a_g is the coefficient of global ambient light, a_l and d_l are the ambient and diffuse coefficients of the object's material. $\overline{L}$ is the unit vector from the light source to the vertex, and $\overline{n}$ is the unit vector normal to the surface at the vertex. The result of $\overline{L} * \overline{n}$ is the cosine of the angle formed between the two vectors.

The math for the hard shading algorithm is essentially the same. But only a discrete number of colors will be used, mostly two (one for the illuminated part and one for the shadowed part of the character). These two colors can be set for each material or can be calculated. The color for the illuminated part can be calculated if we set the dot product of *equation 4* to 1 (as if the light vector and the normal vector point in the same direction) and the shadow color can be computed if we set the dot product in the equation to 0 (as if the light vector and the normal vector are orthogonal to each other). If more colors are desired the dot product can be set to an arbitrary value for the computation of the color. This results in a one dimensional texture containing two or more texels. This texture is only needed to be computed once for every material and therefore can be computed at startup or when creating the model.

Just as in calculating the colors for smooth shading, the colors for the hard shading algorithm depend on the cosine of the angle between light and normal vectors. For every frame and vertex $Max(\overline{L} * \overline{n}, 0)$ is calculated and used as texture coordinates for the pre-computed one-dimensional texture map. A threshold is used to decide which texel is chosen. For example when $Max(\overline{L} * \overline{n}, 0) <$ 0.5 the shadow color is used and for $Max(\overline{L} * \overline{n}, 0) >= 0.5$ the illuminated color is used *(see Figure 15)*. This creates a hard edge between the two colors. If more colors are desired then for every color an interval for the dot product is needed.

Figure 16 shows a rendering of an character with these methods.

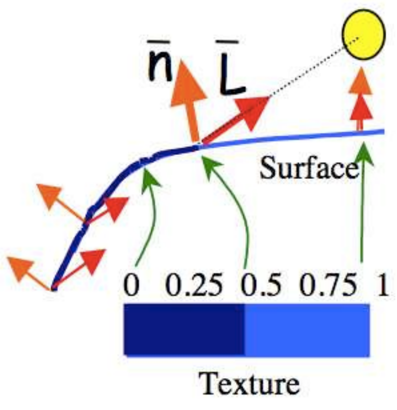


Figure 15: *Generation of texture coordinates from* $Max(\overline{L} * \overline{n}, 0)$. 0.5 *is used as the shadow boundary.*

We can clearly see the hard edges between the shadowed and the illuminated surface. The look of traditional cartoons can be simulated very well.

If you examine the edge closely it may look jagged. In this case 3D graphic APIs offer texture-filters which can help. The linear texture-filtering mode smoothes the color boundaries, but if the polygon is too big, than the smooth transition may be too wide. The "nearest" texture-filtering mode chooses the nearest texel to a pixel and results in an applicable result.

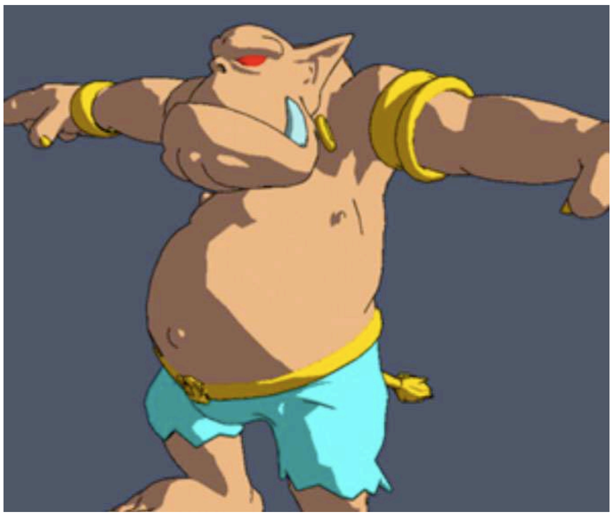


Figure 16: *Ogre Olaf rendered with hard shading and inked silhouettes.*

Notice that this method needs to do a dot product for each vertex per frame. This can easily be done on the GPU which would increase performance, it will be explained later.

Here are the preprocess and runtime portions of the hard shading algorithm as in [Lake et al. 2000]:

- Preprocess
  1. Calculate the illuminated diffuse color for each material:

$$C_i = a_g \times a_m + a_l \times a_m + d_l \times d_m$$

2. Calculate shadowed diffuse color:

$$C_s = a_g \times a_m + a_l \times a_m$$

3. For each material, create and store a one-dimensional texture map with two texels using the texture functionality provided by standard 3D graphics API. Color the texel at the $u = 1$ end of the texture with C_i and the texel at the $u = 0$ end of the texture with C_s.

- Runtime

1. Calculate the one-dimensional texture coordinate at each vertex off the model using $Max(\overline{L} * \overline{n}, 0)$, where $\overline{L}$ is the normalized vector from the vertex to the light source location, $\overline{n}$ is the unit normal to the surface at the vertex, and $\overline{L} * \overline{n}$ is their vector inner (dot) product.

2. Render the model using a standard graphics API with lighting disabled and one-dimensinal texture maps enabled.

By customizing some values of the hard shading algorithms it can be used to create a variety of different styles. The colors can be set manually instead of computing them of the material, so the artist can choose the exact colors he wants to use. When a high contrast between the two colors is chosen, the picture is rendered in a dark style. Two light sources can be simulated by using three colors, with the middle one lighter then the other ones.

Decaudin takes a completely different approach to render the flat looking interior of the cels of the cartoon drawings [Decaudin 1996]. They use the Phong shading algorithm and eliminate the $\overline{L} * \overline{n}$ portion of it, so the whole cel looks flat, using only a solid color that does not change. To get a 3D looking effect for the image using only solid colors, the shadow is calculated using the shadow map technique. Therefore a z-Buffer Image from the viewpoint of the light source has to be calculated. To obtain this buffer, the scene is rendered with the camera replacing the light source and then the z-Buffer is read back. This has to be done for each light source.

To determine if a pixel is lightened or not by a light source, it is compared with its corresponding pixel on the z-buffer image from the light source. If it is visible on the image from the light source it is lightened, and if it is not visible on the image from the light source it is darkened, this calculation is done for every light source. *Figure 17* shows a simplification of a scene with lighting. It can be seen which transformations have to be necessary to calculate the shadows.

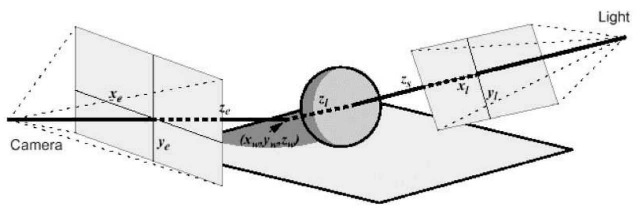


Figure 17: *Calculation of the projected shadow*

For each pixel (x_e, y_e) of the image:

- Calculate the coordinates (x_w, y_w, z_w) of 3D point corresponding to this pixel in the world frame (obtained from x_e, y_e, value read into the z-buffer z_e, and the 4x4 camera projection matrix),
- Calculate the coordinates (x_l, y_l, z_l) of the 3D point in the light frame,
- Compare z_l and the value z_s read from the z-buffer of the light at position (x_l, y_l): if $z_l > z_s$ then the point is not "seen" by the light, it is into shadow, else it is lit.

This method has a big problem because the z-buffer has a fixed resolution. When an object viewed from the camera is big, but viewed from the light source is small the shadow appears aliased. To overcome this problem many different shadowing methods exist, for example calculating the values of the neighboring pixels and averaging the result. This technique could also be use for backface shadowing of the objects, but it is not accurate enough and would not lead to a hard edge between the illuminated and the dark surface of the object. But since the $\overline{L} * \overline{n}$ is already computed, it can be used to calculate if the polygon is not illuminated. When $\overline{L} * \overline{n}$ is less then than the polygon is opposed to the light source.

By using this algorithm we obtain an image with backface and projected shadows for every light source. After this all images have to be merged into one to obtain the final image with shadows from multiple light sources.

This algorithm has the advantage to the previous one that it renders shadows from multiple sources correctly (not just simulating it) but it is not as efficient. The two algorithms could be combined, using the technique for coloring the cels from the algorithm from Lake et al. and implementing multiple light source such as it is done in the algorithm from Decaudin and using an efficient algorithm for calculating projected shadows [Lake et al. 2000; Decaudin 1996].

Winnemoller extends the former standard algorithm for comic style rendering with a comic style specular component [Winnemoller 2002]. Secular lightning is often used in classic, hand drawn comics for shiny objects like cars, weapons, glass, etc. Drawings or renderings use specular lighting usually not to increase realism but to give cues about surface properties such as the material and the geometry. The aforementioned lighting model only uses diffuse light and the aim is to extend this lighting model with a specular component while preserving the banded-shading look. *Figure 18* shows some examples of traditional hand drawn cartoons that use specular lighting. As we can see the specular highlights help to create a more 3-dimensional look while still preserving the cartoon style look. While doing so, they introduce two geometric approximations, namely the *perspective projection correction angle* and the *vertex face orientation measure*.

To create the specular component for the cartoon shader we take a look at the phong reflection model, N is the normal at any given point P, L is the light vector, V is the vector spanning from the viewer's position to the point P (which is not necessary the direction the user is looking at), R is the reflected light vector, α is the shininess constant of the material, i_s and i_d are the intensities of the specular and the diffuse component respectively and k_s, k_a and k_d are the reflection constants of the specular, diffuse and ambient components respectively. *equation 5* is the phong lighting model.

$$I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s) \tag{5}$$

$$R \cdot V = L \cdot V - 2(N \cdot L)(N \cdot V) \tag{6}$$

equation 6 the expansion of $R \cdot V$. The diffuse component can be easily computed, but is view independent. The specular compo-

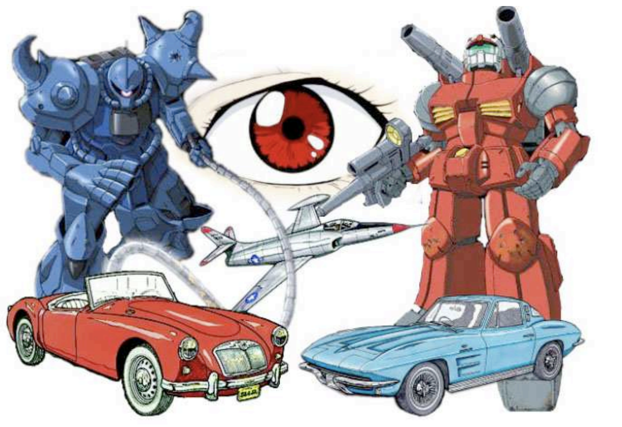


Figure 18: *Example of cartoon drawings that use specular lighting.*

nent $k_s(R \cdot V)^{\alpha} i_s$ extends the creative potential of comic renderings through view dependency and more freedom using shading maps.

In theory the extension of the diffuse only shading approach is pretty easy, instead of using a 1D texture map for use with as a shading texture a 2D texture is used. One dimension indexes the diffuse part $(N \cdot V)$ and the other dimension indexes the specular component $(R \cdot V)$. The shininess value should be applied before normalization and indexing. *Figure 19* shows a 1D and two 2D textures. The y-axis is describes the diffuse component and the x-axis the specular component. As we can see many different effects can be achieved by using different textures.

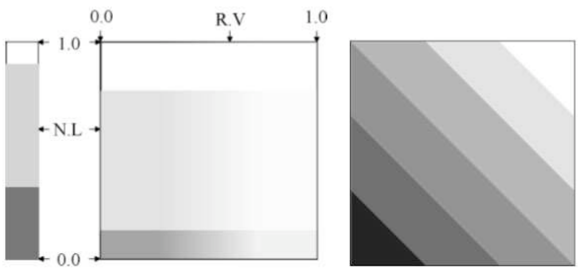


Figure 19: *Examples for 2D-textures for specular shading*

The problem with this approach is the computation of the indexing specular lighting value. As we can see in the former equation the calculation needs some heavy computational overhead, especially since this expression has to be calculated for every vertex. This is due to the fact that the specular component is dependent on the view vector (unlike the diffuse component) but this is the very thing that causes the cartoon renderings with a specular component to be visually more appealing than the ones without it. Therefore the goal is to approximate the exact solution, which remains visually close to the more expensive exact solution.

To include a specular component into the comic lightning model for each vertex additional computations are needed and these calculations easily become a dominant role in the whole rendering process. This leads to the introduction of a face-orientation determination with this goal:

> ”Even though determination of face-orientation will introduce a further load on the system, it would still benefit us if this load were to be balance by a decrease of vertices that have to be included in the lighting calculations as well as a decrease in triangles that have to be rendered.” [Winnemoller 2002]

With the standard approach (with the lack of information about face orientation), lighting values have to be calculated for each vertex and then all triangles have to be rendered twice (once for the silhouette (assuming that an object space algorithm is chosen) and once for the shading of the interior. So the cost of the standard algorithm is shown in *equation 7* (*Ver* are the computations per vertex and Tri are the computations per triangle).

$$\Theta(Ver + 2 * Tri) \tag{7}$$

It should be noted that the performance calculations get slightly skewed up when implemented, because part of the algorithm is run on the host computer (e.g. custom lighting calculations and manual face sorting and part is run on the graphics cards (i.e. rendering of triangles and face culling). Additionally in some cases it is faster to render more faces while using display list optimization than rendering fewer faces without display list enabled.

So how to determine if a face is front- or back-facing? The correct way to do so is to spawn a vector V from the camera's/viewer's position to the center of the face and then comparing it with the normal N of the face. This is done by calculating $N \cdot V$. If the result is smaller than zero the face is back-facing and when the result is bigger then zero it is front-facing. The method used for flagging faces as back- or front-facing has an big performance impact, therefore we will discuss different methods.

The first method is to mark the triangles during traversal of the triangles. All triangles are traversed and marked as back- or front-facing, also the vertices have to be marked when they are part of a front-facing triangle. Then the vertex list is traversed and the light-values for the front-facing ones are calculated. Then front and back-facing triangles can be rendered separately. If b is the proportion of the number of front facing-triangles to the total number of triangles the costs is shown in *equation 8*.

$$\Theta(b * Ver + 2 * Tri) \tag{8}$$

We can assume that b is approximately 0.5 because the percentage of the front-facing faces is most likely 50%. Therefore this approach is faster than the standard approach when $b < 1$ and this is always the case when there are back-facing faces.

Another approach is marking the triangles during traversal of vertices. For each vertex it is checked if it is front or back-facing and then mark the attached triangles. This is useful because a well behaved, closed object has much more triangles than vertices. Winnemoller defines the normal of a vertex as follows: *”The normal of an object-vertex is the average of the normals of all triangles of which this vertex is a part”* [Winnemoller 2002]. Because of this definition the face orientation can be concluded from the vertices that define the face. But the normals from the vertices mostly differ from those of the attached faces, but with finely tessellated objects the difference is negligible.

The algorithm is as follows:

> Traverse the vertex list and determine the orientation of a vertex. If it is front-facing, perform the necessary lighting calculations and mark the attached triangles as front -facing. Otherwise mark attached triangles as back-facing. [Winnemoller 2002]

An alternative is to check the vertices only when the triangle is rendered. This adds the benefit that different rules can be applied

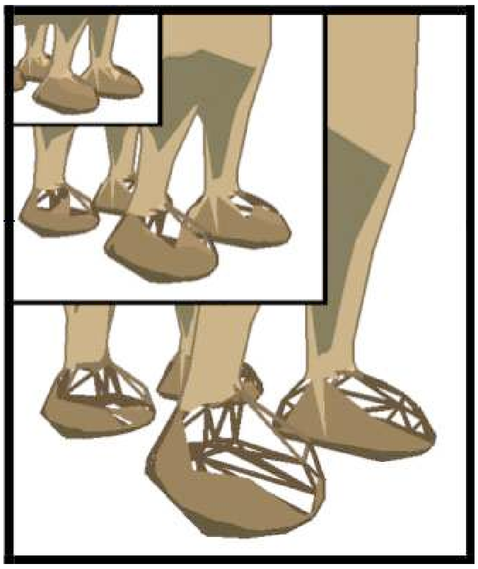


Figure 20: *Example of artifacts of face sorting without perspective correction at different distances to the object.*

in the rendering phase to vary quality and performance. The rules tested by Winnemoller are:

1. All vertices of a triangle have to be front-facing for the triangle to be considered front-facing (This is the most restrictive condition and will render the least triangles. This results in the best performance, but may produce artifacts if the viewer is too close to the object).
2. At least two vertices have to be front-facing (possible artifacts, but fewer).
3. Only one vertex has to be front facing (perfect rendering without artifacts).

[Winnemoller 2002]

The rules can be dynamically chosen at runtime. For instance the rules can be made dependent of the distance from the viewer to the object as well as from the deviation of a vertex normal from its attached triangle normals (this can be calculated when loading the object-model). So the cost for this method is determining the orientation and calculating the lighting for each vertex plus rendering the front- and back-facing triangles (see *equation 9*).

$$\Theta(Ver + Tri) \tag{9}$$

Using this method for face-orientation determination in combination with the rendering process the performance is significantly better than without face-orientation determination. And objects that are far away from the viewer, the level of detail also speeds up the rendering performance.

So the amount of geometric elements has been reduced successfully, but the rendering of specular lighting is still very expensive due to its view dependent nature. To correctly implement specular shading the view-vector has to be calculated for each vertex under consideration and these computations are very expensive. In order to reduce the complexity of the computation the view-vector can be calculated once for every object and then regarded as constant throughout the object. Theoretically this is only true when the viewer is infinitely far away from the object but in practice this heuristic is still pretty good for objects that are relatively far away from the viewer, but it fails when the viewer is near the object. *Figure 20* shows this problem, by rendering the objects from three different distances. From far away (upper left image) the object renders normally but when the proximity from the camera to the object decreases then the holes get bigger.

To correct the problem for objects with holes, we have to look into the problem more thoroughly. The constant view vector $\vec{V}$ is defined in *equation 10*.

$$\vec{V} = C - V \tag{10}$$

C is the center of the bounding box of the object and V is the position of the viewer. We look at the situation where the viewer is close to the object and near the edge of the bounding box. Then the following situation may exist: A surface element S close to the edge of the bounding box has a normal N_s that when compared with the normal of the view-vector N_V appears to point backwards, while actually being a front-facing face. This situation is illustrated in *Figure 21(a)*.

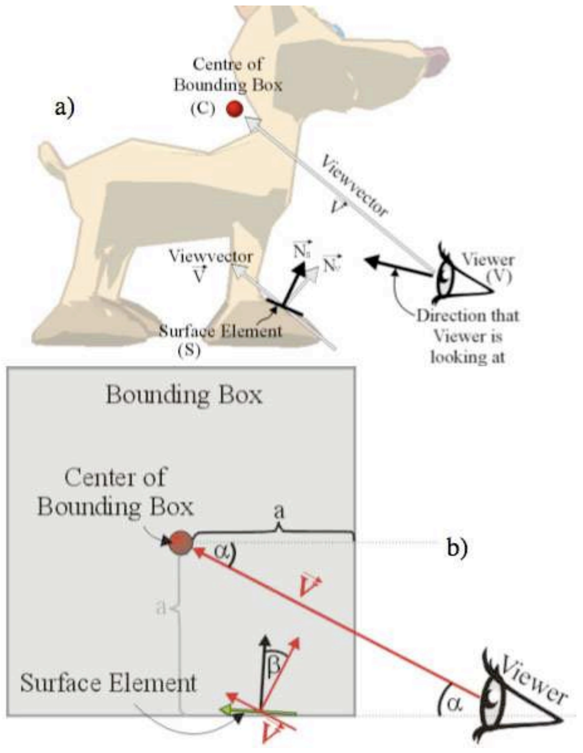


Figure 21: *a) a situation in which object holes arise. b) the angles that have to be considered when correcting the problem.*

In order to find a solution *Figure 21(b)* shows the angles involved. The viewer is placed on the most extreme position, on the bottom edge of the bounding box where the viewer is able to move towards or away from he object and therefore changing the angle α. Then a surface element on the bottom of the bounding box that is almost parallel to the lower edge would lead to a negative evaluation of $\vec{V} \cdot N$ and therefore be considered back-facing even though it is not. Now it can be observed that the angle between the view vector and the view vector β is always less or equal to α. Because of this observation an heuristic can be applied to the face orientation detection. All faces that deviate by at most α from the front facing criterion (the dot product of the surface normal with the view vector is negative) are marked as front-facing. Thus all holes in the rendering can be filled. From *Figure 21* this formula for α can be derived, s given in *equation 11*.

$$\sin\alpha = \frac{a}{|\vec{V}|} \tag{11}$$

V is the view vector and a is the length of the largest edge of the bounding box (the largest edge is taken to get the most conservative measure). α is called *the perspective angle* or *perspective correction angle* because of its usage as a offset in order to counteract the effects of perspective projection.

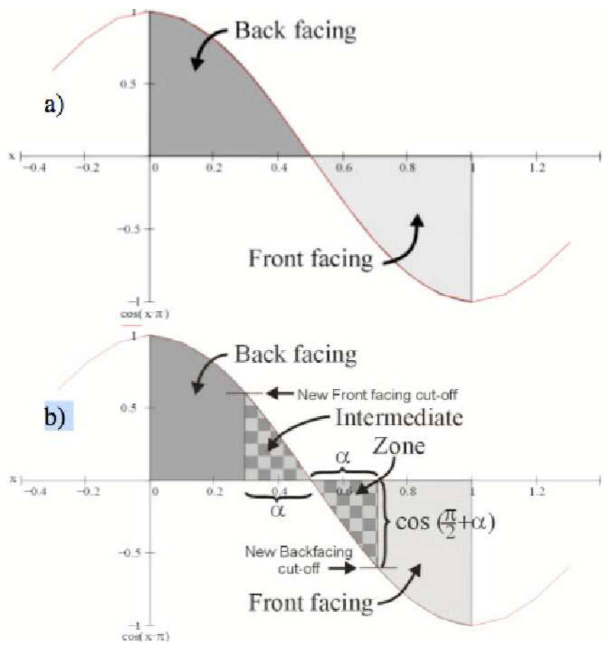


Figure 22: *a) shows the normal approach for determining the orientation of faces ($N \cdot V$) and b) shows the new approach with α as an offset*

The equation for calculation the face orientation $V \cdot N$ is zero when the angle between V and N is $\pi/2$. This means that the two vectors are perpendicular and is used as threshold for the normal face orientation test, but with a constant view vector, a different threshold is needed. $cos(\pi/2-\alpha)$ is perfectly suitable. This results in more faces being marked as front facing (even some back facing) but no holes are rendered any more. *Figure 22* shows the cosine function of the angle between V and N and compares the use of the threshold for the standard and the hole-correcting approach.

To see the benefits of this approach we again take a look at the shading equations. *equation 12* is the phong lighting model and *equation 13* is how the $R \cdot V$ part of the phong lighting model can be computed.

$$I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s) \tag{12}$$

$$R \cdot V = L \cdot V - 2(N \cdot L)(N \cdot V) \tag{13}$$

The $L \cdot V$ part of the equation is not vertex dependent and thus can be computed once for every rendering pass. The second dot product $L \cdot N$ is already computed in the normal diffuse component and therefore is not an additional cost. The last dot product $N \cdot V$ is new, but is also used to determine the face orientation of a vertex. So this approach renders the specular component virtually for free. For large values of the shininess value α heuristics or optimizations should be used to reduce the additional cost for power computations.

Figure 23 shows a comparison of the different lighting approaches on a single object. The leftmost is rendered with only diffuse lighting. The middle one with only specular lighting and the right one with a combination of both. We can see that the specular highlights help to communicate geometry and material properties while still maintaining a cartoonish look.

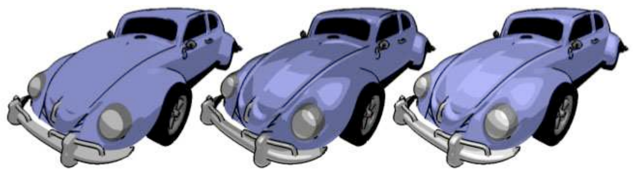


Figure 23: *Examples of a cartoon rendering. From left to right: diffuse only, specular only and combined diffuse and specular*

Figure 24 shows example renderings of a model using a diffuse only lighting component, using a specular only lighting component and a composition of both diffuse and specular lighting. The specular component retains the comic style perfectly and also adds geometry cues.

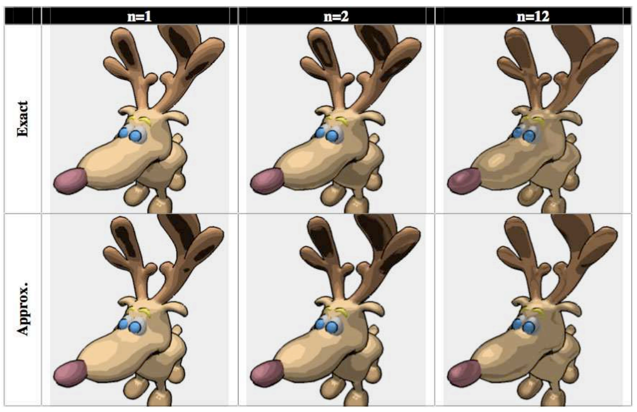


Figure 24: *A comparison of the exact and the approximative rendering approach at different shininess values (here n is α of the lighting equation)*

In a comparison of the performance test the gouraud lighting model is the fastest, followed by standard cartoon style rendering without a specular component and at last the specular and diffuse lighting model, but with larger numbers of triangles per model the rendering process with specular and diffuse lighting even outperforms the standard cartoon style renderer with only diffuse lighting. This can be explained as the benefit from the approximations described before. The rendering of the exact specular lighting is much more expensive and the approximations are very well suited when speed is more important than exactness of the renderings.

3.3 Discussion

In combination the silhouette edge detection and the painter produce a basic cartoon looking rendering and offer many ways to customize the look. But there are still some features missing, like how to render transmissive surfaces like glass or reflective surface (e.g. mirrors). And how can we achieve cartoon looking animations of water, smoke or other particle effects? What are the ways for an artist to customize the algorithms to achieve his specific style and how is it implemented efficiently on modern hardware (mainly using vertex and pixel shaders) to run efficiently for usage in real time applications. All this questions are answered in the next sections, when we describe various applications of cartoon style rendering.

4 Applications

4.1 Pencil drawings

Pencil drawing has a long tradition in art. It is used to draw sketches and has also been established as an independent art style. Characteristic for pencil drawing is the outline and the inner shading with pencil strokes.

Traditional NPR methods do not work with pencil drawings. The reason for this problem is that the silhouette edges in the basic NPR methods appear as a continuous black stroke, whereas in pencil drawing an hand drawn pencil stroke should be simulated. Furthermore the shading of the object is not done with solid colors but with hatching or cross hatching. The interior parts of an object are filled with strokes having a small space to each other. In normal hatching the strokes are drawn parallel and in cross hatching the strokes are drawn so that they are crossing. Another important point is the material of the paper. The typical style of a pencil drawing is a combination of the used paper and pencil. This has to be considered in NPR pencil drawings.

In this part two NPR pencil drawing methods are discussed. The first one is from [Lake et al. 2000] and uses textures to simulate the hatching. The interaction of the pencil with the material of the paper is done with multitexturing. Multitexturing is a technique to combine different layers of textures and is well supported on todays hardware.

A more advanced NPR pencil drawing method is shown by [Lee et al. 2006]. The silhouette edges are detected in image space and randomly perturbed to get a hand drawn looking stroke. Instead of a simple texture to generate the material of a paper a more sophisticated texture mapping procedure is used. The surface of the paper is simulated with normal mapping. Normal mapping is a technique to modify the surface normal to make the surface uneven.

In the NPR pencil drawing method of [Lake et al. 2000] the diffuse lighting component $L*N$ is calculated, where L is the light vector and N is the surface normal. This value is used in the basic NPR cartoon shading method for a lookup in a one dimensional texture to acquire the color of the surface. Instead of that the value of $L*N$ choose a texture with a specific density. As in the cartoon shading process we discretize this selection process. The available hatching textures are splitted into intervals and the value of $L*N$ maps to the texture to avoid a non-linear mapping between the value $L*N$ and the density of the texture. A high value means that the surface receives much amount of light, so a texture with a low density is used. If the light is lower a texture with an higher density is selected.

The texture for the hatching is composed of different stroke types in advance *(see Figure 25a)*. The different strokes are selected randomly and are put with an random space onto the texture. For textures representing a higher density the space of the strokes are reduced. In addition the density effect can be increased by cross hatching that is strokes with an horizontal direction are combined with strokes having a vertical direction *(see Figure 26)*.

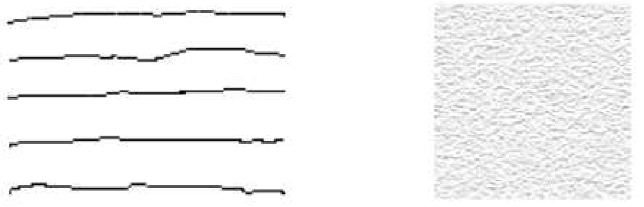

(a) Texture with different strokes. (b) Texture representing the material of the paper.

Figure 25: *The strokes are selected randomly from (a) to compose a hatching texture. The material of the paper (b) is combined with the final hatching texture.*

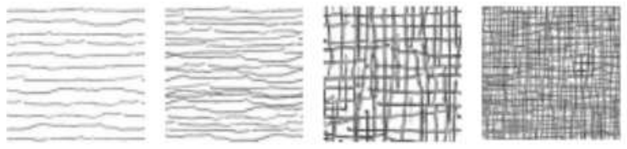


Figure 26: *Texture to simulate the hatching. The level of intensity decreases from left to right.*

As mentioned before the material of the paper is simulated by an separate texture *(see Figure 25b)*. With the multitexturing technique the paper texture and texture representing the hatching are combined together and mapped onto the object to produce a drawing, sketched on paper.

To complete the texture mapping step the coordinates for the texture have to be calculated, because the value $L*N$ selects a texture instead of a single texel. The coordinates can be determined with a projection of the texture through the viewpoint onto the object and the normalized device coordinates are used as the texture coordinates *(see Figure 27)*. For faces with vertices assigned to different texture density a subdivision of the face is needed. This approach works for static objects but is annoying in animated objects, because the texture seems to move on the object.

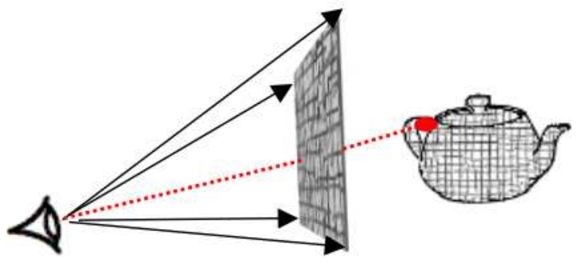


Figure 27: *The texture is projected onto the object to generate the texture coordinates.*

Silhouette edges are detected in object space and with the additional geometry information, more stylistic lines can be generated as described in the chapter about silhouette edge detection. In *Figure 28* shows the final image composed of the textures (simulating the hatching) and the black silhouette edges.

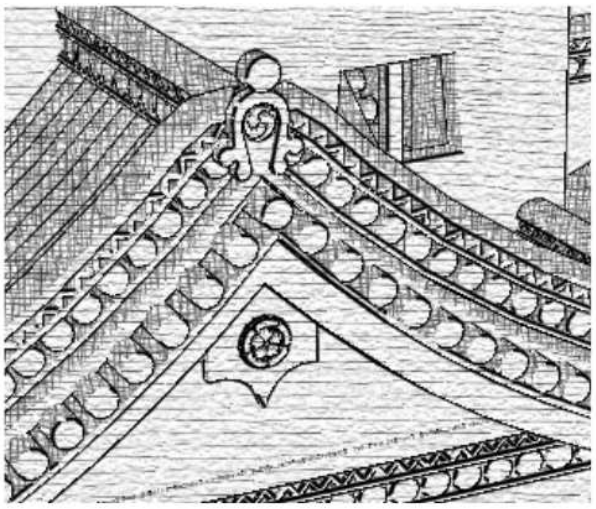


Figure 28: *Example of an NPR generated pencil drawing generated with the method from Lake in the work [Lake et al. 2000].*

In the paper of [Lee et al. 2006] a more sophisticated NPR pencil rendering method is introduced. The silhouette edges are detected in image space and are drawn slightly distorted to imitate an hand drawn style. For the interior shading the pixel intensity is calculated and the contrast of the pixel brightness is adjusted to achieve pencil drawing tones. The two steps are combined with a pixel shader to produce the final image.

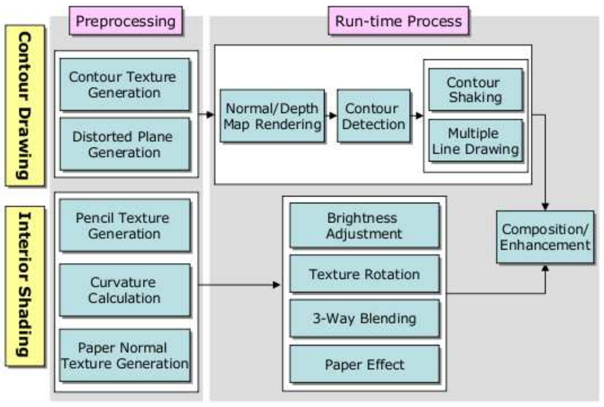


Figure 29: *Rendering Pipeline of the method from [Lee et al. 2006].*

In the following section the algorithm from [Lee et al. 2006] is described in more detail. The silhouette edge detection is done with the depth-buffer and the normal-map and results in an black and white or gray scale image. Afterwards the lighting effect for the contours are calculated in a pixel shader with *equation 14*.

$$I'_c = l_c * I_c, \tag{14}$$

where l_c is the light effect factor, I_c the original pixel value and I'_c is the modified pixel value of the contour.

The generated silhouette edges are straight lines and do not look hand drawn. A characteristic of handdrawn lines is that they consists of small errors. These errors are in an limited range, because the artist will remove noticeable errors during the drawing process. For this reason the contour shaking can be approximated by the sine function, as shown in *equation 15*.

$$y = a * sin(bx + c) + r, \tag{15}$$

where a and b determine the range and period of the error, c is a shift of a periodic function and r adds randomness to the shaking value y. The generation of random contours is hard to implement with a pixel shader, where writing is only possible to the current pixel. In the preprocessing step the screen space is divided into regularly sized rectangles and the coordinates on the contour image are assigned to the texture coordinates of the rectangle. So the contour image is redrawn by texturing the rectangles with the contour image *(see Figure 30)*. The contour shaking is done by adding the approximated value y to the texture coordinates.

Another characteristic of hand drawn lines is that they are composed of multiple overlapping lines. This effect can be achieved with multi-texturing of several distorted textures. Furthermore the intensity of a pixel is set to a darker value in the pixel shader for contours with much overlapping to simulate a darkening effect of the pencil strokes *(see Figure 31)*.

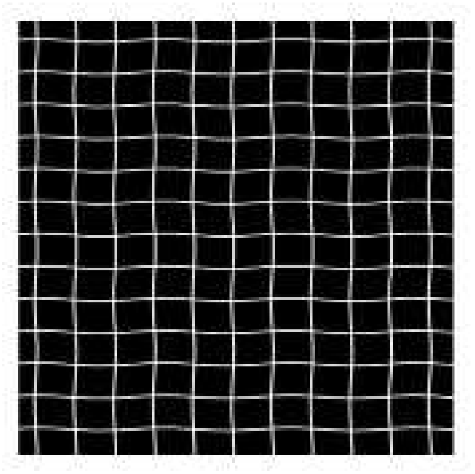


Figure 30: *Texture with randomly generated contours.*

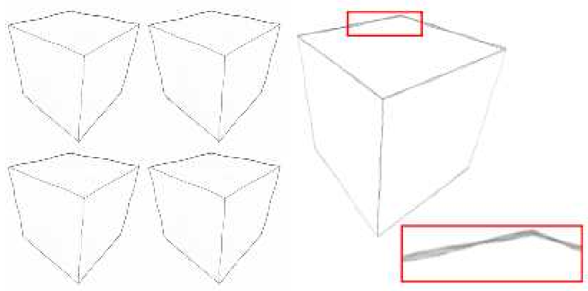


Figure 31: *Several cubes with shaked contours are combined to simulate an hand drawing style.*

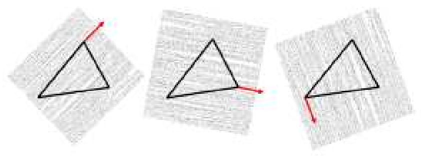
The brightness of the interior shading depends on the density of the strokes, which are placed uniformly in the texture image. The brightness of a pixel is adjusted by drawing the strokes with a perturbation on the top of the current texture. With a lot overlapping strokes the brightness of the texture get darker but as in real pencil drawing the increase of brightness becomes smaller. This is simulated by *equation 16*.

$$c_t' = c_t - \alpha_b * c_a; c_a = c_t * (1.0 - c_s), \tag{16}$$

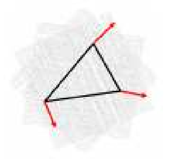
where c_t' is the updated texture color, c_t is the current texture color, c_s is the stroke color and c_a is the maximum increase of darkness by this stroke and α_b is a user controlled parameter. The different brightness levels are stored in a 3D texture starting with a pure white image for the highest brightness level and are calculated in advance. The decision parameter for the brightness level of the texture is determined by a general shading technique, like Goraud Shading.

In pencil rendering the intensity variation in a bright region is more important than the contrast between bright and dark regions so this decision parameter is adjusted with a square root function in the pixel shader. In a hand made pencil drawing the direction of the interior strokes depend on the curvature of the surface.

For each vertex of the face the curvature direction is precomputed and the texture is rotated according to the calculated value. Usually a face is represented by a triangle and therefore consists of three vertices *(see Figure 32a)*. Because of the texture rotation technique three different aligned textures are mapped onto the every face. The interior strokes for surfaces with complicated shapes consists of differently orientated pencil strokes *(see Figure 32b)*. Since there are already three textures per face this effect can be achieved by blending this textures per multi-texturing, as well the discontinuities of the neighboring faces are eliminated.



(a) Different rotated textures.



(b) Combined textures.

Figure 32: *Different rotated texture for each vertex of the face. These are combined to simulate the interior shading of pencil drawing.*

The material of the paper, such as roughness, is simulated with a separate texture. The surface structure of the paper is stored in an height map and a normal map is precomputed of it. The interaction of the paper and the pencil stroke is calculated by the function, as shown in *equation 17*.

$$c_t' = c_t + \alpha_p * (d \cdot n), \tag{17}$$

where c_t' is the modified texture color, c_t is the original texture color, α_p is a weighting factor, d is the curvature direction and n is the normal direction *(see Figure 33)*. If the direction of the stroke and the normal vector of the paper surface are similar than the texture color is brighter, otherwise it is darker.

The composition of the final image is done in an pixel shader. First the silhouette edges are generated and applied to the final pixels. Than the interior shading is added and finally a paper texture is blended onto the background, because the color of the paper is not completely white *(see Figure 34)*.

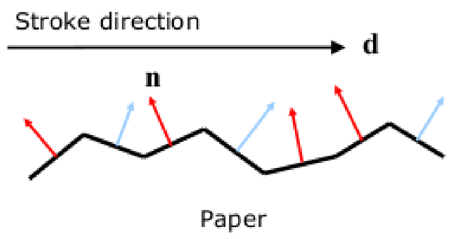


Figure 33: *Interaction of the paper material and the pencil stroke.*

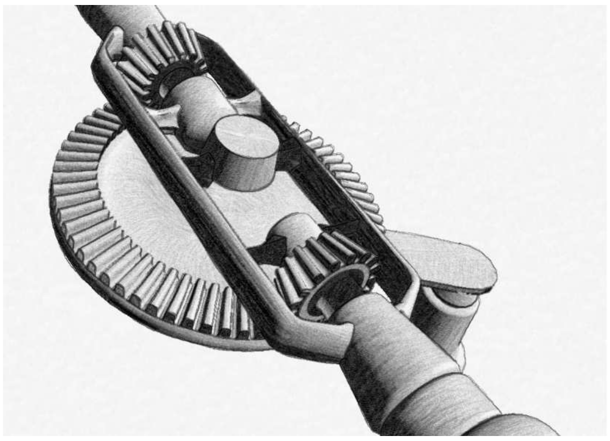


Figure 34: *Composition of the final image.*

4.2 Chinese Painting

Chinese Painting is famous for its freehand brushwork and has evolved as a school of its own in the world animation circle. In the traditional chinese painting process the ink is made of a black or colored pine soot. A brush pen is used to apply the ink onto a specific paper, called Xuan paper. The irregular dispersion of the ink onto the Xuan paper makes the generation of chinese painting time consuming. In contrast to western cartoon generation, where the artist is assisted by advanced techniques during the process of creation, little research have been done in this area for chinese painting.

[Yuan et al. 2007] present a NPR method including the interior shading, the silhouette detection and the background shading for the simulation of a chinese painting. The aim of this method is that artists are able to efficiently create chinese style paintings and animations for movies or games. Another demand of the presented method is the rendering efficiency, which is mostly done on the GPU with vertex and pixel shaders to achieve the framerates needed for real-time applications.

Instead of physically model the brushes and paper, the appearance is improved by pre-processing of textures and smoothing filters. This method works for static and dynamic objects and only a triangular surface mesh is required as input. The rendering process at runtime consists of three passes. In the first pass the interior shading is determined, in the second pass the silhouette edges are extracted and in the third pass the results of the previous two passes are combined. *Figure 35* shows the stages of the rendering pipeline of the NPR chinese painting process.

The general steps in the rendering process at runtime are [Yuan et al. 2007]:

1. Initialize two 2D textures as rendering targets.
   - In vertex shader, compute the diffuse color as the texture coordinate delivering to pixel shader.
   - In pixel shader, map the shading texture using the incoming coordinates to output the color.
   - Render not to the screen, but to the first rendering target texture.
   - If Silhouette enhancing is required, add another silhouette-extracting operation in step 3 but rendering to the target texture in this pass.
2. Draw the mesh to render the silhouette to the other rendering target texture.
   - In vertex shader, compute the texture coordinate delivering to pixel shader.
   - In pixel shader, map the edge texture using the incoming coordinates to output the color.
   - Render not to screen, but to the second rendering target texture.
3. Draw a rectangle with the same dimension as the window, and then determine the display color in pixel shader.
   - Calculate the smoothed color by box filtering on the silhouette in the second rendering target texture.
   - Multiply it with the color in the first rendering texture and background texture.
   - Render to screen.

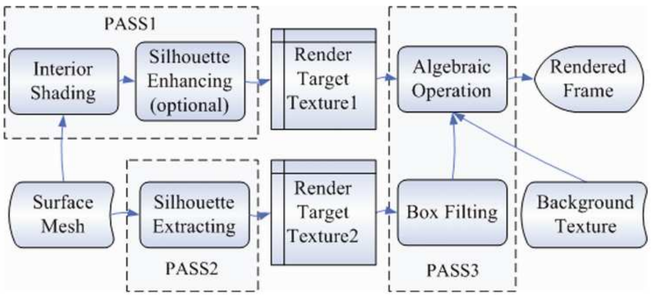


Figure 35: *Three passes in the rendering pipeline.*

As in the most NPR methods the interior shading depends on the diffuse lighting component. The diffuse lighting value corresponds to a color value in a pre-computed shading texture. A point light source is defined for every object, which is static relative to the object. In a vertex shader the diffuse color value can be computed with *equation 18*.

$$C = N \cdot L, \tag{18}$$

where C is the diffuse color value, N is the surface normal and L the vector of the light direction. The range of this continuous value lies between 0 and 1 and is quantized afterwards. There are five levels of brightness and the function for the quantization is described in *equation 19*.

$$C_0 = \begin{cases} 0.1 & 0.8 < C_i \\ 0.4 & 0.55 < C_i \leq 0.8 \\ 0.7 & 0.25 < C_i \leq 0.55 \\ 1.0 & C_i \leq 0.25 \end{cases} \tag{19}$$

where C_i is the diffuse color of a point and C_0 is the quantized color. The conclusion of *equation 19* is that for a dark diffuse color C_i the rendered color C_0 is brighter.

This step function is implemented with a 1D texture, *see Figure 36*, and bilinear filtering is used to smooth the sharp transition. The irregular dispersion of the ink onto the Xuan paper is simulated by blurring the shading texture with a Gaussian function *equation 20*.

$$G(x) = \frac{1}{2\pi} e^{-\frac{x^2}{2}} \tag{20}$$

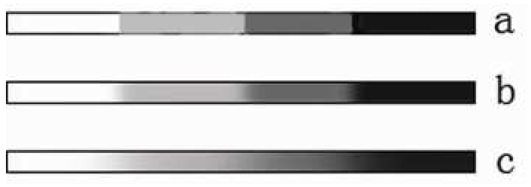


Figure 36: *(a) Original 1D shading texture, (b) Bilinear filtering, (c) Blurred with the Gaussian function*

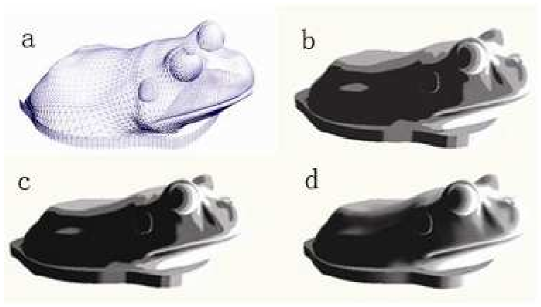


Figure 37: *(a) unshaded mesh, (b) shaded with texture from Figure 3.a, (c) shaded with texture from Figure 3.b, (d) shaded with texture from Figure 3.c*

The outline of an object in chinese painting is usually irregular due to the dispersions of the ink. Yuan et. al. extract the silhouette by texture mapping and smooth the silhouette edges by a box filter [Yuan et al. 2007].

Geometrically a silhouette edge for a free form surface exists if the surface normal is perpendicular to the vector from the viewpoint. A point p lies on a silhouette edge if the *equation 21* is satisfied.

$$V \cdot N = 0, \tag{21}$$

where V is the vector from the viewpoint and N is the vector of the surface normal. Since this condition is to strict a threshold value is introduced, as shown in *equation 22*.

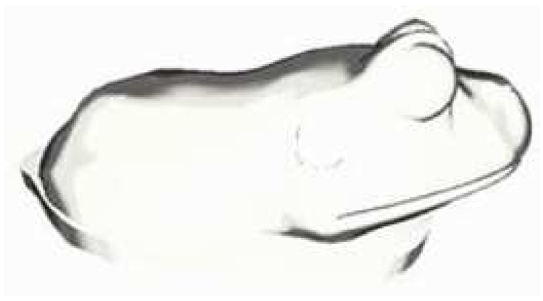
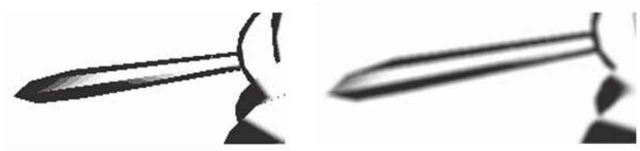


Figure 38: *Object rendered only with the detected silhouette edges.*

$$edge = V \cdot N, \tag{22}$$

if *edge* is greater than or equal to zero and is less than or equal to the *threshold* than p lies on a silhouette edge. The calculation of the value *edge* can be done fast in a vertex shader on the GPU. For performance reasons the threshold value can be stored in a 1D texture which is fully supported by todays graphics hardware. At the left part of the 1D texture the color represents the silhouette edges (usually black) and on the right part of the texture it is filled with white color. To avoid the aliasing of the silhouette edges multi-sampling is used. Moreover to eliminate unwanted broken lines a smooth filter operation is applied to the result p of the silhouette detection process. A Gaussian filter is used to smooth the silhouette edges and is calculated in advance *(see Figure 39)*.



(a) Aliased silhouette edge. (b) Antialiasid silhouette edge.

Figure 39: *The aliased image (a) is blurred with an Gaussian filter (b).*

Finally the interior shading and the silhouette edges are merged and combined with a background texture, which represents the characteristics of the Xuan paper used in traditional chinese paintings *see Figure 40)*.

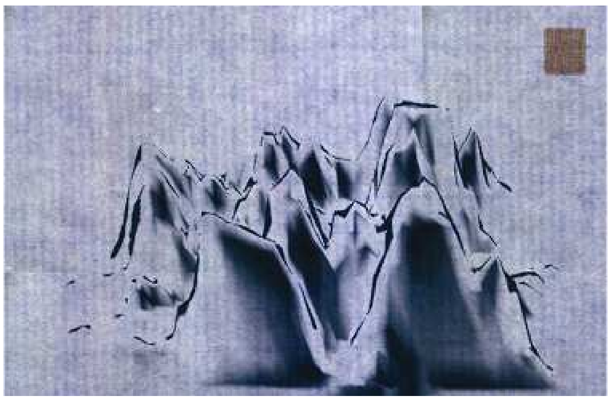


Figure 40: *Final chinese painting picture on a background texture, imitating the Xuan paper.*

In [Yuan et al. 2007] a animation framework is discussed in which a set of mesh models is ordered by key-frames and linearly interpolated the vertices between them *(see Figure 41)*. Animation in a chinese painting cartoon gain dynamic effects by softly waving parts of the character. This effect is simulated by the morphing technique as described before, the vertex position between two frames is linearly interpolated. A constraint for morphing is that the amount and order of the objects vertices stay constant during the animation. The key-frame animation is done in a vertex shader separately for every animated object and the algorithm from [Yuan et al. 2007] looks as follow:

At time t, for a character:

1. Find out the nearest two key frames with time t_0, t_1, where $t_0 \leq t \leq t_1$;
2. Compute $Scalar = (t - t_0)/(t_1 - t_0)$ for interpolating, and deliver $Scalar$ into vertex shader with two meshes.

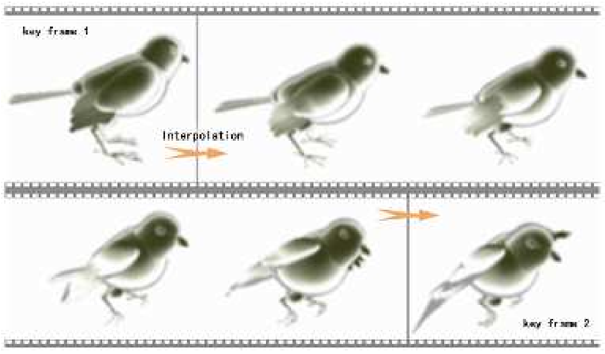


Figure 41: *Two key frames and the linear interpolated frames in between.*

The algorithm works well with most animations in chinese painting cartoons, such as animals and plants, but do not fit for animations with water or clouds. With this NPR chinese painting method the artists can concentrate on other important things than the tedious painting painting process, such as the scenarios or stories.

4.2.1 Cartoon Styles

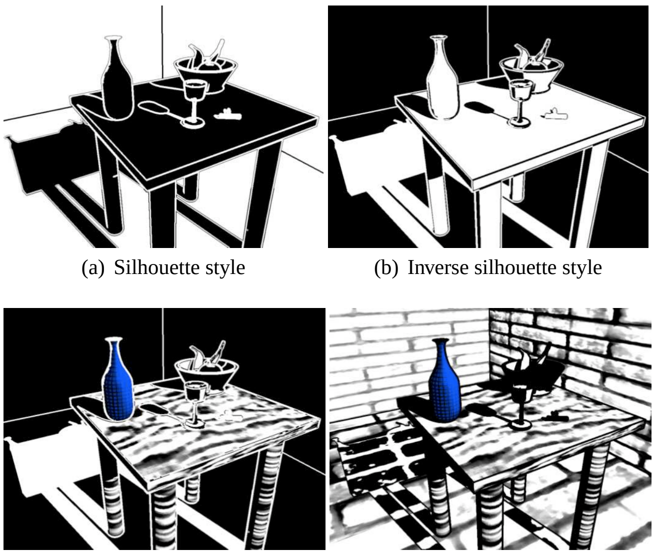
Cartoons or comics as we know them today evolved from comic or sequential art strips in newspapers and magazines at the end of the 19th century. Since then, their styles has changed remarkably and different and independent styles have been created in different regions. In the USA the western comic (e.g. "Superman"), in Europe for example frankobeligan comics (e.g. "The advantures of Tintin") and in Japan manga comics (e.g. "Akira") evolved.

All this comics have completely different styles, but they have also some characteristics in common which makes them recognizable as comics respectively cartoons. For example the usage of silhouette edge lines (although they are sometimes used differently), the abstraction from real world, sometimes the use of discrete colors, the usage of special effects like speed lines, certain camera techniques and perspectives, the usage from text in the artwork and unrealistic physical properties e.g. for shadows. Modern cartoon or comic artists like Frank Miller for example use many styles and mix them to create there very own style. The methods are not only chosen by artistic considerations, but also to help telling the story and underlining the plot.

Spindler et al. observed a lack of research in computer graphics of how to create such different styles for computer rendering [Spindler et al. 2006]. They use the algorithms mentioned in the Techniques section and extend them to create four different cartoon styles for a real-time game engine, namely those styles used by Frank Miller in ”Sin City” and those used McFarlane’s in ”Spawn”.

Frank Miller’s ”Sin City” style

The ”Sin City” comics by Frank Miller have a distinct ”pen and ink” look, but he does not use only one style, he uses an unconventional combination of hatching, stippling, large black faces, and monochromatic silhouettes seeking the contrast to the background. His style seems very dark and dusky throughout his comics and he sometimes uses colored objects or characters to emphasize its importance in the storyline.


(a) Silhouette style (b) Inverse silhouette style

(c) Foreground shaded only (d) Foreground and background shaded

Figure 42: *Renderings of the four different styles in Frank Miller’s ”Sin City” comics*

Spindler et al. analyzed the ”Sin City” style and identified four different plot-dependend cases [Spindler et al. 2006]:

- silhouette style: background white and foreground black
- inverse silhouette style: background black and foreground white
- foreground shaded only: background is black
- foreground and background shaded

In *figure 42* different styles are simulated with the rendering engine from [Spindler et al. 2006]. In addition to these styles they found two very distinct characteristic style elements, namely double contour lines and stylistic shadows, as shown in *figure 43* and *figure 44*

The stylistic shadows are very complex, there are two different cases, when the shadow falls on an object that is in the foreground it is drawn black, but if the shadow falls on a background object, the color of the object is inverted where the shadow falls so that the structure of the background-object is still recognizable. For the

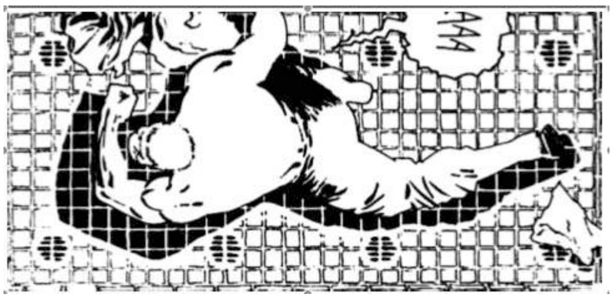


Figure 43: *Example for stylistic shadows in Frank Miller’s Sin City*

implementation of stylistic shadows in a game engine, objects have to be classified into foreground and background objects and then be treated differently.

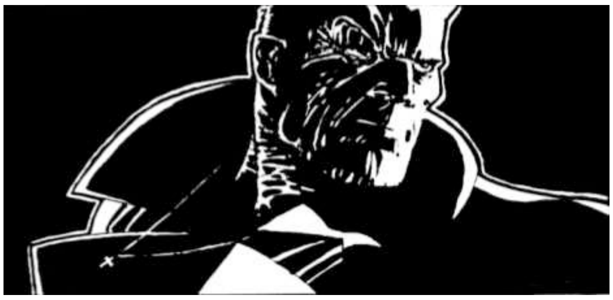


Figure 44: *Example for double contour lines in Frank Miller’s Sin City*

In comparison to other comics foreground objects the ”Sin City” style use a rather thick contour line, which is also often drawn doubly *(see figure 44)*. For use in a game engine they are simplified as thick white lines surrounded by very thin black outlines. As for the stylistic shadows ,foreground objects have to be rendered differently than background objects.

The rendering pipeline for the ”Sin City” style rendering consists of three passes. A projection of the scene is computed including color, normals depths and object information. This pass is used to determine foreground regions and shadow areas. In the second pass the information from the first pass is used for image space computation of the silhouette and crease edges. In the third pass the shadowed areas of the background are inverted and the shadowed areas of the foreground are draw as solid black surfaces. Then the contour lines are thickened and drawn with black outlines (the method for doing this is described more thoroughly in the Techniques section). Afterwards the computed images are combined. *Figure 45* shows these three rendering passes.

Note that the rendering pipeline mentioned before is the one used by Spindler et al. and is in no way mandatory, it uses image space methods for silhouette detection, which may in some cases not be useful, especially if modern hardware is available object space methods can be computed with the support of vertex and pixel shaders and may be faster than this approach [Spindler et al. 2006]. Nonetheless this method successfully reaches its goal and showed how to enhance the current NPR techniques to render a different style.

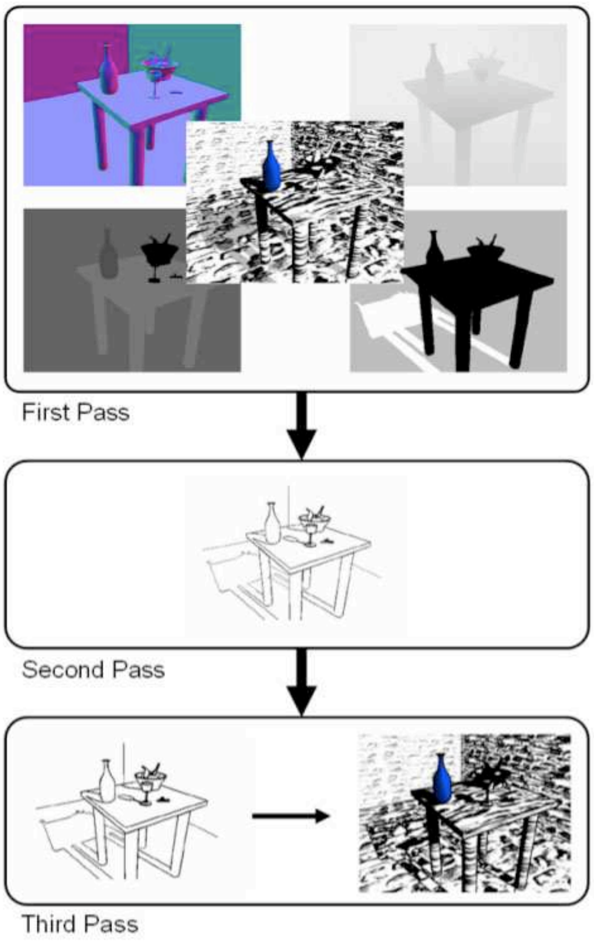


Figure 45: *Rendering passes used for "Sin City" style rendering*

Todd McFarlane's "Spawn" style

The "Spawn" style from Todd McFarlane uses many different colors and even color gradients. Many feature edges are inked, emphasizing on small details. This results in more realistic look but is still recognizable as a cartoon.

For the implementation of this style Spindler et al. use the methods for painting the cels described in the techniques section by Lake et al. and extend it [Spindler et al. 2006; Lake et al. 2000]. This method uses a 1D texture to map the colors onto the object. The cosine of the angle between the normal of the surface and the vector to the light source is computed ($\overline{L} * \overline{n}$). This value is used to determine which texel of the 1D texture is used. The intervals of $\overline{L} * \overline{n}$ where each texel is used are uniformly distributed between 0 and 1.

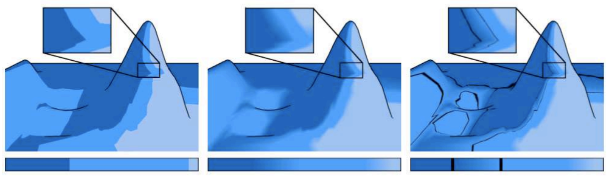


Figure 46: *Enhancement of the cel shading (left) technique: soft cel shading (middle) and pseudo edges (right).*

By increasing the resolution of the 1D texture a combination of solid colors and gradients can be achieved. This method is called soft cel shading. With this approach a designer can decide where to use color gradients and where to use a hard border between two colors. *Figure 46* shows the effect of the enhancement of the hard shading algorithm to soft shading and feature edges.

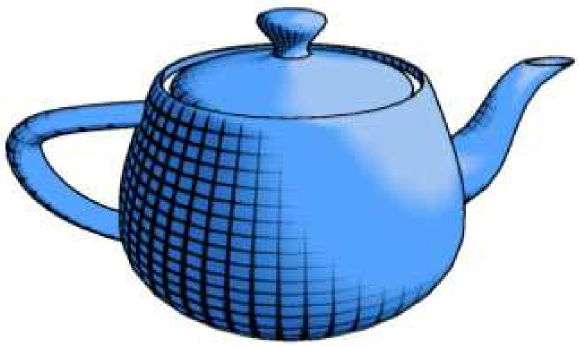


Figure 47: *Cross-hatching using pseudo edges*

To add another specific feature from the spawn style, namely feature edges between solid color and gradients the same method is used. Small black intervals between the solid color and the gradient are added to the 1D texture. This method is called pseudo edges. The same technique can be used to simulate cross hatching. *Figure 47* shows how the simulation of cross hatching looks on tea pot.

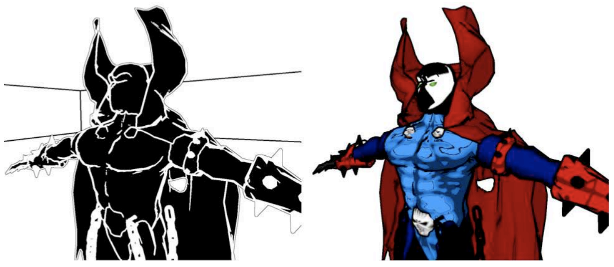


Figure 48: *Renderings of the same character in Sin City style (left) and in Spawn style (right)*

Figure 48 shows a character rendered with the to new styles and compares them. We can see that with only little enhancements to the basic algorithms completely new styles can be created.

Mike Mignola's "Hellboy" style

Mike Mignola uses high contrast lighting, angular design and very textured line work in the drawings in his Hellboy comic series. The minor details in the "Hellboy" style, like the ragged line on Hellboy's arm or leg, is problematic to mimic with traditional NPR methods *(see Figure 49)*. Highly detailed texture only works for certain angles and representing the details geometrically, increase the complexity of the object to much.

Brown explains a method to simulate the Hellboy style of Mike Mignola [Brown 2007]. The following four techniques are applied to a black and white image to provide the impression of an un-inked illustration of this artist:

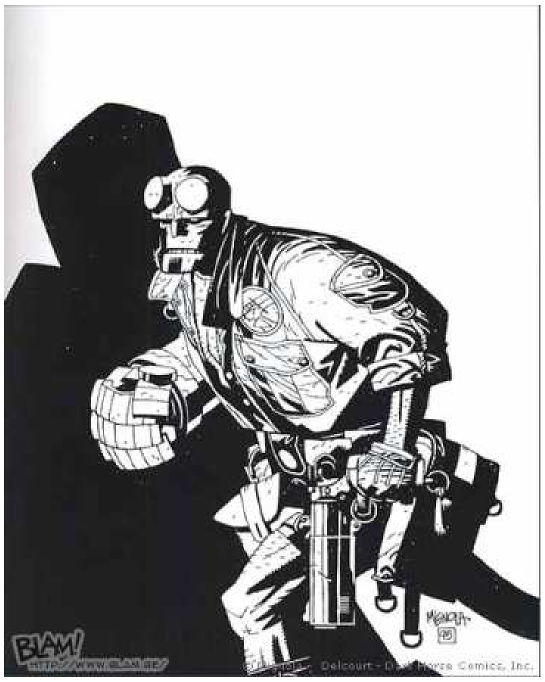


Figure 49: *A image from Mike Mignolas Hellboy comic.*

1. Outline the model with the finning technique described in [McGuire and Hughes 2004]
2. High contrast lighting to imitate the inky shadows and a the usage of texture to simulate the interior shading with a marker.
3. Graftals are used to mimic the otherwise invisible surface details, like pock marks or bumps.
4. Textured outlines and graftals are used to create the complex division between lit and shaded areas.

As usually for silhouette edge detection or outlining in object space the vector dot product is evaluated for the normal vectors of the faces adjacent to an edge. This general method is GPU-accelerated by using Vertex Buffer Objects described by [McGuire and Hughes 2004], where 6 values have to stored for every edge. This 6 values are the 4 vertex position defining the two triangles and the normal vectors on the vertices defining the edge *(see Figure 50)*. The surface normal vector can be calculated in the vertex shader with the other passed 6 values.

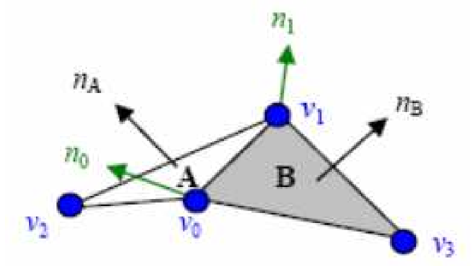


Figure 50: *Encoded edge information.* $v0$, $v1$, $v2$, $v3$ *define the faces.* $n0$ *and* $n1$ *are the normals on the endpoint of the edge. The surface normals nA and nB can be calculated in the vertex shader.*

The edge information is determined by iterating over the faces of the object and are stored in an single vertex structure, so that they can be used with Vertex Buffer Objects (VBO). The edge information is encoded into the Position, Color and MultiTexCoord0-3 and is denoted by the name ”Edge Vertex Object”.

Since with a vertex shader or fragment shader, additional geometry can not be created on the GPU, the Edge Vertex Object has to be duplicated four times to define an edge quad. Maybe with the possibility of the newly programmable geometry stage with a shader this additional expense can be avoided in the future. The vertices are transformed by this attributes in the vertex shader and the stroke information and appearance are mapped with a texture to define stylized silhouette edges.

The black and white interior shading is simulated with a modified shading algorithm where the intensity of the light is adjusted. If the intensity is below a specified value (a), the pixel color is set to black, otherwise the *equation 23* is used to provide a subtler gradient.

$$Color = ((i*0.5)*(i*0.5)+0.75)*White, \tag{23}$$

where i is the intensity. The usage of textures to draw black shadows works for static objects but fails for animations, because of the missing information on surface orientation.

The small marks used for the surface details follow the 2D orientation of the surface so graftals must be generated according to the surface geometry. Unfortunately generating the graftals parallel to the screen with one edge set to an edge on the mesh failed, because of self intersection problems. A time consuming self intersection prevention has to be done to produce randomly placed graftals.

The self intersection problem also exists in the shadow outlining process, but is more manageable with textured strokes. An edge, adjacent to face one and face two, is a shadow-bounding if the intensity of face one is greater than the value a from the clamping of the interior shading and the face two is smaller than this value. The intensity is calculated with the usual equation of Lambert, as shown in *equation 24*.

$$Intensity = L \cdot N, \tag{24}$$

where L is the light direction and N is the normal of the face. Due to the fact that the stroke-generation is done in an vertex shader it is not possible to get shadow-boundaries smoother than the edges defining the face. This leads to a zigzagging stroke which can not be avoided without any interpolation across the triangles, defining the faces.

Besides the mentioned problems the NPR method represented by [Brown 2007] produces a comparable style to Mike Mignolas Hellboy comic style *(see Figure 51)*

4.3 Effects

4.3.1 Liquid Animations

Liquid animations in photorealistic computer graphics exist for a long time, but how can these techniques be used to create visually appealing cartoon style renderings? Prior art exists of cartoon style rendering of smoke but these methods can not be adapted for liquid animations because liquid renderings mainly focus on the surface, which is not existing in smoke effects.

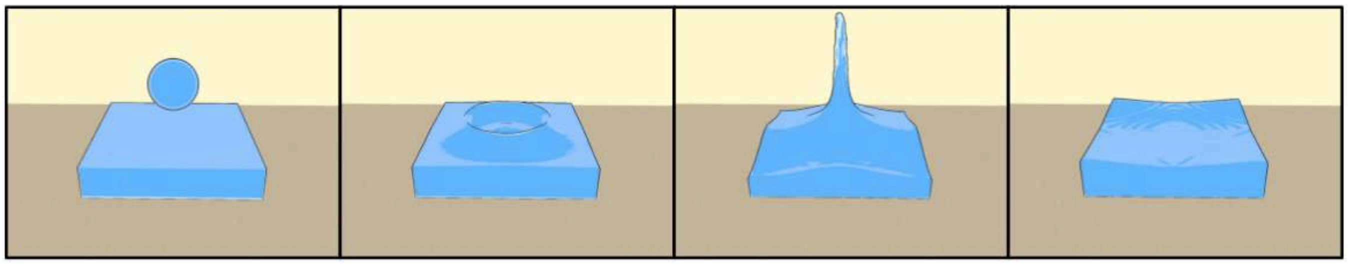


Figure 52: *A drop of water falling into a pool of liquid. The near silhouette and thinness coloring effects as well as the bold outline can be seen.*

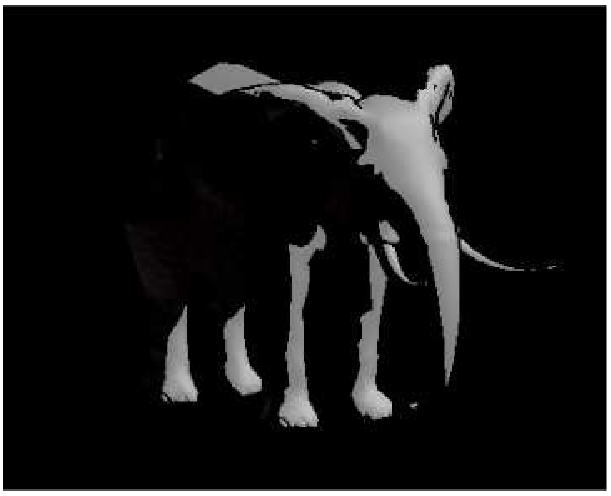


Figure 51: *Image rendered with the NPR method to mimic the Hellboy style of Mike Mignola.*

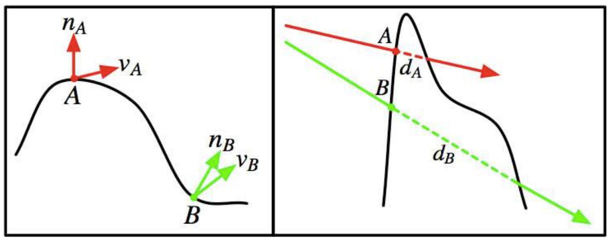


Figure 53: *The near silhouette and thinness criteria. Left: Near silhouette: when the angle between the normal and the vector to the viewpoint exceeds a certain threshold a point (A) is identified as a near silhouette region. Right: Thinness: When the distance between the entry and exit point from a ray between the viewpoint and a point on the surface is less than a threshold (A) the point is colored in the color of thin regions.*

In [Eden et al. 2007] a method for rendering such liquid animations in cartoon style is shown that uses a liquid simulator that computes physically correct liquid motion. The simulator generates a mesh that represents the surface of the liquid. It is not required that the surface is represented as a mesh, the only constraint is that a normal of every point must be computable. To achieve a cartoon looking animation like it is seen in classic comics, large regions of constant color are needed. The presented implementation only uses three different blue tones and black. The black color is used for the silhouette lines and the different blue shades are used for the normal water, for near silhouette edges and for thin water.

Near silhouette areas can be identified with the same approach we used for painting the cels in different colors, but this areas are viewpoint dependent and therefore we use the vector to the viewpoint instead to the light source. When the term $n * v$ (n is the surface normal, v the vector to the viewpoint) is smaller than a user-defined threshold t than it is a near silhouette area, otherwise not. The mapping of colors is done as described in *equation 25*.

$$C = \begin{cases} C_{near\_silhouette} & \text{if } n * v < t \\ C_{body} & \text{else} \end{cases} \tag{25}$$

The shape of the near silhouette areas helps the viewer to perceive the geometry and motion of the liquid.

Another view-dependent cue is highlighting the areas where the liquid is thin. This can give the impression of foam and transparency at thin regions. Here thinness is defined as the distance between the entry and exit point of a ray from the camera. If the distance is smaller than an user-defined threshold the region is highlighted.

Outlines are detected by performing an edge detection on the depth map generated during the rendering pass. This image-space method is described more thoroughly in the Techniques section. The detected outlines and depth discontinuities are drawn with bold black lines.

All three methods are user-controllable. By adjusting the threshold the user can customize the first two effects and thereby altering the painting of the water surface. The latter method can by adjusted to draw more detail.

Motion is not well perceived if there is little change in the geometry of the liquid. To help convey motion many cartoon animations add objects or shapes that appear attached to the surface. To do this with the liquid simulator points have to be tracked on the surface and a texture is mapped on the points. But not only the position of the point is necessary but also their orientation, so that the texture can be mapped properly. This oriented points are tracked by a set of unoriented points. This set of points is required to move rigidly so that it can be assumed that the orientation can be computed. By backtracking the position of the set of point the orientation of point can be calculated.

The combination of these four techniques result in a cartoon looking animation of liquids with a good abstraction of details while still having enough information available in the picture so that the viewer perceives the motion and geometry of it. But a downside of this method is that in most cartoons, like everything else, the liquid animations are normally not physically correct. Liquids in cartoon behave somewhat differently and serve more an artistic purpose than to be physically correct. Artistically behaved liquid simulation would be a field for future research. But nonetheless the output of this rendering can be used for prototyping cartoons or can be used as a basis for the artist that would afterwards enhance the renderings for use in an animation.

Figure 52 shows these techniques in action. A drop of water falls into a pool. In the second as well as in the third and fourth picture we can see the effect created by the usage of different colors in near silhouette areas. And in the third picture a we see the effect of thin water as well.

4.3.2 Animation of Cartoon Smoke and Clouds

Drawings and animations of smoke are often used in cartoons. For real time rendering in cartoon style they pose many problems. The structure of smoke and clouds is normally very complex and a mesh with a very high polygon count would be needed, when traditional algorithms for cartoon style rendering would be used to render the smoke animation. This would make the render process very slow and especially rendering with shadowing and self-shadowing would not be possible in real time.

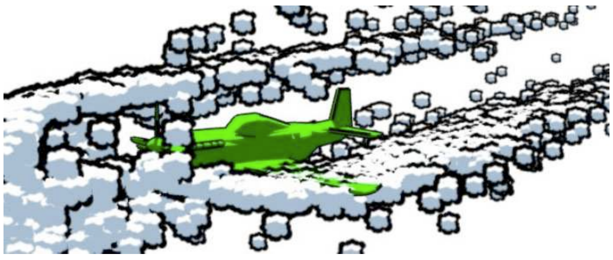


Figure 54: *Clouds deflected by collisions with an airplane and interacting with the forces from the propellor.*

McGuire and Fein introduced a technique for rendering real time smoke animations in cartoon style that overcomes the mentioned problems, by using billboards for rendering the particles [McGuire and Fein 2006]. This technique relies, like the rendering of water animations, on a fluid simulator. Individual smoke molecules have little mass or volume therefore smoke simulation is actually fluid simulation of the surrounding medium. Many smoke molecules are combined to one particle, so that every particle represents a set of spherically arranged smoke molecules.

To approximate the behavior of smoke with a liquid simulator it is represented as a compressible fluid on which six environmental forces can act upon, namely vortex, buoyancy, gravity, wind, turbulence, and drag. The artist using the simulator can configure the values of these six forces to adjust the behavior of the smoke. Collisions with the rigid bodies can also be calculated by the simulator *(see Figure 54)*, but collisions between particles are not calculated because they do not constrain each other and it would be too expensive to compute. So motion is limited to the interaction with the environment. Smoke disappears over time due to diffusion, so the particles density is reduced over time. This makes them shrink and when they disappear they have a zero size, which makes the simulation more realistic.

For rendering the particles, billboards are textured parallel to the image plane. The size of the billboard is proportional to the density of the smoke, so that the particle disappears unnoticed over time. To provide variety, four different smoke puff variations are assigned to each particle at creation time.

The rendering process uses for values four each pixel of the billboard texture: a unit camera-space surface normal N_{xyz}, signed camera space depth d, diffuse color C_{rgb}, and coverage mask α. To speed up the computation time these values are packed into the eight 8-bit channels of two RGBA texture maps, so only two 32-bit memory operations are needed per pixel. The surface normal and the camera space depth are packed into one RGBA-texture using integers as in *equation 26*.

$$(r,g,b,a) = \left(\frac{N_x+1}{2}, \frac{N_y+1}{2}, \frac{N_y+1}{2}, d+0.5\right) \tag{26}$$

The diffuse color C_{rgb} and the coverage mask are packed into the second texture map as in *equation 27*.

$$(r,g,b,a) = \left(C_r, C_g, C_b, \alpha\right) \tag{27}$$

Figure 55 shows the texture maps of four smoke puff variations. These are created as a preprocessing stage. In order to create these texture maps 3D meshes are rasterized and colored by the surface normal, depth, and unshaded diffuse color as appropriate for N, d, and C. The α map is created by rendering white puffs on a black background and dilating the resulting shape.

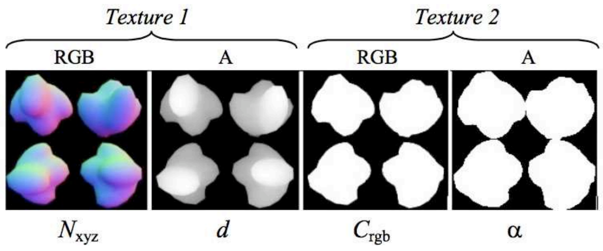


Figure 55: *The two texture maps representing the billboards including surface normals, depth, color, and alpha maps.*

The rendering is done with a pixel shader. The pixel shader reads the values of the texture maps and converts them into floating point numbers and then computes the camera space depth and pixel colors.

For ambient light color K_a, light color K_L, camera-space direction L to the light source and diffuse color C from the texture map, the pixel at (x,y) is colored by a pixel shader using Lambertian shading as in *equation 28 and equation 29*.

$$pixel_{rgb} = K_a(x,y) + C * K_L * q(max(0, N * L)) \tag{28}$$
$$pixel_{\alpha} = \alpha \tag{29}$$

The function q is the quantification function and is implemented as an 1D texture. This is basically the same approach as for painting cells by Lake et al. as described in the techniques section of this article [Lake et al. 2000]. The function takes a number between 0 and 1 and returns a color. Variations of q can be used to stylize the shadows.

$K_a(x,y)$ also represents a function and can be used to implement gradient fills, which is very popular in cartoon drawings. Gradients differ from surface-normal based shading because they provide color changing in image-space rather than using properties of the surface.

For creating the appearance of 3D intersections of billboards, so called nailboards are used. Nailboards offset the depth at each pixel of a billboard to achieve this effect. By using the depth output register on a modern graphics card, an implementation on the GPU is possible.

Therefore in the same pixel shader that performs illumination, d (the depth of the billboard) is added to the depth of the pixel and the depth buffer test automatically produces correct intersections. A drawback of this method is the reduction of peak fill-rate performance due to disabling early-out depth tests.

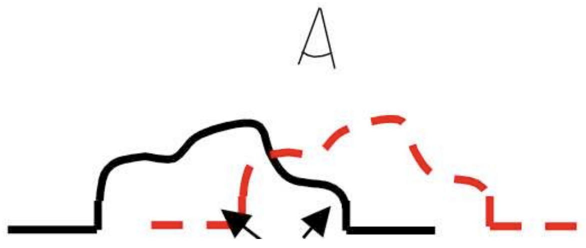


Figure 56: *Schematic view of depth offset for two particles. The depth test will conceal outlines on the interior of the clouds.*

As we can see in Figure 55 the α-shapes are slightly bigger than the others. This is used to create the outlines of the smoke. In these areas α is one and the color C is black and farther away from the camera because the depth d is zero. With this technique different smoke particles seem to merge together because the body of the neighboring particle is nearer than the outline and therefore the body of the particle is visible and the outline is not. So in the interior of a cloud the outlines of are occluded by other particles and on the outside they are visible. This although requires a linear depth buffer because normal depth buffers use hyperbolically interpolated depth values, which results in a higher depth precision near the image-plane and therefore is not suitable because it could distort the outline.

Big clouds are subject to self-shadowing, this is especially important when the clouds are between the light source and the camera so that only small parts of the clouds are illuminated.

McGuire and Fein use stencil shadow volumes and extend it to work with billboards [McGuire and Fein 2006]. For each particle a square with the normal perpendicular to the view vector is created and the shadow volume cast by this square is calculated. This volume consists only of two polygons, one front- and one back-facing polygon, which exactly overlap in image space (technically the volume would consist of six polygons but the other ones are not necessary because they are perpendicular to the view vector). *Figure 57* illustrates the creation of stencil shadow volumes with billboards.

Now the depth values of the shadow volume are modified so that the intersection with other billboards is realistic. The depth values are taken from pre-calculated depth map. With this enhancement the shadowed regions of clouds/smoke can be marked and drawn with a different and darker color and the details in this area can be reduced to achieve a cartoon style effect.

With this algorithm complex smoke or clouds can be created and rendered in real time. There are many values, both in the simulator and in the renderer that can be adjusted to create the exact effect one wants to create. The colors of the cel shading can be adjusted, just like described in the technique section. Self shadowing can be switched on or off and by adjusting the position of the light source, as much detail as desired can be abstracted away from the smoke

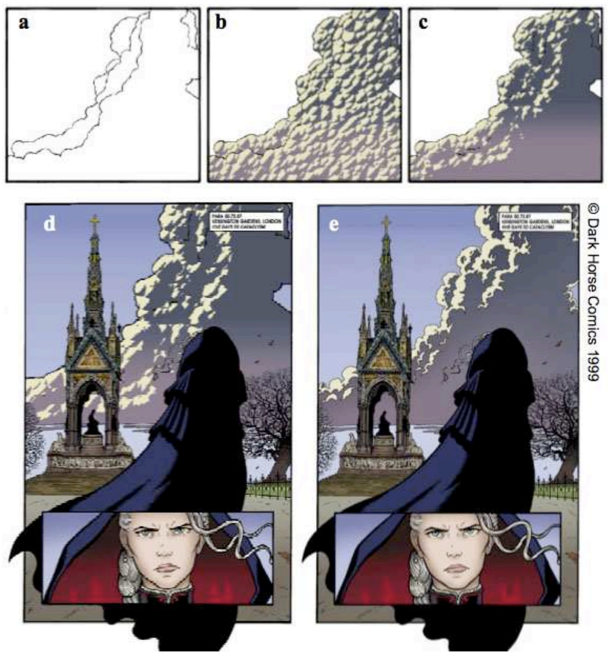


Figure 58: *An example of the output of the real time smoke renderer. (a) Cloud rendered only with outlines (b) The same cloud rendered with cel-shading turned on. (c) And finally rendered with outlines cel-shading and self-shadowing effects. (d) The cloud from c composited into a hand-drawn scene is comparable to (e) the artist's original.*

rendering. This is a very common property of cartoons and comics. And the behavior of the smoke can be adjusted by changing values in the simulator so that artistically behaved smoke and clouds can be created. This is especially useful for computer games. *Figure 58* shows how the output of the smoke rendering engine compares to hand drawn cartoon smoke. Three pictures of a smoke rendering are shown (a-c) with different rendering options. (a) only silhouette rendering turned on, (b) with additional cel shading and c) a combination of silhouettes, cel shading and self shadowing. Then picture (c) is composited into a hand drawn cartoon (d) and compared with the original drawing (e).

The engine renders thousands of particles at 30 frames per second. This makes it suitable for use in a computer game and in rapid development of cartoon animations. Even scenes with 500,000 particles can be rendered in real time on current hardware. But the PCI-express bus is a bottleneck because it is necessary to expand each particle to a billboard and then transmit it over the bus (80 bytes/particle). Future hardware is likely to eliminate this bottleneck by allowing to compute this data on the graphics board.

4.3.3 Shadows

Shadows plays an important role in the perception of the world described with a cartoon looking scene. They provide information of the light direction, atmospheric conditions and can be used for dramaturgy effects. Besides this information, shadows have the main objective to anchor the character in the scene. In cartoon looking scenes the character and the background is normally rendered in different style and the shadow gives the viewer a cue where the feet of the character meets the ground.

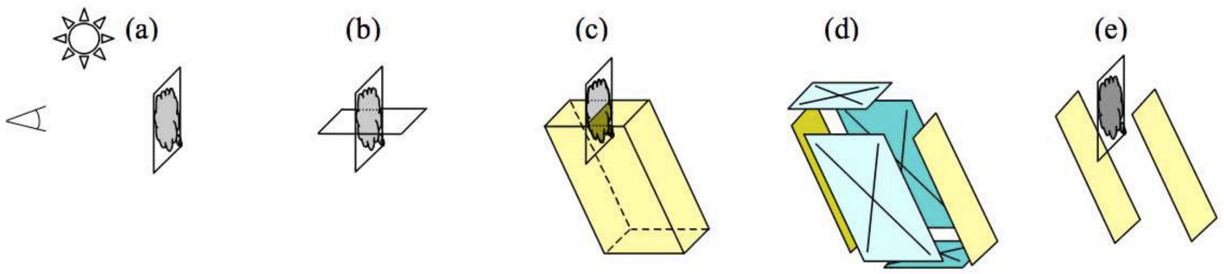


Figure 57: *Construction of a shadow volume out of a billboard. (a) Eye, light source and billboard geometry. (b) Square perpendicular to the normal of the billboard, used as an approximation to generate the shadow volume. (c) The volume cast by the square. (d) Removal of unused polygons, they are perpendicular to the view vector and therefor invisible. (e) The three polygons that are needed for rendering: the billboard and the two shadow faces.*

In the paper of [Petrovic et al. 2000] a semi-automatic method to create shadow maps in hand drawn cel animations is described. The system requires small amount of user input and produces shadow mattes based on hand drawn art.

Much research has been done for rendering physically correct shadows. In cartoon looking scenes physically correct shadows distract the viewer with unnecessary details. [DeCoro et al. 2007] presents a method that gives an artists the opportunity to render more abstract and stylized shadows by adjusting a few parameters. In the remaining part of this section these two techniques are described in more detail.

The method described by [Petrovic et al. 2000] is not fully automatic. The user has to set up the scene and following the shadows can easily be customized by altering several lighting conditions. Initial point is a hand drawn line art and a hand painted scenery, both created by a human artist. The basic algorithm can be outlined by the following points:

1. Arrange the camera, ground plane and background objects by marking features in the painted background.
2. Inflate a 3D mesh out of the character drawn in 2D.
3. Specify the depth for the character in the scene and the light positions.
4. Based on the preceding input three different shadow mattes are rendered for the character:
    - Tone mattes: Self-shadowing and shadows of other objects on the character. *(see Figure 59b, blue)*
    - Contact shadow mattes: Shadow representing the contact point of the character with the ground. *(see Figure 59b, green)*
    - Cast shadow mattes: Shadows cast by the character onto the background. *(see Figure 59b, red)*
5. Compose the shadow mattes in the final image. *(see Figure 59c)*

The background scenery is constructed with a fixed field of view and aspect ratio. The camera roll (rotation about the z-axis) and ground plane both point upright. A few other settings have to be done to establish the relationship between the camera and the scene. The pitch ϕ (rotation about the x-axis) is defined by *equation 30*.

$$\phi = arctan(h/d), \tag{30}$$

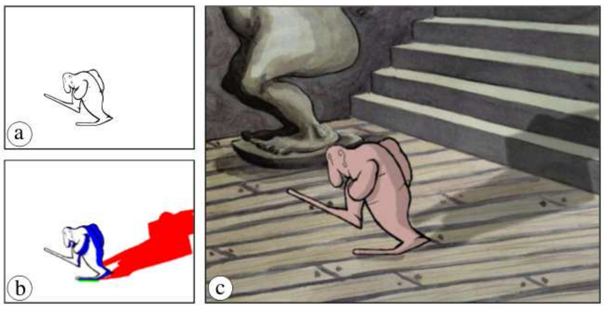


Figure 59: *Images used in shadowing process.*

where h is the height of the horizon relative to the image center and d is the distance from the camera to the image plane. Two parallels lines, which intersects at the horizon in a perspective projection, are drawn by the user on the ground plane to determine the horizon's height *(see Figure 60)*.

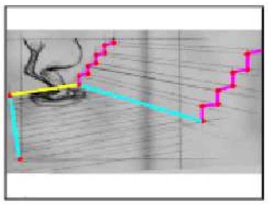
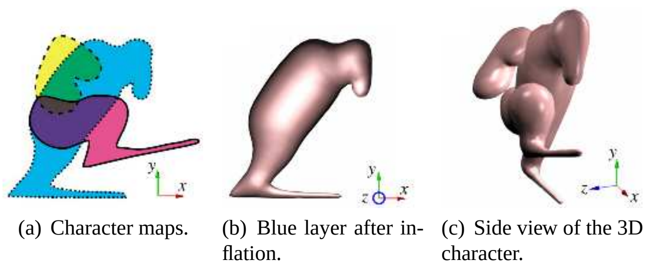


Figure 60: *Parallels lines to determine the height of the horizon.*

Because presently no objects are included in the scene the yaw (rotation about the y-axis) of the camera is set arbitrarily and the height is chosen that the camera stays above the ground plane. Lastly a coordinate system is aligned and objects are constructed relative to the ground plane.

The 2D line art is inflated to a 3D character in the next step to cast plausible shadows. The line art is converted manually into character maps, which are bitmaps that define regions covered by the character *(Figure 61a)*. These map are also used in the normal cel animation pipeline for filling and clipping of the character in the final compositing of the scene. To improve the control possibilities, the character maps are splitted into several layers. Each layer of a character mate is converted automatically in an closed 2D polyline and subsequently inflated to form a 3D shape *(Figure 61b and c)*.

After inflating the layers of the character their depth in the 3D world



(a) Character maps. (b) Blue layer after inflation. (c) Side view of the 3D character.

Figure 61: *Inflated character maps.*

is determined. The two methods depth-translation and depth-shear provides a depth adjustment where the image plane projection is preserved. With a uniform scale about the camera center the depth-translation method moves the figure out of the image plane *(Figure 62)* For finer control of the shadows the depth of the layers can be adjusted with the depth-shear method. In the latter method the layers are shared so that the objects are closer or further away from the image plane. Both of the methods can be adjusted with parameters by the user, separately for each layer. Furthermore for the purpose of animations key frames can be set.

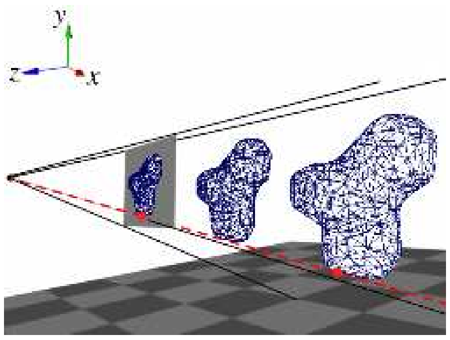


Figure 62: *Depth-translation moves the figure out of the image plane.*

Till now the background and the camera is justified and the a 3D character is positioned in the scene. In the next step different lights (directional light, point light) are positioned and the shadow mattes are rendered. A standard ray tracer is used to render tone shadows and cast shadows. Contact shadows needs a greater extend and are rendered in two passes. In the first pass an orthographic camera is placed in the ground and this captured image is re-projected in the second pass viewed from the original camera position. The color of the character is darkened by the tone mattes and the background is accommodated by the cast shadows and contact shadows. The final image consists of the shadowed character and shadowed background.

A number of reason exists why artists avoid physically accurate shadows. Detailed shadows deflect the viewer from the scene and viewers tend not to recognize if shadows are drawn physically correct or not. Artists have a lot of freedom in designing shadows and some basic methods are already used in practice to abstract shadows, like blurring the shadow map or cast the shadow of an simplified object. [DeCoro et al. 2007] describes a image based method to produce stylized shadows from a shadow matte by varying the following four parameters:

1. *Inflation* (i) controls the size of the shadow, relative to the original, such that increased i gives the effect of a shadow emanating from a larger version of the shadow-casting object.
2. *Brightness* (b) is the intensity of the shadow region when fully occluded (or the effect of indirect illumination).
3. *Softness* (s) indicates the width of the transition region from fully occluded to fully visible, simulating the effect of an area light.
4. *Abstraction* (α) is a measure of shadow's accuracy; lower values yield more detailed, accurate shadows, whereas larger values produce, simplified shadows.

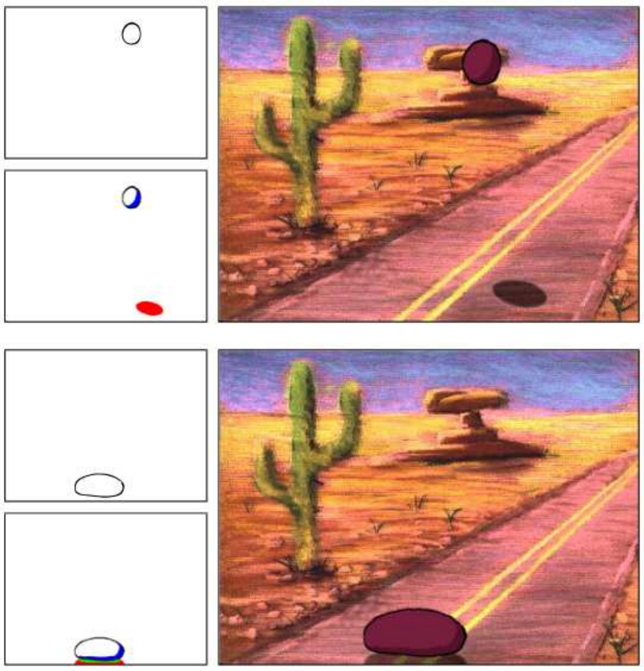


Figure 63: *Final image, composed of a background, the character and the shadow.*

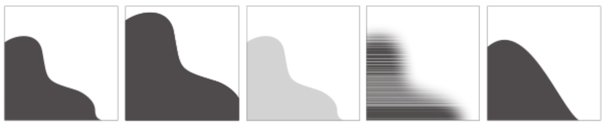


Figure 64: *The effect of the different control parameters. Listed from left to right: Original, Inflation, Brightness, Softness, Abstraction*

Figure 64 shows the influence of the different control paramters (inflation, brightness, softness and abstraction) based on the original image. Following equation is used to calculate the standard illumination of a point x, describe in *equation 31*.

$$L_0(x,\omega_0) = \sum_l \rho(x,\omega_l,\omega_0)S_l(x)L_l(\omega_l), \tag{31}$$

where L_0 is the exitant radiance from x in direction ω_0, ρ is the reflectance, L_l is the incident radiance and $S_l(x)$ is a shadowing function. In real-time applications the shadowing function is the binary visibility $V_l(x)$ and is called the shadow matte. For stylistic control an alternative formulation of the shadowing function $S_l(x)$ is given by defining an operator on $V_l(x)$:

1. Render the visibility buffer $V_l(x)$ corresponding to the l-th light.
2. Compute a signed L_p-averaged distance transform $D(V_l)$
3. Filter D with a Gaussian G, producing $G \otimes D(V_l)$
4. Apply a transfer function f, yielding $S_l(x) = f(G \otimes D(V_l))$.
5. Light the scene with $S_l(x)$ according to *equation 31*.

Increasing the inflation parameter i approximates the inflating of the original mesh by inflating the shadow represented in the matte. A metric is defined that $D(V(x))$ is the distance from x to the original shadow contour and V is the boundary between shadowed and unshaded regions. An inflation at distance i of the hard shadow is equivalent to $D(V(x))$. This can be represented by a threshold transfer function $f_i(D) = threshold_i(D)$, so that a modification of i requires no recomputation of f from $D(V)$. The metric used is based on the *L_p-averaged distance metric* defined over R^3 relative to a surface [DeCoro et al. 2007]. It is used for the displacement along the normal, which is free from cusps and visual artifacts. In *Figure 65* a comparision of an accurate shadow to an inflated shadow is given.

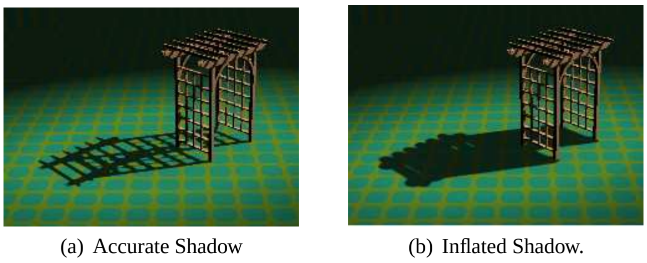

(a) Accurate Shadow (b) Inflated Shadow.

Figure 65: *Comparison of an accurate shadow and an inflated shadow.*

The brightness parameter b simulates the effect of the ambient light in the shadowed area and is the minimum of the transfer function f. It represents the darkest light intensity a shadow can have. The maximum value is 1, that means objects outside a shadow are fully visible to the light. A shadow with a low brightness for the umbra and an inflated shadow with a higher brightness can be combined to produce a more stylized shadow. *(see Figure 66)*

The softness parameter softens the hard shadows. The visibility of a shadow varies continuously across the penumbra region and two shadow contours are considered. One deflating from the location of the hard shadow and one delineating the outermost boundary of the penumbra. These contours can be extracted with the distance transform function, where the softness is the width of the transition from $f(D) = b$ to $f(D) = 1$, which is fully visible.

Figure 67 shows a comparision of an accurate shadow to an softened shadow.

The original hard shadow defines the least abstraction of a shadow and a perfect circle defines the highest abstraction of a shadow. Therefore the abstraction parameter α is a limit of the curvature, a high value of α leads to a rounder shadow. It is implemented by

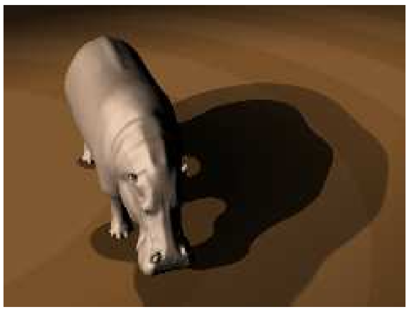


Figure 66: *Usage of the umbra and penumbra to produce a stylized shadow.*

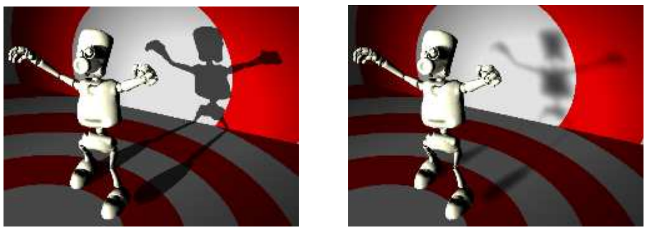

(a) Accurate Shadow (b) Softened Shadow.

Figure 67: *Comparison of an accurate shadow and an softened shadow.*

convolving the distance function with a Gaussian kernel of standard deviation α. *Figure 68* shows a comparision of an accurate shadow to an abstract shadow.

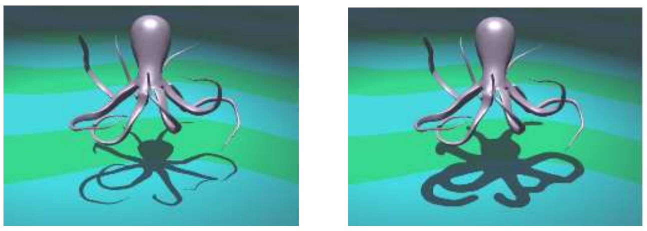

(a) Accurate Shadow (b) Abstract Shadow.

Figure 68: *Comparison of an accurate shadow and an abstract shadow.*

Additional geometry information is used to provide a more intuitive shadow behavior. The geometry information is specified per pixel and the three additional values are the world-space position, normal and distance to the occluder. With the assistance of the world-space position, foreshortening of shadows away from the camera is possible. Normal discontinuities prevent sampling artifacts and with the distance to the occluder the softness or brightness for shadows with a greater distance can be varied.

5 Conclusion

The main focus in computer graphics today lies in the production of photorealistic looking images and animations. Other important aspects of animated scenes and behaviour of the included objects also concentrate on being physically correct. In the beginnings of computer graphics the resources were limited and photorealism was not possible in real time applications. But this forced computer graphic artists to create artistic and abstract looking scenes to attract the viewer. This resulted in a broad variety of different computer graphic styles. Nowadays the artistic aspect is rather small, especially in computer games, because the aim is to produce realistic environments and lighting effects.

Cartoon style rendering is an example to conquer these developments and bring back the stylistic aspect to computer graphics. The most distinctive attributes in comics are the mostly black drawn silhouettes and the discrete interior shading.

We describe methods for silhouette edge detection and their implementation in image space as well as in object space. Image space methods tend to be more efficient than silhouette edge detection methods in object space. The advantage of object space methods is the additional information of the object geometry which can be used to produce stylized outlines. The calculation of the object space methods include an additional overhead that can be overcomed with the programmable geometry stage in the future series of graphic cards. The interior shading of the cels depends on the traditional lighting equation, while using discrete colours. The dot product of the surface normal and the light direction vector ($L * N$) is subdivided into intervals, which represent a solid color value. This creates the distinct look of the solid shaded cels with hard transitions between the shaded and illuminated parts of the object. A method to simulate cartoon looking specular highlights by altering the specular part of the traditional lighting equation is also presented.

We summarize methods to imitate several drawing styles, which extend the basic silhouette detection and cel shading.

To produce hand drawn looking outlines in pencil drawings we present techniques which map textures on the detected silhouettes or alter the textures in a pre-processing step before. The hatching process for the interior shading is done by choosing textures with different stroke intensities. The temporal coherence between successive frames in an animation cycle is problematic and appears in artifacts on the texture borders. The usage of the coherence information should be improved to reduce this errors. Another progress is to support different colors and more typically pencil drawing effects, like erasers. A chinese painting effect operating on the GPU is shown, which simulates the interaction between the monochromatic ink and the traditional Xuan paper. In this interrelation the morphing animation is introduced, but as in pencil rendering the coherence information should be utilized in future research.

Then we also showed how to create different artistic styles by using and extending the existing methods. This was done by the examples of Frank Miller's "Sin City" style, Todd McFarlane's "Spawn" style and Mike Mignola's "Hellboy" style. These were all styles used before in comic books and the goal was to implement them in real time. It turned out that some styles were easy to approximate with only some minor extensions of the basic methods while others are hard to achieve. The main problem lies in the mostly 2-dimensional nature of cartoons and comics. Objects behave differently, shadows are stylistic and animations emphasize on the action going on and mostly look good from the intended perspective.

Another important aesthetic aspect is the shadow of the objects in an scene. A semi automatic method supporting the artist in the creation process of shadows for traditional animations is discussed. Shadows in fine art are more abstract than the physically correct ones generated with the standard shadowing algorithms. We summarize a method to produce more stylized shadows by varying different parameters. In future work this parameters should be functions over time for the possibility of key framing in animated scenes. Moreover the methods can be extend to work with area light sources.

We also showed methods for smoke and water animations and realized that the effects simulated were not very cartoonish while the drawings were. These are all problems we are facing when trying to implement real time cartoon style rendering. Future work has to be done in the stylizing effects and their animations. And maybe we should not concentrate on transferring styles from comic books or cartoons to real-time computer graphics but should keep in mind that we are dealing with a completely different medium with other advantages but also disadvantages compared to the traditional mediums. And therefore create styles suitable for real-time 3-dimensional animations and interactions.

References

BROWN, C., 2007. A gpu-based approach to non-photorealistic rendering in the graphic style of mike mignola. http://www.csee.umbc.edu/∼olano/635/chbrown1.pdf; (last visited on 23-11-2007).

DECAUDIN, P. 1996. Cartoon looking rendering of 3D scenes. Research Report 2919, INRIA, June.

DECORO, C., COLE, F., FINKELSTEIN, A., AND RUSINKIEWICZ, S. 2007. Stylized shadows. In *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*.

DIEPSTRATEN, J., AND ERTL, T. 2004. Interactive Rendering of Reflective and Transmissive Surfaces in 3D Toon Shading. In *Proceedings of GI Workshop Methoden und Werkzeuge zukuenftiger Computerspiele '04*.

EDEN, A. M., BARGTEIL, A. W., GOKTEKIN, T. G., EISINGER, S. B., AND O'BRIEN, J. F. 2007. A method for cartoon-style rendering of liquid animations. In *GI '07: Proceedings of Graphics Interface 2007*, ACM, New York, NY, USA, 51–55.

GOOCH, A., GOOCH, B., SHIRLEY, P., AND COHEN, E. 1998. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 447–452.

HEARN, D. D., AND BAKER, M. P. 2003. *Computer Graphics with OpenGL*. Prentice Hall Professional Technical Reference.

HERTZMANN, A. 1999. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In *ACM SIGGRAPH 99 Course Notes. Course on Non-Photorelistic Rendering*, S. Green, Ed. ACM Press/ACM SIGGRAPH, New York, ch. 7.

LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. 2000. Stylized rendering techniques for scalable real-time 3d animation. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, ACM, New York, NY, USA, 13–20.

LANDER, J. 2000. Under the shade of the rendering tree. *Game developer magazin* (02).

LEE, H., KWON, S., AND LEE, S. 2006. Real-time pencil rendering. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, ACM, New York, NY, USA, 37–45.

MARKOSIAN, L., KOWALSKI, M. A., GOLDSTEIN, D., TRYCHIN, S. J., HUGHES, J. F., AND BOURDEV, L. D. 1997. Real-time nonphotorealistic rendering. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 415–420.

MCGUIRE, M., AND FEIN, A. 2006. Real-time rendering of cartoon smoke and clouds. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, ACM, New York, NY, USA, 21–26.

MCGUIRE, M., AND HUGHES, J. F. 2004. Hardware-determined feature edges. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, ACM, New York, NY, USA, 35–147.

PETROVIC, L., FUJITO, B., WILLIAMS, L., AND FINKELSTEIN, A. 2000. Shadows for cel animation. In *Siggraph 2000, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., 511–516.

SELLE, A., MOHR, A., AND CHENNEY, S. 2004. Cartoon rendering of smoke animations. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, ACM, New York, NY, USA, 57–60.

SPINDLER, M., RÖBER, N., DÖHRING, R., AND MASUCH, M. 2006. Enhanced Cartoon and Comic Rendering. D. Fellner and C. Hansen, Eds., 141–144.

WANG, A., TANG, M., AND DONG, J. 2004. A survey of silhouette detection techniques for non-photorealistic rendering. In *ICIG '04: Proceedings of the Third International Conference on Image and Graphics (ICIG'04)*, IEEE Computer Society, Washington, DC, USA, 434–437.

WINNEMOLLER, H. 2002. Geometric approximations towards free specular comic shading. *Computer Graphics Forum 21*, 309–316(8).

YUAN, M., YANG, X., XIAO, S., AND REN, Z. 2007. Gpu-based rendering and animation for chinese painting cartoon. In *GI '07: Proceedings of Graphics Interface 2007*, ACM, New York, NY, USA, 57–61.