Cartoon-Looking Rendering of 3D-Scenes

Philippe Decaudin[1]

Research Report INRIA #2919

June 1996

Abstract

We present a rendering algorithm that produces images with the appearance of a traditional cartoon from a 3D description of the scene (a static or an animated scene).

The 3D scene is rendered with techniques allowing to:

- outline the profiles and edges of objects in back,
- uniformly color the surface inside the outlines,
- render shadows (backface shadows and projected shadows) due to light sources.

Keywords

computer graphics, non-photorealistic rendering, cartoon, cel shading.

[1] http://www-evasion.inrialpes.fr/people/Philippe.Decaudin

Research Unit INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY cedex, France
Phone : +33 1 39 63 55 11 – Fax : +33 1 39 63 53 30

1 Introduction

This paper presents a rendering algorithm that produces "cartoon looking" computer generated images from data representing a 3D scene. In contrast to classical rendering algorithm which try to generate images as real as possible (photo-realism), cartoon and cel animation images use a more stylized and suggestive rendering [1]. For instance, characters and objects shown on a drawing are represented by their outlines, and the shape inside outlines is usually filled with a uniform color (see figure 1).

1.1 Typical features

Many variations and levels of quality exist in drawings. The size of the outline may vary, colored areas may include gradients, or light and shadow effects may create a kind of atmosphere. We will not try to reproduce all possible effects here. We would like to obtain images that have a stylized "cartoon" (or "cel") look, with constant-size outlines and uniformly colored areas. We add three more effects: the ability to replace a uniformly colored area by a texture, the ability to add specular highlights on some objects, and shadows.

- Texture is not required, but it is an easy way to enrich a drawing, for instance by repeating a regular pattern on a surface.

- Specular highlights provide visual feedback about the material type of an object (flat, shiny, metallic…). They correspond to the reflection of light sources on the objects and are represented by small, localized, brightly colored spots.

- Shadows add information to the image that helps to perceive the 3D aspect of the scene. We will distinguish two types of shadows: *backface shadows* that correspond to unlit sides of objects[2], and *projected shadows* that correspond to areas occluded by an object in-between this area and a light source[3]. We will deal with both types. In 'cheap' classical cartoons, shadows are often not drawn. In more elaborated cartoons, backface shadows are often figurative; an area of an object that is half in light and half in shadow will have two colors: the object's color for the lighted part, and this color darkened for the unlit part. The boundary between the two is sharp, contrary to classical computer graphics images where the transition is smooth (gradient). In the case of a single static light source, backface shadows are quite easy to draw by hand, even if the object moves, because they stay relatively localized during the animation, especially if it is a directional light source (like the sun). On the other hand, projected shadows of moving objects may change a lot during the animation and are difficult to draw by hand, which incites the artist to draw only projected shadows of static objects such as those in the background.

[2] i.e. areas opposed to the light source, where $\boldsymbol{n}\cdot\boldsymbol{l}<0$ (dot product between $\boldsymbol{n}$, the normal of a point and $\boldsymbol{l}$, the vector from this point to the light source)
[3] projected shadows include 'self shadows' that an object projects onto itself.

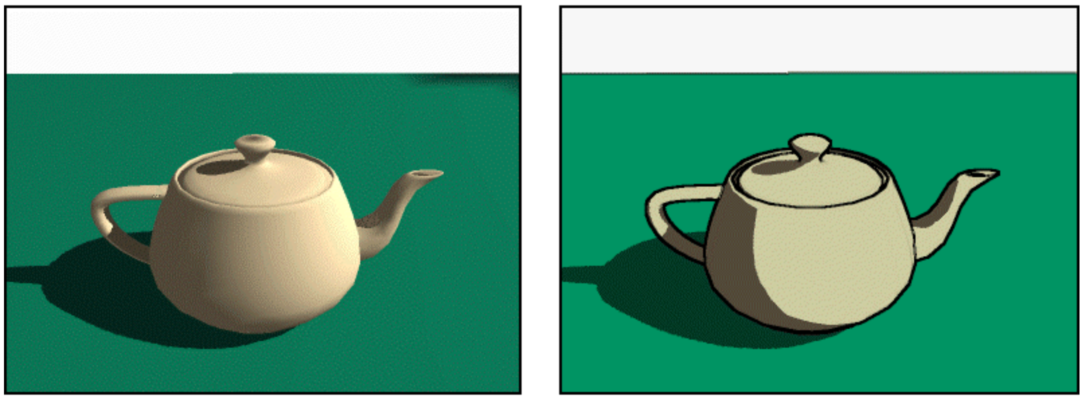


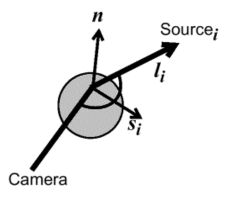
Figure 1: a teapot rendered by a photo-realistic algorithm (left) and by our "cartoon-looking" algorithm (right).

1.2 Principle

Our algorithm produces an image from the geometrical description of a 3D scene. The type of description used does not matter; the algorithm is general and can handle any kind of description (facets, patches, CSG, metaballs…). In our implementation, input scenes are described by an OpenInventor [7] scene graph and geometries are described by Bezier patches and facets. These allow us to use classical projective rendering techniques as a basis for our algorithm. Renderings are produced using OpenInventor and OpenGL [3] graphics libraries and take advantage of hardware to accelerate calculations. Extending to other rendering techniques may be straightforward.

The shading model used by OpenGL is based on the Phong shading model. The formula to calculate the color of a point is:

$$\begin{aligned} \mathit{color} &= \mathit{ambient}_{\mathit{global}} \times \mathit{ambient}_{\mathit{material}} \\ &+ \sum_{i=0}^{\mathit{num\ sources}} \mathit{ambient}_{\mathit{source}\ i} \times \mathit{ambient}_{\mathit{material}} \\ &\qquad + \max\{\boldsymbol{l}_i \cdot \boldsymbol{n}, 0\} \times \mathit{diffuse}_{\mathit{source}\ i} \times \mathit{diffuse}_{\mathit{material}} \\ &\qquad + \max\{\boldsymbol{s}_i \cdot \boldsymbol{n}, 0\} \times \mathit{specular}_{\mathit{source}\ i} \times \mathit{specular}_{\mathit{material}} \end{aligned}$$



Next, we will apply this formula to obtain several desired effects.

Winkenbach and Salesin [9] as well as Salisbury and al. [6] dealt with pen and ink illustration. The effect they obtain is relatively close to a cartoon-looking rendering, but the method they use is more restricted. In [9], only objects composed of plane surfaces can be treated, and in [6], the process requires user interactions, so it's difficult to use it to generate animations automatically.

Our algorithm uses a set of computer graphics techniques: the "shadow maps" [8] technique is used to produce shadows, and the outline drawing technique is inspired by image processing algorithms described by Saito and Takahashi [5]. The originality of our approach owes mainly to how these techniques are adapted and mixed to produce a cartoon looking animations.

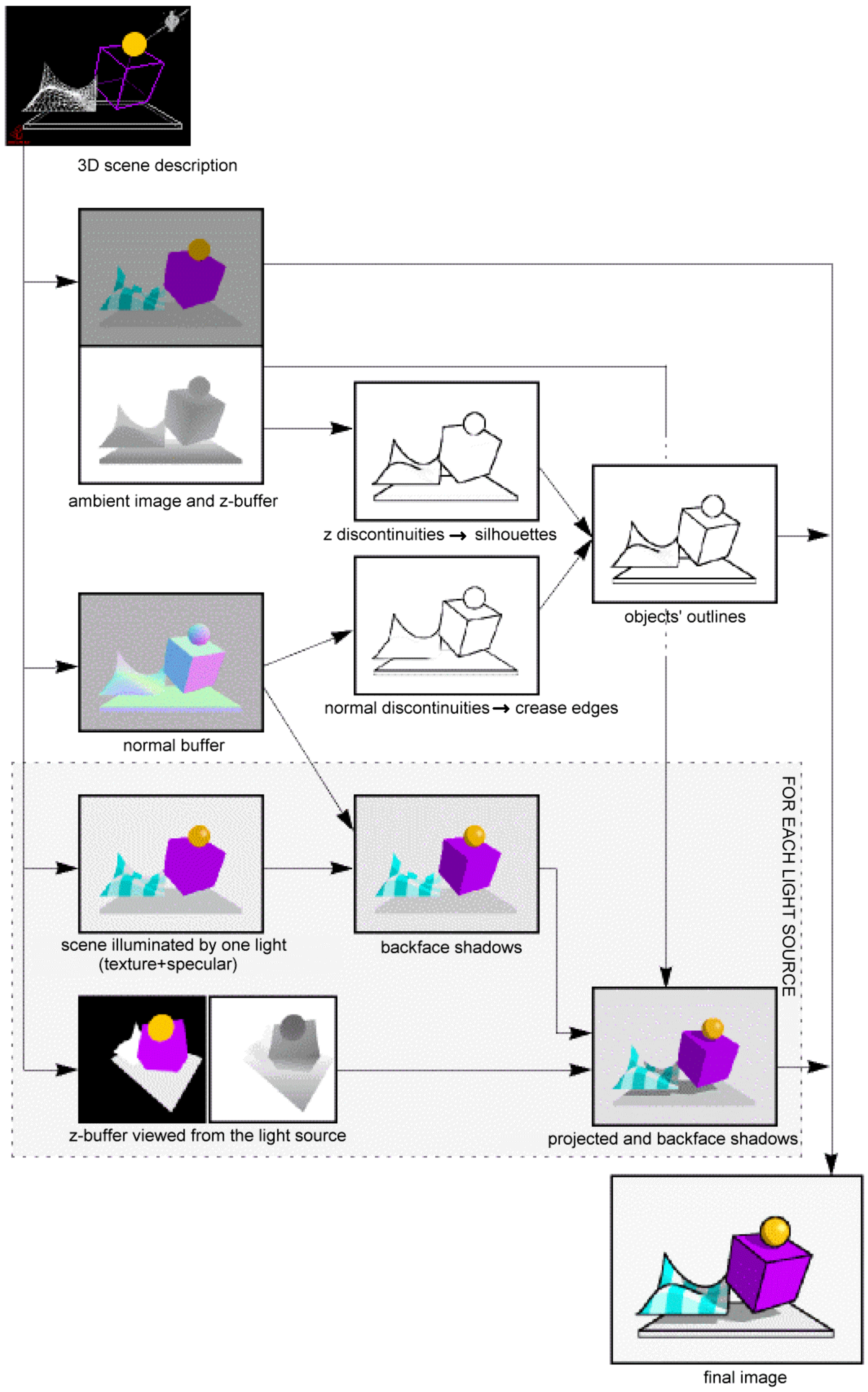

3D scene description
ambient image and z-buffer
z discontinuities → silhouettes
objects' outlines
normal buffer
normal discontinuities → crease edges
FOR EACH LIGHT SOURCE
scene illuminated by one light
(texture+specular)
backface shadows
z-buffer viewed from the light source
projected and backface shadows
final image

Figure 2: overview – algorithm steps

2 Algorithm

2.1 Overview

Figure 2 presents the different steps of our rendering algorithm. Each step produces an image; the combination of these images produces the final image. These steps, which we will elaborate on below, are:

- rendering of the scene with ambient light only;
- calculation of outlines of scene objects;
- for each light source:
  - rendering of the scene illuminated by this light;
  - calculation of backface shadows and projected shadows due to this light;
- combination of the images

2.2 The calculated images

Images and buffers required by the algorithm are:

- Rendering of the scene with ambient light only. This image is produced by rendering the scene viewed from the camera with all lights off. Each object is colored or textured without any illumination (colors are uniform), attenuated by the global ambient intensity of the scene. For instance, if the global ambient intensity is 0.2, a red object (R=1, G=0, B=0) will be displayed uniformly dark red (R=0.2, G=0, B=0).

- *Z*-buffer of the scene viewed from the camera. We take advantage of the previous rendering (the ambient lighted scene) to get its *z*-buffer (i.e. depth buffer, each pixel stores its distance to the camera).

- Normal buffer of the scene. Three values are stored at each pixel of this buffer: the three coordinates (n_x, n_y, n_z) of the normal associated to the corresponding 3D point. It is possible to get this buffer by performing two renderings of the scene viewed from the camera (figure 6). For these two renderings, colors and textures of all objects are replaced by a pure diffuse white material (ambient=black, diffuse=white, no specular). Light sources are off. The first rendering (I_1) is produced by adding three directional light sources to the scene, the first one is red and its direction is $+x$, the second one is green and its direction is $+y$, and the third one is blue and its direction is $+z$. The calculation of the color where normal is $n=(n_x, n_y, n_z)$ gives:

  $$color = red \times \max\{n_x, 0\} + green \times \max\{n_y, 0\} + blue \times \max\{n_z, 0\} .$$

  So the red component of point is $\max\{n_x, 0\}$, the green component is $\max\{n_y, 0\}$, the blue component is $\max\{n_z, 0\}$. The second rendering (I_2) is produced by inversing the direction of the three light sources: red along $-x$, green along $-y$ and blue along $-z$. The color calculation gives:

  $$color = red \times \max\{-n_x, 0\} + green \times \max\{-n_y, 0\} + blue \times \max\{-n_z, 0\} .$$

Finally, subtracting the two renderings I_1-I_2 gives a buffer in which each component red, blue, green corresponds to n_x, n_y, n_z respectively.

For each light source:

- Rendering of the scene illuminated by this light only. For this rendering, we don't take the global ambient of the scene into account. We want to render an image without realistic shading calculations, in order to get the flat colors without gradient that are characteristic of traditional cartoon and cel drawings. To do so, we eliminate the n.l (n is the object's normal, l is the vector directed to the light source) term of the Phong shading equation. If the object is specular, we keep its specular highlight because it provides information about object's material type (flat, shiny, metallic…). Thus, we modify the objects' materials as follows: copy its diffuse color to ambient, set diffuse to black, keep specular. We obtain an image where each object is uniformly colored by its diffuse color (eventually combined with its texture and specular highlight).

- *Z*-buffer viewed form the light source. To obtain this buffer, a camera replaces the light source; a rendering of the scene viewed from this new camera is done, and the *z*-buffer is then read back. It will be used for the calculation of shadows (detailed further) cast by this light source; we will use the "shadow map" technique.

2.3 Algorithm steps

The final image is obtained by processing and combining the buffers and images introduced in the previous section.

2.3.1 Objects' outlines

We are interested in two kinds of outlines: silhouettes (object's profiles) and crease edges (sharp creases).

Silhouettes correspond to places where the view direction (half-line starting from the camera's center and going through a given pixel) is tangent to the object's surface. Silhouettes can be obtained by numerical calculation in object space, but for high level surfaces like patches this often leads to complex algorithms, which are difficult to program due to numerous particular cases and become unstable for points where a silhouette disappears because of a curvature change on the surface (see, for example, the blue and white patch of figure 2). We favor a simpler and more stable approach, with little loss of flexibility, inspired by [5]. Silhouettes also correspond to discontinuities (of zero-order) in the z-buffer of the scene viewed from the camera. A contour detection filter can be applied to this z-buffer to get these discontinuities. To do so, we apply the following differential operator of order 1 and size 3x3:

$$g=\frac{1}{8}\left(|A-x|+2|B-x|+|C-x|+2|D-x|+2|E-x|+|F-x|+2|G-x|+|H-x|\right)$$

where A,B,…,H are the 8 neighbor pixels of x :

A	*B*	*C*
D	*x*	*E*
F	*G*	*H*

This operator produces a gradient of the z-buffer, it has to be threshold to obtain silhouettes that correspond to areas with a high gradient. The following 3x3 non-linear filter is used:

$$p = \min\left\{\left(\frac{g_{\max} - g_{\min}}{k_p}\right)^2, 1\right\}$$

where g_{max} and g_{min} are the maximum and minimum values of the gradient in the considered 3x3 neighborhood, and k_p is the detection threshold in-between 0 and 1. The lower k_p is, the more silhouettes will be detected. Choice of this parameter is not obvious; it is an input parameter of the algorithm, it depends mainly on the size (width x height) of the *z*-buffer, because the more precise the *z* variations, the easier it is to distinguish a discontinuity from a quick but continuous decrease of *z*. In our implementation, the value k_p=0.0001 gives good results for an image at video resolution (768 x 576).

Visible crease edges of objects can be detected in a similar way. Theoretically, a second order differential operator applied to the *z*-buffer should solve the problem (crease edges are *z* discontinuities of order 1), but in practice this operator proves to be instable because it is too approximate. A filter of higher size should give better results, but calculations will become much more expensive. We opt for another solution: these crease edges also correspond to zero-order discontinuities of the normal buffer, so a contour detection filter can be applied to this buffer. It is similar to the one used for silhouettes, except that it may take into account the three normal components n_x, n_y, n_z for the gradient calculation. The choice of k_a (equivalent to k_p but for crease edges) is less sensitive than for k_p, the value k_a=0.2 gives good results.

These several filters compute outlines that characterize objects in a stable way and in a moderate time. Two more problems have to be solved: stroke thickness and stroke antialiasing. Stroke thickness is implied by the two 3x3 filter passes that give a final stroke thickness of approximately 5 pixels. This stroke is aliased because of the abrupt threshold we used: it avoids fuzzy strokes but it generates aliasing. To overcome these drawbacks, we extract outlines on an image of higher resolution than the final image. The result is resampled to get an image of the correct size. Thus, by filtering a double sized image, outlines will finally have a thickness of about 2.5 pixels and will also be antialiased since they are obtained by averaging outlines of 5 pixels thick. Of course, if we want thinner and more precise (antialiased) outlines, calculations become more expensive. Doing calculations on double sized images gives acceptable results to produce video resolution images; the examples included in this paper (figures 1 and 4) are computed using this approach.

2.3.2 Shadows

For each light source, a rendering of the scene illuminated by this light only is generated. We will compute shadows due to this light.

Projected shadows are obtained using the "shadow map" technique. For each pixel, it checks if the corresponding 3D point is visible or not from the light (figure 3). If not, it is in a shadowed area and will be darkened. The algorithm steps are:

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.

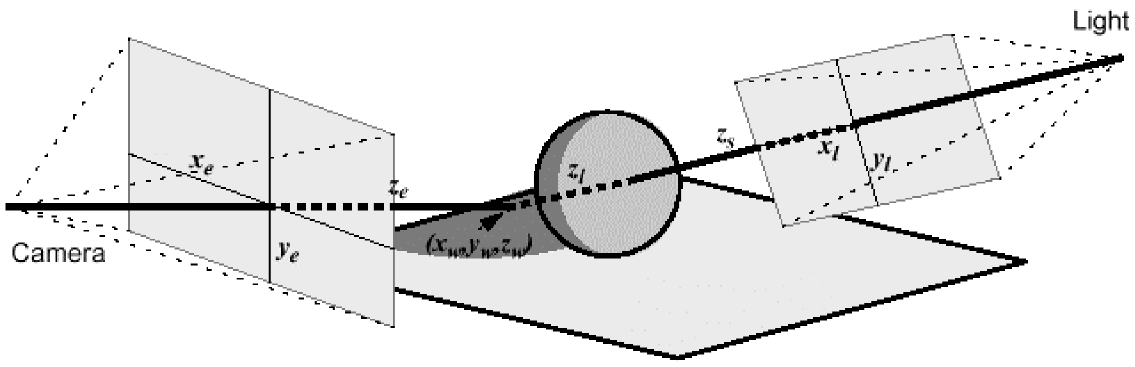


Figure 3: notations used for projected shadow calculation

In practice, this algorithm has a major drawback: it generates aliased shadows because the *z*-buffer has a fixed resolution that may not be enough in some locations. An object that occupies a big space on the image viewed from the camera can be small on the image viewed form the light source. The resulting projected shadow is then imprecise (aliased). Reeves, Salesin and Cook [4] proposed a method to overcome this drawback and to obtain soft-shadows. It consists in comparing several pixels near (x_l,y_l) and averaging the result (see their paper for more details).

This algorithm should allow object backface-shadowing as well, but the result is not good enough for a "cartoon-looking" rendering where the backface shadow boundaries must be precise: zs read from the *z*-buffer of the light source is not accurate enough around this shadow boundary. Fortunately there is another way to get object backface shadows: backface shadows are object areas opposed to the light source, i.e. where $\boldsymbol{n}\cdot\boldsymbol{l}<0$ (dot product between ***n***, the normal of a point and ***l***, the vector from this point to the light source). Thus the normal-buffer of the scene can be used with the *z*-buffer of the scene viewed from the camera to do this test.

We obtain an image with projected and backface shadows due to the given light source. Cumulating the images obtained for each light source will produce the final image as described in the next section.

2.3.3 Calculation of the final image

Our algorithm produces the final image by adding the ambient rendering and all the lit and shadowed images (one for each light source), and then multiplying the result by {one minus the outline buffer}, which is a buffer with 1 everywhere except near the outlines. Thus, the global ambient of the scene, light sources and shadows are taken into account, and outlines appears as black lines.

3 Implementation and results

This algorithm has been implemented in C++ on a Silicon Graphics workstation. It takes as input a scene in OpenInventor format and uses OpenInventor and OpenGL graphic libraries to execute the different steps of the algorithm. These libraries are used to modify the scene (materials and camera) and to do offscreen renderings to get the required buffers.

We used our algorithm to realize a short cartoon animation (1mn12) named "Rendez-vous". Image sequences are generated directly from the 3D descriptions of the scenes and their animations. There is no hand drawing. These scenes were modeled using some Silicon Graphics tools such as SceneViewer (figure 5) and Noodle, and saved as OpenInventor scripts. Animations are created by editing these scripts and by adding some animated nodes to OpenInventor scene graphs that describe scenes. Figure 4 shows some images from the movie. It is composed of more than 1500 images generated at video resolution (768 x 576). Calculation of an image took about 6 minutes on an INDY workstation (MIPS R4400 200MHz).

4 Conclusion and perspectives

Compared to traditional cartoon/cell drawing techniques, an advantage of our technique is that it produces images automatically. In fact, the artistic work is moved ahead: the scene must first be created and animated using 3D modeling tools. Such a system is of course less flexible because it is difficult to create expressive animations such as those found in some cartoons; but on the other hand, it becomes easier to keep spatial and temporal coherency during the animation. For instance, object and camera rotations and also light movements can be easily created; furthermore shadows will stay coherent.

The method still has a number of shortcomings, however. We'd like, for example, to deal with transparent objects, or to control the thickness of strokes independently of antialising, and to vary this thickness according to the curvature of the surface at this place (a technique used in cartoons and comics to emphasize the curvature).

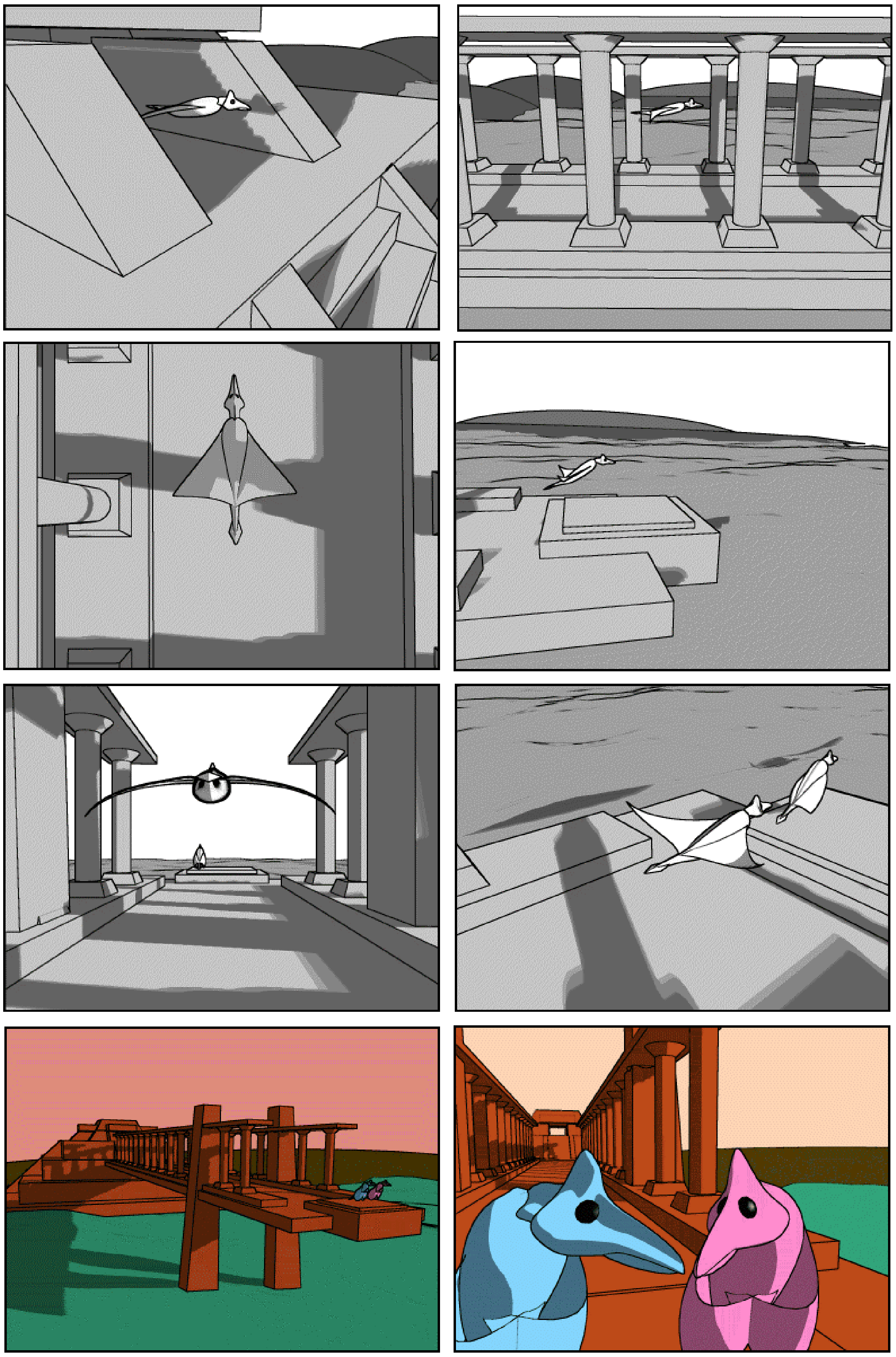


Figure 4: some images from the movie “Rendez-vous”

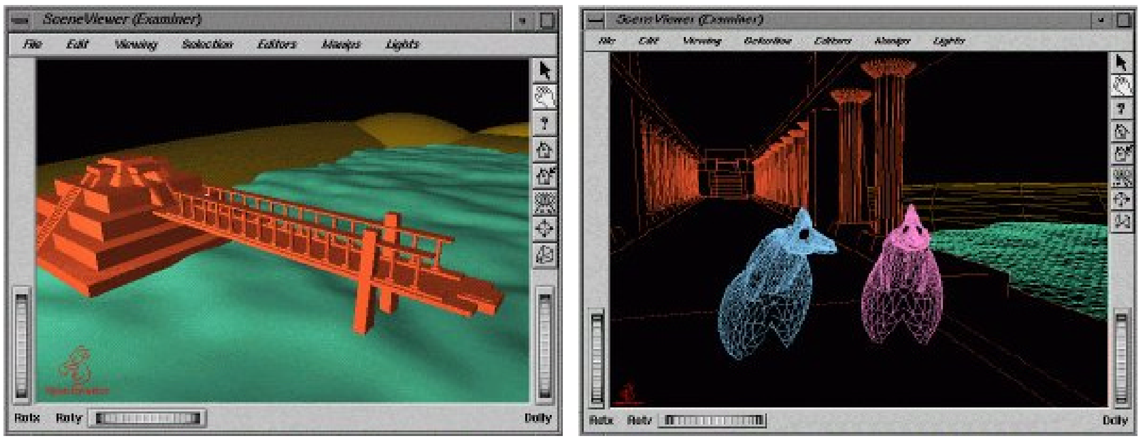


Figure 5: OpenInventor scene used for "Rendez-vous"

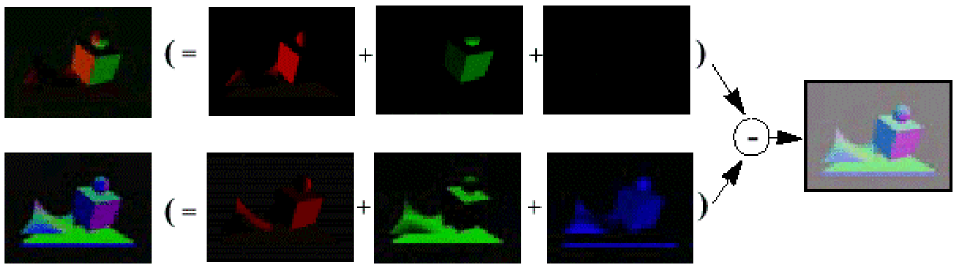


Figure 6 : normal-buffer construction; the resulting image (right) includes RGB values in-between $[-1,+1]^3$ corresponding to n_x,n_y,n_z (coding used: -1,-1,-1=black, 0,0,0=grey, 1,1,1=white)

5 References

[1] B. Duc, L'Art de la BD, Tome 2: La Technique du Dessin, *Editions Glénat*, 1983.

[2] J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes, *Computer Graphics: Principles and Practices (2nd edition)*, Addison Wesley, 1990.

[3] J. Neider, T. Davis and M. Woo, OpenGL Programming Guide (Release 1), *Addison Wesley*, 1995.

[4] W. T. Reeves, D. H. Salesin and R. L. Cook, Rendering Antialiased Shadows with Depth Maps, *Computer Graphics 21,4 (SIGGRAPH'87 proceedings*), July 1987, pp. 283-291.

[5] T. Saito and T. Takahashi, Comprehensible Rendering of 3-D Shapes, *Computer Graphics 24,4 (SIGGRAPH'90 proceedings*), August 1990, pp. 197-206.

[6] M. P. Salisbury, S. E. Anderson, R. Barzel and D. H. Salesin, Interactive Pen-and-Ink Illustration, *Computer Graphics, Annual Conference Series, (SIGGRAPH'94 proceedings*), July 1994, pp. 101-108.

[7] J. Wernecke, The Inventor Mentor - Programming Object-Oriented 3D Graphics with Open Inventor (Release 2), *Addison Wesley*, 1995.

[8] L. Williams, Casting Curved Shadows on Curved Surface, Computer Graphics 12,3, August 1978, pp. 270-274.

[9] G. Winkenbach and D. H. Salesin, Computer-Generated Pen-and-Ink Illustration, *ComputerGraphics, Annual Conference Series, (SIGGRAPH'94 proceedings*), July 1994, pp. 91-100.