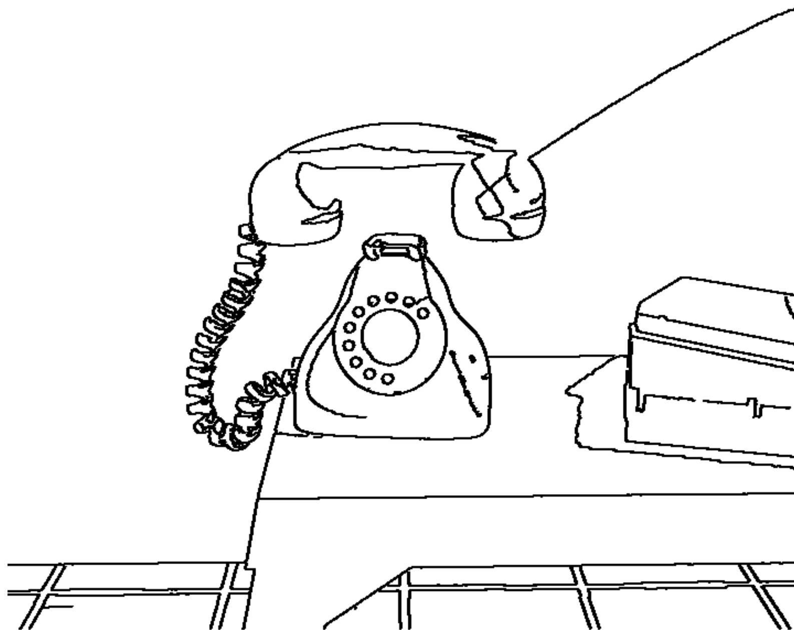
Blending Reality: cartoon looking renders of 3D scenes



Judit Escoda

Msc Computer Animation 2006

Masters Project

Table of Contents

I.- Introduction ……………………………………………………...…………… 1

II.- Previous work …………………………………………………….................. 6

II.i.- Object space methods ………………………………………………………… 6

II.ii.- Image space methods ……………………………………………………….. 8

III.- Edge detection from images ……………………………………………………….. 10

III.i.- Using Y-maps …………………………………………………………..... 10

III.ii.- Canny's algorithm ……………………………………………………….. 12

III.iii.- Cartoon shading ………………………………………………………… 13

IV.- Implementation of Canny's algorithm ……………………………………….. 14

IV.i.- Derivative operators ……………………………………………………… 14

IV.ii.- Noise and the Guassian function ……………………………………… 16

IV.iii.- Non-Maxima Suppression …………………………………………… 17

IV.iv.- Hysteresis thresholding ……………………………………………… 19

V.- Cel shading models ……………………………………………………........... 21

V.i.- Standard shading models ……………………………………………….. 21

V.ii.- Gooch shading model …………………………………………………….. 22

V.iii.- Cartoon shading ……………………………………………………….. 23

References ………………………………………………………………………….. 24

Appendix I: Code for Canny implementation …………………… ………………... 30

Appendix II: Code for cel shading models………………………………………… 40

I.- Introduction

The emergence of computer graphics techniques in the 1980s completely revolutionized the way animated characters and imaginary objects were created. Up until that time, any form of non-realistic animation had to be drawn by hand, performed by artists in celluloid tablets. This traditional style of animation has been evolving over time so as to incorporate computers and computer graphics into their production pipeline. Computer graphics have however been considered a complementary tool providing wider flexibility to changes in the late stages of the production process. Nonetheless, the influence of computer graphics in traditional animation has been gaining strength over the past decade.

On the first place, two dimensional animation systems have been developed in order to facilitate the creation of characters and background. Secondly, it has been observed that certain particularities of animation can now be simplified by the use of three dimensional computer graphics. As such, camera movement can be performed more coherently in three dimensional space, as well as the reusability of characters designed in a three dimensional environment. Such incorporation has been used in a variety of applications, ranging from games such as *Dragon Ball Z* to feature films such as the *Lion King* and *Spirited Away*.

The incorporation of 3D computer graphics into 2D animation does not come along without giving rise to certain considerations. First of all, complexity is reduced at the expense of loosing aesthetic control over the final image. Secondly, 3D techniques are still used as a, now more important, complementary tool. This means that the style of the animation is still driven at the artist's wish and, ideally, the human eye should not be capable of distinguishing those parts of the animation that are two dimensionally drawn from those that arise as the render of a three dimensional scene.

The above problems can be addressed by developing techniques that will render a 3D scene into a particular 2D style. The question is therefore to gain control over those specificities that distinguish a final image arising from one or other environment. Two main distinctions have been addressed here. In cartoon animations, objects appear as

bounded regions filled with a particular colour. As a consequence, objects appear flattened by the influence of light. The first distinction is addressed as an edge detection problem, whereas the second embraces the development of specific shaders so as to achieve the desired look. We shall explore in more detail the above specificities in the following sections.

II.- Previous work

In order to overcome the distinction between object recognition in 2D animations and 3D, the problem has been reduced to that of finding the outlines of objects composing a three dimensional scene. This is known in the literature as the *edge detection* problem, and its applications range from computer generated technical illustrations to architecture. The methods developed over the years are now being applied to other fields of non-photorealistic rendering (NPR), including our subject of study here, that is cartoon looking renders of 3D scenes. While the achievement of a cartoon, *toon* or *cel* shaded look has been considered a major part of this project, special attention has been paid to the edge detection problem due to its complexity. We shall overview here previous work in this field. Any technical specificity relevant to our approach will be described in the next section.

II.i.- Object space methods

In a broad sense, edge detection techniques are classified into object space and image space methods. Image space algorithms extract object outlines from rendered images, whereas object space algorithms do so at render time. While a lot of geometrical information is lost in the first approach, it has the advantage of being less computationally expensive, easier to implement and independent of the surface representation employed for the objects. However, the performance of image space algorithms is highly dependent on the bites stored in the image buffer.

One of the major problems facing object space edge extraction is the hidden line removal problem. This is an important fact about edge detection: visible silhouette edges can potentially be occluded by model geometry. In practice, this implies testing each edge from a geometrical model for existence -determine whether it is a silhouette edge- and visibility -so that only edges that are not occluded by existing geometry are rendered- every time the viewpoint is changed. One can readily see that, for complex objects, this can be extremely expensive in computational time.

Markosian et al. [1] approached this problem by developing a real-time probabilistic method for large polygonal meshes. Their method tests a percentage of the edges of a model at each time step to determine if any of the tested edges compose the silhouette of the object. The set of edges for testing is chosen at random. Once a silhouette edge is found, the algorithm traces out a silhouette curve by recursively testing neighbouring edges. Using this method, the most visually important edges are detected, but it does not guarantee that all silhouette edges are found.

An alternative method was developed by Buchanan and Sousa in [2]. Their method introduces the concept of *edge buffer*. This is defined as a data structure containing edge information extracted from a polygonal mesh. The idea is to identify each edge by the two vertices defining it. Each vertex is then tested to see if it is shared by a front-facing and a back-facing polygon. If this is the case, then the two vertices are marked as defining a silhouette edge. This data structure will be read back at render time in order to produce an image containing silhouette edges.

Vertex	*VFB*	*VFB*	*VFB*	*VFB*
1	200	300	400	500
2	300	500	x00	x00
3	400	500	x00	x00
4	500	x00	x00	x00
5	x00	x00	x00	x00

Figure1: edge buffer structure

Raskar [3] proposed a method consisting on the subdivision of a scene in different layers. Then, using a depth buffer, the rendering process computes the intersection of the two layers in order to determine edge silhouettes. The scene is thus subdivided in a layer containing visible polygons and another one containing those behind them.

A method that is off the scope of this project but that has been considered in its simplified version is that developed by Meier [4] for rendering animations in a painterly style. The idea is to model surfaces as 3D particle sets which are rendered as 2D paint brush strokes in screen space. Her method allows extracting information from the 3D model such as stroke direction and intensity. This method was studied as a way of

extracting edge curvature, allowing only particles lying on the surface boundary to be rendered.

II.ii.- Image space methods

Saito and Takahashi [5], on the other hand, provided a method based on image space that allowed extracting geometry information for edge drawing and curve hatching. Their method computes profile edges from the depth image, whereas internal edges are computed from its second derivative. Their approach has been further enhanced by Decaudin [7]. Decaudin's method augments the depth map information by using normal maps[1].

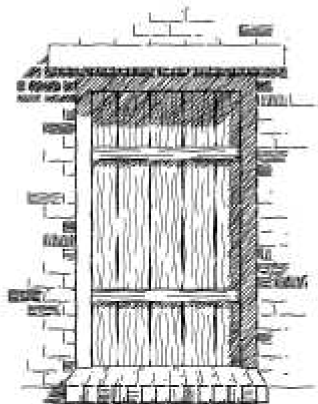


Figure2: curve hatched window using methods described in [6]

The reason why we can extract edge information from the depth and normal maps is because edges appear in an image as luminance discontinuities. In a depth map, objects appear at different intensities depending on their distance from the viewpoint. On the other hand, interior edges are defined as discontinuities in surface orientation. For example, a human face profile can be easily extracted from a z-map. However, if we were to look at the face from the front, details such as nose and mouth would only

[1] The normal map uses the same principle as the depth map, with the difference that the values stored in each pixel correspond to the x, y and z components of the normal to the surface.

appear as revealed by surface orientation discontinuities. Decaudin proposes the following method in order to obtain discontinuities in surface orientation:

- set all the objects to white
- place a red light along the positive x direction, a green light along the positive y direction, and a blue light along the positive z direction
- use the same light colour but with negative intensities for the negative x, y and z directions
- output the rendered scene as an n-map

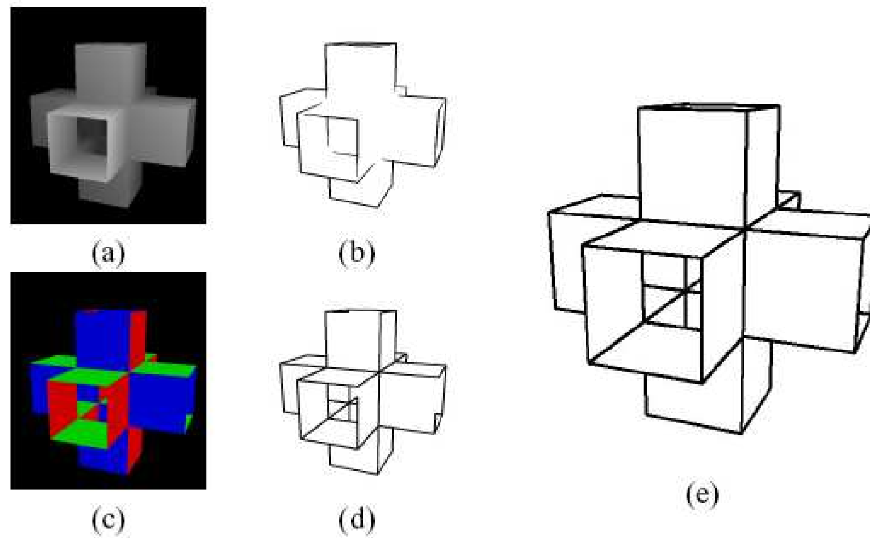


Figure3: Outline drawing with image processing. (a) Depth map. (b) Edges of the depth map. (c) Normal map. (d) Edges of the normal map. (e) The combined edge images. (f)

Because of the object white colour, each surface in the scene will reflect one or other colour depending on its orientation. Where there is a discontinuity in luminosity, there is an interior edge. The approach taken here follows the lines of [5] and [7].

III.- Edge detection from images

According to the discussion above between object space and image space algorithms for silhouette edge extraction, one can readily observe the difficulties arising from one and the other. First of all, a distinction needs to be made in object space methods according to the surface representation of the model. We have not discussed here methods for extracting edge information from NURBS and subdivision surfaces, but a detailed explanation may be found in [8]. Secondly, most of these methods require a special purpose raytracer capable of performing the required operations. Finally, the hidden line removal problem gives rise to computationally expensive renders.

It has thus been considered more suitable to follow an image space approach. The lost of information from the three dimensional model has not been considered of primary importance. If we were to shade the objects using curve hatching or stroke textures, an object space approach would have been considered more suitable. However, for a cartoon looking rather than illustrative or painterly render, edge detection with no information about curvature has been considered sufficient. Curve information should be considered an extension of the method developed here so as to achieve a more hand-drawn look.

III.i.- Using Y-maps

In our approach, Y-maps[2] are considered as an alternative to z- and n- maps as described by [5] and [7]. The reason for that is that z-maps alone do not suffice to extract all the required edge information. It is required to use n-maps at least for interior edges, as proposed by Decaudin. However, the use of negative light intensities is not always permitted. Whereas rendering systems such as OpenGl allow the use of negative light intensities, a rendering system such as Renderman or mentalray do not. In such cases, two separate passes are required: one containing surface normal information in the positive x-, y- and z- directions, and one in the negative x-, y- and z- directions. This

[2] Y-maps contain the grey scale component in NTSC coordinates.

would give rise to a minimum of four passes: one for colour, one for z-depth information, and two for surface normal information. Instead, a grey scale image might be used, and this information is provided in the Y-map of the rendered scene.

Edges might then be extracted from the z-, n- or Y- map using convolution[3] methods. A differential operator is usually convolved with the rendered pass in order to obtain a binary image containing edge information. Several edge operators in differential form have been described in the literature [9], each one of them having its own purpose and different performances. The performance of a given edge operator does not depend however so much on its efficiency as on the image it is convolved with. It has been suggested in [5] to use a Sobel differential operator for the extraction of silhouette edges from the z-map, and using second derivatives for the detection of interior edges. Because of the use of a Y-map in our approach, an alternative general purpose method acting on grey scale images has been preferred. This method was developed by Canny [10] in 1986 and it is considered to be the optimal algorithm for edge detection. For this reason, Canny's algorithm has been studied in more detail. The motivations for this choice have been further enhanced by the criteria behind Canny's purpose. When developing his algorithm, Canny set up the following crietaria:

Canny's algorithm creteria
- low error rate: it is important that edges occurring in images should not be missed and that there are no responses to non-edges - good localization: the distance between the edge pixel as found by the detector and the actual edge is to be at a minimum

[3] In mathematics, convolution is the result of taking two functions *f* and *g* to produce a third function *h* that represents the amount of overlap between *f* and *g*.

It has been considered that low error rate would be crucial for ensuring frame to frame coherence, whereas good localization would ensure that the outlines define in a precise way the shaded object.

III.ii.- Edge detection and Canny's algorithm

The basic idea behind Canny's algorithm is to detect the zero-crossings of the second derivative of the smoothed image. Canny seeks out zero-crossing of

$$\frac{\partial^2 (G * I)}{\partial^2 n}$$

where *n* is the direction of the gradient, *G* * *I* is the convolution of the Gaussian filter with the image *I*. Effectively, this is implemented as taking the gradient of a smoothed image and then seeking local maxima along the gradient direction. A correct implementation of Canny's algorithm can be summarized as follows:

Canny's algorithm
- Gaussian filter: smooth the image with a Gaussian filter to reduce noise and unwanted details and textures - Gradient operator: compute the gradient magnitude of the smoothed image using a gradient operator - Non-maxima suppression: set to zero any magnitude value that is not a local maxima in the direction of the gradient - Hysteresis thresholding: select pixels forming connected contours using edge linking

III.iii.- Cartoon shading

We shall describe in more detail in the next section Canny's algorithm and its implementation. Although edge detection has been considered the major focus of attention in this project because of its complexity, we should not forget that we are seeking to achieve a cartoon looking animation. We therefore must overlay the edge image with a coloured render of the objects in the scene. This has been our second focus of attention, that is, how to shade 3D objects so that they appear as belonging to a two dimensional scene. As we mentioned earlier, objects in 2D animations appear flat-painted. This effect can be simulated by creating two ranges of colours for a same object, one for its shadowed regions and one for the lighted ones. Based on observations from hand-drawn technical illustrations, Gooch [11] developed a shading model in which a warm yellow-like colour is used for lighted areas and a cool blue-like one for the shadowed ones. This cool-to-warm colouring is mixed with the real colour of the object, giving rise to a flattened but still rather three dimensional look. A similar approach has been used here, extending existing shading models such as matte, plastic or metallic.

IV.- Implementation of Canny's Algorithm

As mentioned earlier, edge detection is an area with many applications, including image processing and computer vision. It has been the subject of study for many years, principally due to the assumption of its importance as a low level task for shape recognition. This is one of the processes at work in cartoon animation, in which objects are primarily identified by the outlines defining them. How we draw outlines from three dimensional objects has thus been the main subject of study in this project. We have explained in the previous section that this problem is reduced to finding edges from a rendered Y-map. We shall explain now how this process is carried out by first looking into the theory behind image processing and then describing the implementation of the Canny edge detector.

IV.i.- Derivative operators

In order to understand how edges can be extracted from an image, consider the following figure. Figure 4 shows a functional representation of the visual signal arising from an image. The appearance of edges in an image as shown in (a) gives rise to an image intensity function (b) spatially distributed over its pixels.

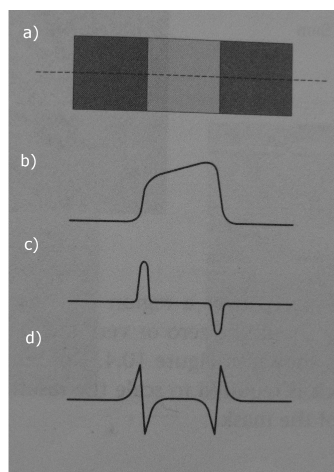


Figure4: a) edge as it appears in an image, b) image intensity function, and its first c) and d) second derivatives.

A very intuitive way to analyse the image intensity function is using mathematical functional analysis and differential calculus. Usual derivative operators can then be applied on a pixel basis for a given intensity function. Areas of constant intensity will have derivative zero, whereas areas of changing intensity such as those defined by an edge will produce a point of extrema in the first derivative and zero-crossing in the second derivative, as one can see in figure 4 c) and d).

In this way, it suffices to locate points of extrema or zero-crossings in the derivative or second derivative respectively of an image in order to extract edge information. The derivative of an image is performed by convolving the original image with a *mask* that approximates a differential operator. The convolved image is a sharpened version of the original one where detail is emphasized.

Conceptually, a mask is a window of small size that is screened over the image, a bit like a usual mask will be overlaid with a face to give rise to a personalized yet undistinguishable appearance. The mask will process one pixel at a time in the original image, replacing the value of the given pixel by an averaged sum of the pixels surrounding it. A mask can be represented as in figure 5, where the letters *a-h* represent the weights of the neighbours of *p*, that is each pixel contribution to the final sum for pixel *p*.

a	b	C
d	p	E
f	g	H

Figure 5: template mask

The gradient approximation of an image $I(x,y)$ for pixel with coordinates (x,y) can thus be expressed as a function $f(a, b, c, d, e, f, g, h, p)$, where a is the weight for pixel $(x\text{-}1, y\text{-}1)$, b for $(x\text{-}1, y)$ etc. Depending on the gradient approximation, the function f may take different forms. A common requirement though is that the weights sum to zero so that if we place the mask over a region of constant or low varying intensity, the output of the convolution will be zero or significantly small. The performance of a mask will depend on the edge contrast. In practice, two different differential masks are used: one along rows and one along columns, the x- and y- masks respectively. The y-mask is

usually obtained with a 90º rotation of the x-mask. In [5], the use of a Sobel mask is suggested. The Sobel gradient mask is given in the following figure

$s_x = \frac{1}{8}$

-1	0	1
-2	0	2
-1	0	1

$s_y = \frac{1}{8}$

1	2	1
0	0	0
-1	-2	-1

Figure 6: The Sobel gradient mask along the x and y directions

IV.ii.- Noise and the Gaussian function

Because the digital approximation to the gradient of an image has the property of sharpening the image by emphasizing its detail, in the presence of noise, certain aspects that should not be emphasized could appear as describing an edge. In order to overcome this problem, the original image is often convolved with a smoothing filter prior to convolution with a gradient mask. Canny makes use of a Gaussian smoothing function

$$G(x,y) = \exp\left[\frac{(x^2+y^2)}{2\sigma^2}\right]$$

where σ is the standard deviation of the Gaussian distribution. When applied to an image, the Gaussian filter., shown in figure 6, will spread out all the values in the image by the shape of the filter. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution mask, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the mask at this point. Hence, σ effectively controls the width of the Gaussian mask.

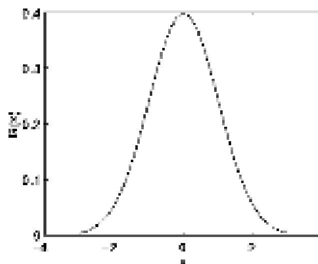


Figure 7: the Gaussian distribution function

The commutative and associative properties of the Gaussian filter, together with its separability, make it very appealing for convolution purposes: as a result, the filter can be implemented as a sequence of convolutions with 1D masks. As a consequence, we have considered a one-dimensional Gaussian mask in our implementation, whose kernel dimension is controlled by the value of σ, convolved with a one dimensional differential Prewitt operator, being the simplest and the fastest

$$dx = [\ -1\ 0\ 1\ ] \text{ and } dy = [-1\ 0\ 1]$$

We have described so far the first two steps in the implementation of the Canny detection algorithm, that is, how to take the derivative of a smoothed image. Whereas the first two steps are widely employed in image detection algorithms, the last two steps are specific to Canny's algorithm.

IV.iii.- Non-maxima suppression

By defining an edge pixel as one with a large change in intensity, we often get edges greater than one pixel thick. That is, there might be wide ridges around the local maxima in the magnitude image. It is sometimes convenient to restrict edges to exactly one pixel. Canny developed a thinning technique that he termed non-maxima suppression. Applying this method, edge pixels are defined as those for which we have a local gradient maxima in the direction perpendicular to the edge.

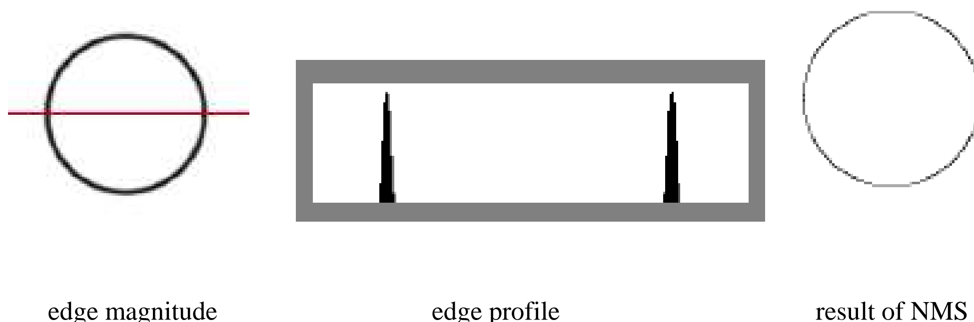


Figure8: Thinning wide contours in edge magnitude images by non-maxima suppression. The intensity profile along the indicated line is shown resized for better visibility.

The gradient of a 2D function indicates the direction in which the function is changing most rapidly, so to determine if a given point in the gradient magnitude image is a maximum, we need to check surrounding points in the direction of the gradient. The algorithm to perform non-maxima suppression is summarized here

- for each position (x,y), step in two directions perpendicular to the edge orientation $\theta(x,y)$.
- denote the initial pixel (x,y) by C, the two neighbouring pixels in the perpendicular direction by A and B (see Figure9)
- if $G(A) > G(C)$ or $G(B) > G(C)$, discard the pixel (x,y) by setting $G(x,y) = 0$

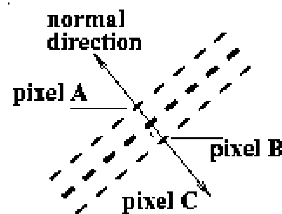


Figure9: edge direction (doted lines) and edge normal (indicated by an arrow)

In practice, interpolation between the four closest neighbours in the direction normal to the edge is used in order to obtain more accurate results. In order to estimate the magnitude at a given point A on the normal direction, the two closest points are chosen, say at pixel $P_{x,y-1}$ and $P_{x+1,y-1}$ in figure 10.

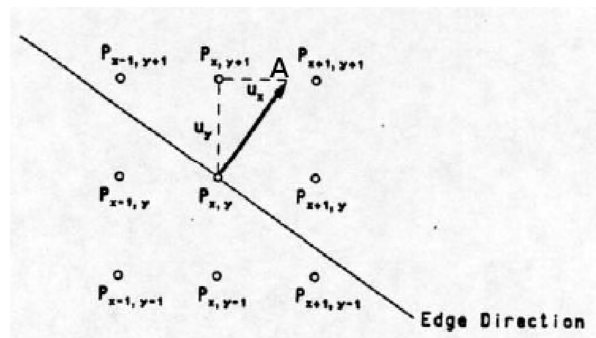


Figure10: pixels chosen for interpolation for estimation of the magnitude value at A

The three points $P_{x,y}$, $P_{x,y-1}$ and $P_{x+1,y-1}$ define a plane from which we can locally approximate the gradient magnitude surface to estimate the value at the point A. The interpolated magnitude at A is given by

$$G_A = (u_x/u_y)\, G_{x+1\ y+1} + ((u_y - u_x)/u_y)\, G_{x,\ y+1}$$

Similarly, in the magnitude value at a point B in the opposite side is given by

$$G_B = (u_x/u_y)\, G_{x-1,\ y-1} + ((u_y - u_x)/u_y)\, G_{x,\ y-1}$$

Then $P_{x,y}$ is at a maximum if $G_{x,y} > G_A$ and $G_{x,y} > G_B$.

IV.iv.- Hysteresis thresholding

Once the non-maxima suppression is applied to the magnitude image, edge strength may be different in different points of a contour and some pixels may be marked as local maxima but still not belonging to a contour. Careful thresholding of $G_{x,y}$ is needed to remove weak pixels while still preserving the connectivity of the contours. Canny suggested the selection of two thresholds, an upper and a lower one, by which a pixel (x,y) in the outputted non-maxima suppression image NMS can be marked as

- strong: if NMS(x,y) > thigh
- weak: if NMS(x,y) < tlow

All other pixels are called candidate pixels. A pixel is then selected as an edge pixel if it is either greater than the upper threshold, or greater than the lower threshold and connected to a pixel which is greater than the upper threshold. This technique is known as hysteresis thresholding and it can be applied recursively once a strong edge pixel is found by looking into its eight neighbours and determining weather they are candidate pixels. If one of them is a candidate pixel, then it is set to a strong edge and we look for neighbouring pixels. This process is continued until we find a candidate pixel for which

no neighbours are candidates. In his 1986 paper, Canny suggested to use a high threshold three times the lower.

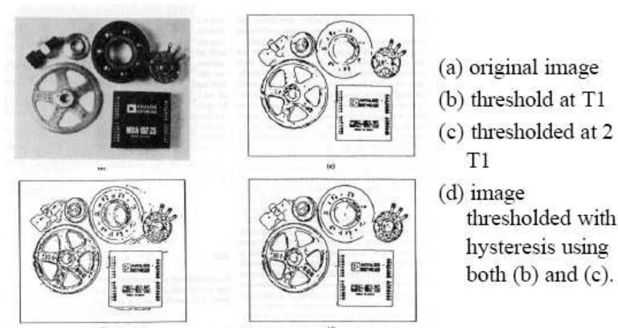


Figure11: different ouputs for the same image at different thresholds

The hysteresis threshold explanation concludes our discussion about edge detection procedures and, in particular, about the knowledge required for a complete implementation of the Canny edge detection, which may be found in the Appendix. We shall next turn our attention to the development of cartoon looking shading models.

V.- Cel shading models

V.i.- Standard shading models

As we have mentioned in previous section, the most important thing to consider for achieving a cartoon looking render is that objects appear flattened. In standard shading models, such the Phong or Lambert models, light calculations are performed over a cone defined by an angle of 90º about the facing normal of the surface.

```
Nn = normalize(N);
illuminance( P, Nn, PI/2 )
{
    Ln = normalize(L);
    Ci += Cs * Cl * Ln.Nn;
}
```

These light calculations are performed in Renderman with the `illuminance` construct specified above. The `illuminance` construct loops through all the lights in the scene and calculates their contribution to the point on the surface being shaded. Since it is sensible not to consider lights behind the point being shaded, only light falling on the hemisphere defined at point `P` will be considered.

This implies that points on the surface lying on a region where no light influence falls in the hemisphere defined by its facing normal will have no illumination whatsoever. The result is that they will appear black in a rendered image. Those points lying between an area of full absence of light and an area of perfect light influence will be shaded gradually according to their proximity to one or the other. This effect reproduces lighting conditions in real life and is precisely the shading characteristic that allow us to acquire a perception of three-dimensionality from the rendered object. For a cartoon looking render, this effect should be avoided.

Figure12: Phong shaded sphere rendered with maya

V.ii.- Gooch shading model

The area where we should focus here is that of non-photorealistic rendering, NPR. As mentioned in previous sections, Gooch developed a shading model in which a warm yellow-like colour is used for lighted areas and a cool blue-like one for the shadowed ones. The effect obtained by applying a Gooch shader to a skeleton is shown in figure13.

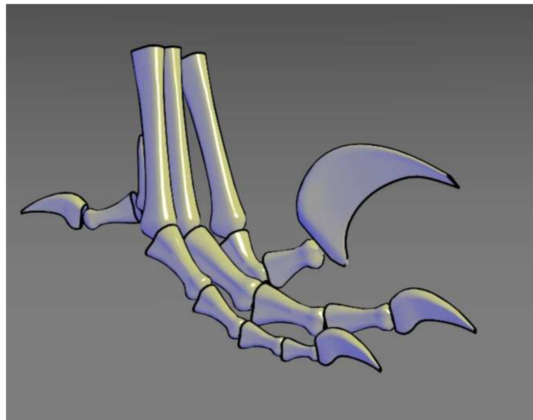


Figure13: Gooch shading model applied to a skeleton

The overall look of the surface appears now more flattened than with traditional shading models. The development of cartoon looking shaders in here has been inspired by the Gooch shading model. Instead of using cool-to-warm colours, surface colour has been modified by considering a darker colour for shadowed areas and a lighter colour for lighted ones. The following statement shows the usual way of assigning colour to a surface point in a realistic shader:

```
Ci = Os * Cs * ( Ka * ambient( ) + Kd * diffuse(Nf) +
                            Ks * specular(Nf, V, roughness));
```

In the simplest version of the Gooch shading model, the `specular` function is ignored and the `diffuse` function is replaced by an `illuminance` statement of the following form

```
normal Nf = faceforward(normalize(N),I),
illuminance(P,Nf,PI) {
    ldotn = (normalize(L)).Nf;
    blendval = 0.5*(1+ldotn);
    finalcolor += mix(cool,warm,blendval);
}
```

The effect of this model was shown in figure13. The first thing to notice is that the `illuminance` construct is considered over a whole sphere on the point being shaded. Thus, any light, whether is behind or in front of the point, will be considered during the shading process, thus cool colouring areas that would appear black with a standard shading model. The second thing to notice is that there is no light colour contribution `Cl` to the output colour `Ci`. This has the implication that the object being shaded will actually not receive shadows from other objects, since there is no way of telling the light colour at this point as specified with a shadow map. Gooch's shading model is mostly used in technical illustrations, where a single light is used and no other objects are present in the scene. In more elaborated Gooch models, white highlights are incorporated so as to give some information about object orientation (notice the highlights in figure13).

V.iii.- Cartoon shading

For a cartoon looking render, it is important to develop a shading model that incorporates shadowed areas arising from objects being placed between the light source and itself. In this way, Gooch model has been modified so as to incorporate light colour `Cl`. A second modification has been made on the actual output colour. Instead of using cool-to-warm tones, the output colour `Ci` has been given a value that is proportional to the input colour `Cs` in two different ways: a darker tone for shadowed areas and a lighter

tone for lighted ones. Furthermore, we seek a more abrupt transition between shadowed and lighted regions.

```
color lightedcolor = Cligh * Cs;
color shadowedcolor = Cshad * Cs;
color brightness = color(0,0,0);
normal Nf = faceforward(normalize(N),I),
illuminance(P,Nf,PI) {
    dot = (normalize(L)).Nf;
    val = 0.5*(1+dot);
    lightcolor = Kl * Cl;
    in = smoothstep(0.49, 0.51, val);
    brightness += mix(shadowedcolor,lightcolor*lightedcolor,in);
}
```

Once the dot product between the normalized light vector and the forward facing normal vector is calculated, it is normalized so that its value lies between `0` and `1`. This means that points in the dark, where the dot product is negative, will now lie between `0` and `0.5`. We can use this new value `val` as in the `mix` statement in Gooch's model, this time considering the light colour `Cl`. In order to achieve the sharp transition, but not too much, a smooth transition is applied to the value of `val` between the value it should take in dark regions and that in lighted regions. Therefore, as smoothstep function between `0.45` and `0.55` is used in order to acquire a new value `in` that will determine the final colour contribution to `brightness`. Note a new light colour `lightcolor` is used as a fraction of `Cl` in order to gain control over the contribution of light to the colour surface. The light contribution is then added to the `lightedcolor` only, producing different layers of luminosity in the presence of several lights while still keeping a fairly darkened colour for the shadowed regions. Finally, the surface colour computed in the `illuminance` loop is passed onto the output colour `Ci` as a diffuse contribution.

```
Ci = Os * ( Cs * (Ka * ambient()) + Kd * brightness);
```

These specifications have been applied to each developed shader. For a metallic and plastic look, two more statements have been added: a highlight is created by considering those points facing directly the light source. For the plastic shader, the highlighted

colour is a fraction of the surface colour, whereas for metallic objects this colour has been given a value of 1,white. This kind of effect is observed in two dimensional animations, where metallic objects appear white where a highlight should be encountered.

In order to enhance the scene, two specific shaders have been developed, which can be found in the Appendix. A brick wall pattern has been used for the building, and a wood floor pattern for the floor in order to further enhance the final animation. Inspiration for the development of these two shaders has been taken from already existing ones.

V.- Conclusion

This project has been focused on the study of the rendering of three-dimensional scenes into a cartoon looking style. It is observed in 2D animations that objects appear as defined by outlines filled with a flattened colour. In order to achieve a similar look for the render of objects from a three dimensional scene, two main questions have been addressed: how to draw object outlines and how to achieve such a flattened colouring.

We have seen that the first question can be solved as an edge detection problem, where existing algorithms defined in either object space and image space methods present both advantages and disadvantages. While image space algorithms are easier to implement, they also give rise to a loss of geometrical information that cannot be recovered from the two dimensional image. This loss of information has not been considered important for the chosen style of cartoon looking render that we wanted to achieve here. However, for a more hand-drawn oriented and style-flexible look, a way for retrieving geometrical information from the two dimensional scene is needed.

Although the motivations for choosing Canny's algorithm for our implementation was its well known reputation as an optimal edge detector, the results were not as expected. The low error rate that was thought to ensure frame coherence did not output the same edges in each frame. Supposedly, this is due to the fact that the edges that will be discarded or picked during hysteresis highly depend on the used threshold but also on the actual detail of the image, which is observed to change from frame to frame possibly due to the presence of image textures and the use of a minimal σ in order not to wash out the animation. This unexpected result has however given to the animation a stronger hand-look than the one we would have obtained by perfect edge localization. It is therefore considered that the waviness on edge localisation from frame to frame enhances the final animation.

Finally, the objects for the scene were modelled in Maya and shaded using developed cartoon looking Renderman shaders. A major problem was encountered at a late development stage, where shaded objects appeared self-shadowed but not receiving shadows cast by objects surrounding them. This problem was overcome by the addition

of light colour contribution to the final shading colour. Overall, a fairly cartoon looking animation has been produced and this should be considered the achievement of this project.

I believe a more flexible cartoon looking style relies on the geometrical information that is lost in the rendered image. I therefore consider that one interesting subject of study would be to develop a method by which this geometrical information could be kept in the renders.

Refrences

[1] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein and John F. Hughes, “*Real-Time Nonphotorealistic Rendering*”, Computer Graphics (Proc. Siggraph), ACM SIGGRAPH, ACM Press, 1997.

[2] John W. Buchanan, “*The edge buffer: a data structure for easy silhouette rendering*”. In Proceedings of the first international symposium on Non-photorealistic animation and rendering, ACM SIGGRAPH.

[3] Ramesh Raskar and Michael Cohen. “*Image Precision Silhouette Edges*”. In Proc. 1999 ACM Symposium on Interactive 3D Graphics, April 1999.

[4]Barbara Meier, “Painterly Rendering for Animation”, Computer Graphics (Proc. Siggraph), 1996

[5] Takafumi Saito and Tokiichiro Takahashi, ”*Comprehensible Rendering of 3D Shapes*”, Computer Graphics (Proc. Siggraph), Vol. 24, No. 4, ACM SIGGRAPH, ACM Press, August 1990.

[6] Winkenbach, G. and Salesin, D.H. (1994). “*Computer generated pen-and-ink illustration*”. In SIGGRAPH 94 Conference Proceedings.

[7] Philippe Decaudin. Cartoon-Looking Rendering of 3D-Scenes. Technical Report 2919, INRIA, June 1996.

[8] Amy, Bruce Gooch, and Mass Natick. “*Non-photorealistic rendering*”. A K Peters 2001.

[9] Alan Watt, “*The computer image*”, Addison-Wesley, 1998.

[10] John Canny. ”*A computational approach to edge detection*”. IEEE Transactions on Pattern Analysis and Machine Intelligence., Vol.8, No.6, IEEE Computer Society, 1986.

[11] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. "*A Non-photorealistic Lighting Model for Automatic Technical Illustration.*" Computer Graphics (Proc. Siggraph), ACM SIGGRAPH, July 1998.

[12] Steve Upstill. "*The Renderman companion: a programmer's guide to realistic computer graphics*". Addison.Wesley, 1990.

[13] Ian Stephenson, "*Essential RenderMan fast*", Springer, 2002.

[14] "*Digital image processing and analysis*", Vol. 2. IEEE Computer Society Press, 1985.

[15] Thomas Strothotte, "*Non-photorealistic computer graphics: modelling, rendering and animation*", Morgon Kaufmann, 2002.

[16] Jerry Beck, "*Animation art: from pencil to pixel, the world of cartoon, animé and CGI*", Flame Tree, 2004.

[17] Anthony Apodaca, Larry Gritz "*Advanced Renderman: creating CGI motion pictures*", Morgan Kauffman, 1999.

Appendix I: Code for Canny implementation

```
/* -------------- Canny's Algorithm implementation ----------
   1) Convolve the image with a separable gaussian filter.
   2) Take the dx and dy the first derivatives using [-1,0,1] and
[1,0,-1]'.
   3) Compute the magnitude: sqrt(dx*dx+dy*dy).
   4) Perform non-maximal suppression.
   5) Perform hysteresis.
*/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <signal.h>
#include <assert.h>
#include <NCCAPixmap.h>
#include <PixFileIO.h>

#define STRONG 0.0
#define CANDIDATE 0.5
#define WEAK 1.0
void step1_smooth_with_gaussian();
void step2_compute_gradient();
void step3_compute_magnitude();
void step4_non_maxima_suppression();
void step5_perform_hysteresis();
void edge_linking(short int lowval, int r, int c);

//GLOBAL VARIABLES
NCCAPixmap image;
NCCAPixmap edge;
unsigned char **nms; //non-maxim suppression
short int **smoothedim, //image after gaussian smoothing
                  **xderiv, //first devivative image, x-direction
                  **yderiv, //first derivative image, y-direction
                  **magnitude; //magnitude of the gadient image

int rows, cols;   //image dimensions and rows and columns counters
float sigma, //standard deviation of the gaussian kernel
                  tlow, //low threshold for hysteresis
                  thigh; //high threshold for hysteresis

main(int argc, char *argv[])
{
      NCCAPixmap orig;
   NCCAPixmap final;

      NCCAPixel p;
      float grey;
      int convert;

      int r, c;
  if(argc < 6){
            fprintf(stderr,"\nCanny usage %s image sigma tlow thigh
edge\n",argv[0]);
      fprintf(stderr,"\timage:\tAn image to process.\n");
```

```
      fprintf(stderr,"\tsigma:\tStandard     deviation     of     the
gaussian.\n");
      fprintf(stderr,"\ttlow:\tHigh threshold.\n");
      fprintf(stderr,"\tthigh:\tLow threshold");
      fprintf(stderr,"\tedge:\tThe edge image to obtain.\n");
      exit(1);
   }

   orig = loadPixmap(argv[1]);
   rows = orig.height;
   cols = orig.width;

   //convert image to greyscale
   image =newPixmap(cols,rows,1,8);
   for(r=0; r<rows; r++){
      for(c=0; c<cols; c++){
            p = getPixel(orig,c,r);
                  grey = p.r;
                  setPixelGrey(image,c,r,grey);
            }
      }

      sigma = atof(argv[2]);
      tlow = atof(argv[3]);
      thigh = atof(argv[4]);

      edge = newPixmapLike(image);

      //allocate space for the smoothedim data
      smoothedim = (short int**)calloc(rows,sizeof(short int*));
      if(smoothedim == NULL){
            fprintf(stderr,  "Error  allocating  the  smoothed  rows
image.\n");
      exit(1);
      }
      else{
            for(r=0; r<rows; r++){
                  smoothedim[r] = (short int*)calloc(cols,sizeof(short
int));
                  if(smoothedim[r] == NULL){
                  fprintf(stderr, "Error allocating the smoothed cols
image.\n");
                  exit(1);
                  }
            }
      }

      step1_smooth_with_gaussian();

      //allocate space for gradient images
      xderiv = (short int **) calloc(rows,sizeof(short int*));
      if(xderiv == NULL){
            fprintf(stderr,  "Error  allocating  the  x  derivative
image.\n");
            exit(1);
      }
      else{
            for(r=0; r<rows; r++){
                  xderiv[r]  =  (short  int*)  calloc(cols,sizeof(short
int));
                  if(xderiv[r] == NULL){
```

```
                        fprintf(stderr,   "Error   allocating   the   x
derivative image.\n");
                        exit(1);
                  }
            }
      }

      yderiv = (short int **) calloc(rows,sizeof(short int*));
      if(yderiv == NULL){
            fprintf(stderr,  "Error  allocating  the  y  derivative
image.\n");
      exit(1);
      }
      else{
            for(r=0; r<rows; r++){
                  yderiv[r]  =  (short  int*)  calloc(cols,sizeof(short
int));
                  if(yderiv[r] == NULL){
                        fprintf(stderr,   "Error   allocating   the   y
derivative image.\n");
                        exit(1);
                  }
            }
      }

      step2_compute_gradient();

      //allocate space for the magnitude image
      magnitude = (short int **) calloc(rows,sizeof(short int*));
      if(magnitude == NULL){
            fprintf(stderr,   "Error   allocating   the   magnitude
image.\n");
      exit(1);
      }
      else{
            for(r=0; r<rows; r++){
                  magnitude[r] = (short int*) calloc(cols,sizeof(short
int));
                  if(magnitude[r] == NULL){
                        fprintf(stderr,   "Error   allocating   the
magnitude image.\n");
                        exit(1);
                  }
            }
      }
      step3_compute_magnitude();

      //allocate space for the non-maxima output
      nms = (unsigned char **) calloc(rows,sizeof(unsigned char*));
      if(nms == NULL){
            fprintf(stderr,   "Error   allocating   the   magnitude
image.\n");
      exit(1);
      }
      else{
            for(r=0; r<rows; r++){
                  nms[r]        =        (unsigned        char*)
calloc(cols,sizeof(unsigned char));
                  if(nms[r] == NULL){
                        fprintf(stderr,   "Error   allocating   the
magnitude image.\n");
```

```
                        exit(1);
                  }
            }
      }

      step4_non_maxima_suppression();

      step5_perform_hysteresis();

      savePixmap(edge, argv[5]);

      //free the pointers
      for(r=0; r<rows; r++){
            free(smoothedim[r]);
            free(xderiv[r]);
            free(yderiv[r]);
            free(magnitude[r]);
            free(nms[r]);
      }
      free(smoothedim);
      free(xderiv);
      free(yderiv);
   free(magnitude);
   free(nms);
}

void step1_smooth_with_gaussian()
{
      int r, c, rr, cc,//counters
            dimension, //dimension of the gaussian mask
            center; //center of gaussian mask
      float  **tempim,//buffer  to  separate  gaussian  mask  in  x  and  y
directions
                  greyval,
                  prod,
                  sum,
                  x,
                  Gx,
                  *gaussmask;

      //compute the gaussian kernel
      //set dimensions accorind to the value of sigma
      dimension = 1 + 2 * ceil(2.5 * sigma);

      gaussmask = (float*)calloc(dimension,sizeof(float));
      if(gaussmask == NULL){
            fprintf(stderr, "Error allocating the Gaussian mask.\n");
            exit(1);
      }

   center = dimension/2;
   sum = 0.0;
   for(r=0;r<dimension; r++){
      x = (float)(r - center);
      //gaussian function in one dimension
      Gx   =   pow(2.718281828,   -0.5*x*x/(sigma*sigma))   /   (sigma   *
sqrt(6.2831853));
      gaussmask[r] = Gx;
      sum += Gx;
   }
```

```
   for(r=0;r<dimension;r++) gaussmask[r] /= sum;

      //allocate space for the temporary buffer
      tempim = (float**)calloc(rows,sizeof(float*));
      if(tempim == NULL){
            fprintf(stderr,  "Error  allocating  the  buffer  rows
image.\n");
            exit(1);
      }
      else{
            for(r=0; r<rows; r++){
                  tempim[r] = (float*)calloc(cols,sizeof(float));
                  if(tempim[r] == NULL){
                  fprintf(stderr, "Error allocating the buffer cols
image.\n");
                  exit(1);
                  }
            }
      }

      //smooth in the x direction
      for(r=0;r<rows;r++){
            for(c=0;c<cols;c++){
                  prod = 0.0;
                  sum = 0.0;
                  for(cc=(-center);cc<=center;cc++){
                        if(((c+cc) >= 0) && ((c+cc) < cols)){
                              greyval = getPixelGrey(image,c+cc,r);
                              prod += greyval* gaussmask[center+cc];
                              sum += gaussmask[center+cc];
                        }
                  }
                  tempim[r][c] = prod/sum;
            }
      }

      //smooth in the y direction
      for(c=0;c<cols;c++){
            for(r=0;r<rows;r++){
                  prod = 0.0;
                  sum = 0.0;
                  for(rr=(-center);rr<=center;rr++){
                        if(((r+rr) >= 0) && ((r+rr) < rows)){
                              prod      +=      tempim[r+rr][c]      *
gaussmask[center+rr];
                              sum += gaussmask[center+rr];
                        }
                  }
                  smoothedim[r][c] = prod/sum;
            }
      }

      for(r=0; r<rows;r++)
            free(tempim[r]);
   free(tempim);
   free(gaussmask);
}


//use 1D Prewitt operator for caluclating gradient image
```

```
void step2_compute_gradient()
{
	int r, c, pos;

	//gradient along x direction
	for(r=0;r<rows;r++){
		xderiv[r][0] = smoothedim[r][1] - smoothedim[r][0];
		for(c=1;c<(cols-1);c++){
			xderiv[r][c] = smoothedim[r][c+1] - smoothedim[r][c-
1];
		}
		xderiv[r][cols-1]      =      smoothedim[r][cols-1]      -
smoothedim[r][cols-2];
	}

	//gradient along y direction
	for(c=0;c<cols;c++){
		yderiv[0][c] = smoothedim[1][c]-smoothedim[0][c];
		for(r=1;r<rows-1;r++){
			yderiv[r][c]  =  smoothedim[r+1][c]  -  smoothedim[r-
1][c];
		}
		yderiv[rows-1][c] = smoothedim[r-1][c] - smoothedim[rows-
2][c];
	}
}

//compute magnitude
void step3_compute_magnitude()
{
	int r, c, pos, gx, gy;
	for(r=0;r<rows;r++){
	for(c=0;c<cols;c++){
			gx = xderiv[r][c] * xderiv[r][c];
			gy = yderiv[r][c] * yderiv[r][c];
			magnitude[r][c] = sqrt(gx + gy);
		}
	}
}

//apply non-maxima suppression
void step4_non_maxima_suppression()
{
	int rowcount, colcount,count;
	int r, c;
	short mag,gx,gy, ux, uy;
	float p1, p2, magA,magB,xperp,yperp;

	//assign zeros to the borders of the final image
	for(c = 0; c<cols; c++){
		nms[0][c] = (unsigned char) 0;
		nms[rows-1][c] = (unsigned char) 0;
	}

	for(r=0; r<rows; r++){
		nms[r][0] = (unsigned char) 0;
		nms[r][cols-1] = (unsigned char) 0;
	}

	for(r=1; r<rows-2; r++){
		for(c=1; c<cols-2; c++){
```

```
			mag = magnitude[r][c];
			if(mag == 0){
				nms[r][c]= WEAK;
		}
			else{
				gx = xderiv[r][c];
				ux = -gx/mag;
				gy = yderiv[r][c];
				uy = gy/mag;
			}

			//deal with each case
			if(gx >= 0){
				if(gy >= 0){
					if (gx >= gy)
						{
							p1 = magnitude[r][c-1];
							p2 = magnitude[r-1][c-1];
							magA = (mag - p1)*ux + (p2 -
p1)*uy;
					 p1 = magnitude[r][c+1];
							p2 = magnitude[r+1][c+1];
							magB = (mag - p1)*ux + (p2 -
p1)*uy;
						}
						else
						{
							p1 = magnitude[r-1][c];
							p2 = magnitude[r-1][c-1];
							magA = (p1 - p2)*ux + (p1 -
mag)*uy;
							p1 = magnitude[r+1][c];
							p2 = magnitude[r+1][c+1];
							magB = (p1 - p2)*ux + (p1 -
mag)*uy;
						}
					}
					else
					{
						if (gx >= -gy)
						{
							p1 = magnitude[r][c-1];
							p2 = magnitude[r+1][c-1];
							magA = (mag - p1)*ux + (p1 -
p2)*uy;
							p1 = magnitude[r][c+1];
							p2 = magnitude[r-1][c+1];
							magB = (mag - p1)*ux + (p1 -
p2)*uy;
						}
						else
						{	p1 = magnitude[r+1][c];
							p2 = magnitude[r+1][c-1];
							magA = (p1 - p2)*ux + (mag -
p1)*uy;
							p1 = magnitude[r-1][c];
							p2 = magnitude[r-1][c+1];
							magB = (p1 - p2)*ux   + (mag
- p1)*uy;
						}
					}
```

```
                        }
                        else
                        {
                              gy = yderiv[r][c];
                              if (gy >= 0)
                              {
                                    if (-gx >= gy)
                                    {
                                          p1 = magnitude[r][c+1];
                                          p2 = magnitude[r-1][c+1];
                                          magA = (p1 - mag)*ux + (p2 -
p1)*uy;
                                          p1 = magnitude[r][c-1];
                                          p2 = magnitude[r+1][c-1];
                                          magB = (p1 - mag)*ux + (p2 -
p1)*uy;
                                    }
                                    else
                                    {
                                          p1 = magnitude[r-1][c];
                                          p2 = magnitude[r-1][c+1];
                                          magA = (p2 - p1)*ux + (p1 -
mag)*uy;
                                          p1 = magnitude[r+1][c];
                                          p2 = magnitude[r+1][c-1];
                                          magB = (p2 - p1)*ux + (p1 -
mag)*uy;
                                    }
                              }
                              else
                              {
                                    if (-gx > -gy)
                                    {
                                          p1 = magnitude[r][c+1];
                                          p2 = magnitude[r+1][c+1];
                                          magA = (p1 - mag)*ux + (p1 -
p2)*uy;
                                          p1 = magnitude[r][c-1];
                                          p2 = magnitude[r-1][c-1];
                                          magB = (p1 - mag)*ux + (p1 -
p2)*uy;
                                    }
                                    else
                                    {
                                          p1 = magnitude[r+1][c];
                                          p2 = magnitude[r+1][c+1];
                                          magA = (p2 - p1)*ux + (mag -
p1)*uy;
                                          p1 = magnitude[r-1][c];
                                          p2 = magnitude[r-1][c-1];
                                          magB = (p2 - p1)*ux + (mag -
p1)*uy;
                                    }
                              }
                        }

                        //is the current point a maximum point?
                        if ((magA > 0.0) || (magB > 0.0)){
                              nms[r][c] = WEAK;
                        }
                        else
```

```
                        {
                               if (magB == 0.0){
                                    nms[r][c] = WEAK;}
                              else{
                                    nms[r][c] = CANDIDATE;}
                        }
                  }
            }

}

//hysteresis
void step5_perform_hysteresis()
{
      int  r,  c,  pos,  numedges,  lowcount,  highcount,  lowthreshold,
highthreshold,
            i, hist[32768], rr, cc;
      short int maximum_mag, sumpix;
      float p;
      int pint;

      //initialise the edge image
      for(r=0; r<rows; r++){
            for(c=0; c<cols; c++){
                  if((r==0)||(c==0)||(r==(rows-1))||(c==(cols-1))){
                        p = WEAK;
                        setPixelGrey(edge,c,r,p);
                  }
                  else{
                        if(nms[r][c] == CANDIDATE){
                              p = CANDIDATE;
                              setPixelGrey(edge,c,r,p);
                        }
                        else{
                              p = WEAK;
                              setPixelGrey(edge,c,r,p);
                        }
                  }
            }
      }

      for(r=0;r<rows;r++){
            for(c=0;c<cols;c++){
                  p = getPixelGrey(edge,c,r);
                  if((pint==CANDIDATE) && magnitude[r][c]>=thigh){
                        p = STRONG;
                        setPixelGrey(edge,c,r,p);
                        edge_linking(tlow,r,c);
                  }
            }
      }

      //set all the edges that are not connected to oher edges to weak
      for(r=0; r<rows; r++){
            for(c=0;c<cols;c++){
                  p = getPixelGrey(edge,c,r);
                  if(p!=STRONG){
                        p = WEAK;
                        setPixelGrey(edge,c,r,p);
                  }
            }
```

```
	}
}

//edge linking
void edge_linking(short int t, int r, int c)
{
	float p;
	int u, v;
	for(u=-1;u<1;u++){
		for(v=-1; v<1; v++){
				if((u!=0)&&(v!=0)){
					p = getPixelGrey(edge,c+v,r+u);
					if((p       ==       CANDIDATE)       &&
(magnitude[r+u][c+v]>t)){
						p = STRONG;
						setPixelGrey(edge,c+v,c+u,p);
						edge_linking(t, r+u, c+v);
				}
			}
		}
	}
}
```

Appendix II: Code for cel shading models

```
/*------------ celBrick.sl ---------------*/
/*
Surface shader for a cartoon looking wall brick pattern.
Params:
      Ka, Kd: standard ambient and diffuse components
      Kl: fraction of Cl contributing to lighted color
      Cshad: fraction of Cs contributing to the shadowed color
      Cligh: fraction of Cs contributing to the lighted color
      Y: grey component of Ci used in edge detection
      irregularity: how dented or uneven the brick surface appears
      brickwidth: width of the brick
      brickheight: height of the brick
      mortarwidth: with of the mortar
      mortardepth: the bumpiness of the mortar
      texturescale: overall scale of the applied pattern
      brickcolor: standard color for the brick
      mortarcolor: standard color for the mortar
      colorvariation: deviation from the standard color from brick to
brick*/

surface
celBrick(
      //standard variables
      float Ka = 1,
            Kd = 1,
            Kl = 1,
            Cshad = 0.25,
            Cligh = 0.5;
      output varying float Y = 0;
      //physical variables
      float irregularity = 0.06,
            brickwidth = 0.25,
            brickheight = 0.12, //give always even values here
for it to work
            mortarwidth = 0.05,
            texturescale = 2;
      //color variables
      color brickcolor = color "rgb" (0.83,0.01,0.12),
            mortarcolor = color "rgb" (0.5,0.5,0.5);
```

```
	float		colorvariation = 0.5;
)
{
	#define snoise(x) (2 * noise((x)) - 1)

	normal Nf = faceforward( normalize( N ), I );
	vector V = -normalize( I ) ;

	//brick shape variation
	point PP = point noise ( u/brickwidth, v/brickheight );
	float ss = s + irregularity * xcomp ( PP );
	float tt = t + irregularity * ycomp ( PP );

	//divide by brick dimensions so that ss and tt vary from 0 to 1
within a single brick.
	ss = ( texturescale * ss )/brickwidth;
	tt = ( texturescale * tt )/brickheight;

	//identify which brick contains the point being shaded
	float brickrow = floor( tt );

	//offset even rows by half a row
	if( mod( tt, 2 ) > 1 )
		ss += 0.5;

	//identify which brick contains the point being shaded
	float brickcol = floor( ss );

	//texture coords within brick
	ss -= brickcol;
	tt -= brickrow;

	//choose brick color that varies from brick to brick
	color Ct = brickcolor * ( 1 + colorvariation * ( noise(
(brickcol * brickrow)/2.3 + 3.7 )));
	//detrmine if in mortar for mortar color
	if( ss < mortarwidth || tt < mortarwidth )
		Ct = mortarcolor;

	//cartoon shading color declaration and initialisation
	color lightedcolor = Cligh * Ct;
```

```
color shadowedcolor =Cshad * Ct;
color brightness = color( 0, 0, 0 );
color lightcolor = color( 0, 0, 0 );

//auxiliary variables for illuminance loop
float dot, val, instep;

//illuminance loop at P with a cone of 360 degrees, considering
Nf for illumination
illuminance( P ){
      //see difference in angle
      dot = ( normalize( L ) ).Nf;
      //normalize so that it lies between 0 and 1
      val = 0.5 * ( 1 + dot );
      //consider lightcolor a fraction of the light color
      lightcolor = Kl * Cl;
      //increase brightness after increasing the lightedcolor by
the lightcolor
      //accumulate light by mixing shadowedcolor and
lightedcolor depending on the angle between the facing normal vector
Nf and the light vector L
      brightness += mix( shadowedcolor, lightcolor *
lightedcolor, val );
}

//detrmine if in mortar for mortar color
//if( ss < mortarwidth || tt < mortarwidth )
//    Ct = mcolor;

Oi = Os;
Ci = Oi * ( Ct * ( Ka * ambient() )
                  + Kd * brightness );



//detrmine if in mortar for mortar color
//if( ss<mortar || tt<mortar )
      //Ci = mcolor;



// -- Second method for colouring --
```

```
/*     float smwidth=smoothstep(0,mortar/2,ss);
       float tmwidth=smoothstep(0,mortar/2,tt);
       float mwidth=smwidth*tmwidth;
       color brick=mix(mcolor,bcolor,mwidth);
*/

       //calculate final color for the brick
       //Ci = brick*Os * (Ka*ambient() + Kd*diffuse(Nf));
        Y = comp(ctransform("yiq", Ci), 0) ;
}
```

```
/*

------------ celMatte.sl ---------------

*/
/*
Params:
      Ka, Kd: ambient, diffuse coefficients
      Kl: fraction of Cl contributing to lighted color
      Cshad: fraction of Cs contributing to the shadowed color
      Cligh: fraction of Cs contributing to the lighted color
      Y: grey component of Ci used in edge detection
*/

surface
celMatte (
      float Ka = 1,
                  Kd = 1,
                  Kl = 1,
                  Cshad = 0.25,
                  Cligh = 0.5;
      output varying float Y = 0;
)
{
    normal Nf = faceforward( normalize( N ), I );

      //cartoon shading color declaration and initialisation
      color lightedcolor = Cligh * Cs;
      color shadowedcolor = Cshad * Cs;
      color brightness = color ( 0, 0, 0 );
      color lightcolor = color( 0, 0, 0 );

      //auxiliary variables for illuminance loop
      float dot, val, instep;

      //illuminance loop at P with a cone of 360 degrees, considering
Nf for illumination
      illuminance( P ){ //same as illuminance(P,Nf,PI)
            //see difference in angle
            dot = ( normalize( L ) ).Nf;
            //normalize so that it lies between 0 and 1
            val = 0.5 * ( 1 + dot );
            //consider lightcolor a fraction of the light color
            lightcolor = Kl * Cl;
            instep = smoothstep( 0.49, 0.51, val );
            //increase brightness after increasing the lightedcolor by
the lightcolor
            //accumulate light by mixing shadowedcolor and
lightedcolor depending on the angle between the facing normal vector
Nf and the light vector L
            brightness += mix( shadowedcolor, lightcolor *
lightedcolor, val );
      }

      //opacity
      Oi = Os;
      //color: Cs contribution to ambient only, since brightness
contains already Cs. brightness contribution to diffuse, since diffuse
has been replace essentially by the loop above
```

```
    Ci = Oi *( Cs * ( Ka * ambient() )
            + Kd*brightness );
      //YIQ component 0 component - grey
      Y = comp(ctransform("yiq", Ci), 0) ;
}
```

```
/*------------ celMetalic.sl ---------------*/
/* cartoon looking surface shader simulating the effect of light on a
metalic object
Params:
      Ka, Kd: ambient, diffuse coefficients
      Kl: fraction of Cl contributing to lighted color
      Cshad: fraction of Cs contributing to the shadowed color
      Cligh: fraction of Cs contributing to the lighted color
      Y: grey component of Ci used in edge detection
*/


surface
celMetalic(
      float       Ka = 1,
                  Kd = 1,
                  Kl = 1,
                  Cshad = 0.25,
                  Cligh = 0.5;
       output varying float Y = 0;
       )
{
    normal Nf = faceforward( normalize( N ), I );

      //cartoon shading color declaration and initialisation
      color lightedcolor = Cshad * Cs;
      color shadowedcolor = Cligh * Cs;
      color brightness = color( 0, 0, 0 );
      color lightcolor = color( 0, 0, 0 );
      color white = color( 1, 1, 1);
      //initial specularity
      float spec = 0;

      //auxiliary variables for illuminance loop
      float dot, val, instep;

      //illuminance loop at P with a cone of 360 degrees, considering
Nf for illumination
      illuminance( P ){
            //see difference in angle
            dot = ( normalize( L ) ).Nf;
            //normalize so that it lies between 0 and 1
            val = 0.5 * ( 1 + dot );
            //specular highlight if dot product very close to one
            if( val > 0.975 ) spec += val * val;
            //consider lightcolor a fraction of the light color
            lightcolor = Kl * Cl;
            //increase brightness after increasing the lightedcolor by
the lightcolor
            //accumulate   light   by   mixing   shadowedcolor   and
lightedcolor depending on the angle between the facing normal vector
Nf and the light vector L
            brightness   +=   mix(   shadowedcolor,   lightcolor   *
lightedcolor, val );
      }

      //opacity
      Oi = Os;

      //produce white specular highlight imitating cartoon technique
      if(spec!=0){
```

```
            float instep = smoothstep(0.97, 0.99, spec);
            Ci = Oi * mix( brightness, white, instep );
      }
      else
            Ci = Oi * ( Cs * ( Ka * ambient() ) + Kd * brightness );

      Y = comp(ctransform("yiq", Ci), 0) ;
}
```

```
/*------------ celPlastic.sl ---------------*/
/* cartoon looking surface shader simulating the effect of light on a
plastic object
Params:
      Ka, Kd: ambient, diffuse coefficients
      Kl: fraction of Cl contributing to lighted color
      Cshad: fraction of Cs contributing to the shadowed color
      Cligh: fraction of Cs contributing to the lighted color
      Y: grey component of Ci used in edge detection
*/

surface
celPlastic(
       float      Ka = 1,
                  Kd = 1,
                  Kl = 1,
                  Cshad = 0.25,
                  Cligh = 0.5,
                  Cspec = 1;
       output varying float Y = 0;
)
{
    normal Nf = faceforward( normalize( N ), I );

      //cartoon shading color declaration and initialisation
      color lightedcolor = Cligh * Cs;
      color shadowedcolor =Cshad * Cs;
      //make the specular color the same as Cs
      color specularcolor = Cspec * Cs;
      color brightness = color( 0, 0, 0);
      float highlight = 0;
      color lightcolor = color( 0, 0, 0);

      //auxiliary variables for illuminance loop
      float dot, val, instep;
      float count = 1;

      //illuminance loop at P with a cone of 360 degrees, considering
Nf for illumination
      illuminance( P ){
            //see difference in angle
            dot = ( normalize( L ) ).Nf;
            //normalize so that it lies between 0 and 1
            val = 0.5 * ( 1 + dot );
            //specular highlight if dot product very close to one
            if(val>0.925) highlight += val * val;
            lightcolor = Kl * Cl;
            specularcolor += Cspec * lightcolor;
            instep = smoothstep( 0.49, 0.51, val);
            //increase brightness after increasing the lightedcolor by
the lightcolor
            //accumulate   light   by   mixing   shadowedcolor   and
lightedcolor depending on the angle between the facing normal vector
Nf and the light vector L

            brightness    +=    mix(    shadowedcolor,    lightcolor    *
lightedcolor, val);
      }

      //opacity
      Oi = Os;
```

```
        if(highlight!=0){
              instep = smoothstep(0.94, 0.985, highlight);
        Ci = mix( brightness, specularcolor, instep);
        }
        else
        Ci = Oi * ( Cs * ( Ka * ambient() ) + Kd * brightness );

        Y = comp(ctransform("yiq", Ci), 0) ;
}
```

```
/* ----------- celWood.l ---------------*/
/* cartoon looking surface shader simulating a tiled wooden floor
Params:
      Ka, Kd: ambient, diffuse coefficients
      Kl: fraction of Cl contributing to lighted color
      Cshad: fraction of Cs contributing to the shadowed color
      Cligh: fraction of Cs contributing to the lighted color
      plankwidth: width of the plank
      spacewidth: width of the space between planks
      texturescale: overall scale factor for the pattern
      ringfactor: multiplicative factor for the ringsize
      wavefactor:  multiplicative  factor  for  the  waviness  on  the
distribution of the rings
      Y: grey component of Ci used in edge detection
*/


surface
celWoodShadows(
      //usual attributes
      float Ka = 1,
            Kd = 1,
            Kl = 1,
            Cshad = 0.25,
            Cligh = 0.5;
      //phyical attributes
      float       plankwidth = 0.2,
                  spacewidth = 0.02,
                  texturescale = 8,
                  ringfactor = 25,
                  wavefactor = 0.8;
      //color attribute
      color       woodcolor = color "rgb" ( 0.57, 0.292, 0.125 ),
                  ringcolor = color "rgb" ( 0.275, 0.15, 0.06 ),
                  spacecolor = color( 0, 0, 0 );
      float colorvariation = 0.5;
      //output
      output varying float Y = 0;
)
{

      float width = plankwidth;
      float height = plankwidth;

      #define snoise(x) (2 * noise((x)) - 1)
      #define MINFILTERWIDTH 1.0e-7

      normal Nf = faceforward( normalize( N ), I );


      //divide by plank dimensions so that ss and tt vary from 0 to 1
within a single brick.
      float ss = ( texturescale * s )/width;
      float tt = ( texturescale * t )/height;

      //identify which plank contains the point being shaded
      float plankrow = floor( ss );
      float plankcol = floor( tt );

      //twist the planks so that they are alternatively in horizontal
or vertical direction
```

```
	if( mod( plankrow + plankcol, 2 ) >= 1 ){
		float tmp;
		ss = ( texturescale * t )/width;
		tt = ( texturescale*s )/height;
		plankrow = floor( ss );
		plankcol = floor( tt );
	}

	//texture coords within plank
	ss -= plankrow;
	tt -= plankcol;

	//work out the ring adding several layers of noise
	float nnoise = tt/4 + plankcol + wavefactor * noise( 8 * ss, 
tt/4 );
	float r =  ringfactor * noise ( ss - plankcol, nnoise);
	r -= floor( r );
    r = 0.5 + 0.5 * smoothstep( 0.2, 0.55, r) * (1 - smoothstep(0.75, 
0.8, r));

	//mix between ringcolor and woodcolor
	color Ct = mix( ringcolor, woodcolor, r);

	//let the color change from plank to plank
	Ct *= (1 - colorvariation/2 + colorvariation * (snoise( plankrow 
* plankcol + 0.7 ) ) );

	if( ss < spacewidth/width || tt<spacewidth/height)
		Ct = spacecolor;

	color lightedcolor = Cligh * Ct;
	color shadowedcolor = Cshad * Ct;
	color brightness = color( 0, 0, 0);
	color lightcolor = color( 0, 0, 0);

	//auxiliary variables for illuminance loop
	float dot, val, instep;

	//illuminance loop at P with a cone of 360 degrees, considering 
Nf for illumination
	illuminance( P ){ //same as illuminance(P,Nf,PI)
		//see difference in angle
		dot = ( normalize( L ) ).Nf;
		//normalize so that it lies between 0 and 1
		val = 0.5 * ( 1 + dot );
		//consider lightcolor a fraction of the light color
		lightcolor = Kl * Cl;
		//increase brightness after increasing the lightedcolor by 
the lightcolor
		//accumulate   light   by   mixing   shadowedcolor   and 
lightedcolor depending on the angle between the facing normal vector 
Nf and the light vector L
		brightness   +=   mix(   shadowedcolor,   lightcolor   * 
lightedcolor, val );
	}


	Oi = Os;
    Ci = Oi * ( Ct * ( Ka * ambient() )
		+ Kd * brightness);
```

```
        //YIQ component 0 component - grey
        Y = comp(ctransform("yiq", Ci), 0) ;
}
```