Fast 3D Cartoon Rendering with Improved Quality by Exploiting Graphics Hardware

Johan Claes[†], Fabian Di Fiore[‡], Gert Vansichem[‡], Frank Van Reeth[‡]

[†] Department Mathematics & Informatics
Universitat de les Illes Balears
Cra. de Valldemossa, 07071 Palma
Spain
jclaes@studio1.uib.es

[‡] Expertise Centre for Digital Media
Limburg University Centre
Wetenschapspark 2, B-3590 Diepenbeek
Belgium
{fabian.difiore, gert.vansichem, frank.vanreeth}@luc.ac.be

Abstract

Our work in the area of non-photo-realistic rendering (NPR) focuses on visualising 3D models in a cartoon rendering style: the geometrical objects are internally coloured using two or three colours, while simultaneously using an explicit outline. Many NPR techniques are time-consuming processes, whereas we aim at real-time rendering performance on mainstream graphics PC hardware without losing a high image quality. Particular emphasis is also put on getting smooth transition borders between the bi-tonal cartoon-colours by applying an object precision triangle subdivision technique. Rapidly finding these transition borders as well as silhouette edges is a topic of a new algorithm. As silhouette lines and light effects are useful indicators for computer vision, this algorithm also looks interesting for applications in that field.

1. Introduction

Techniques originally meant for 3D computer graphics are starting to find their way into 2D animation. Cartoon-style rendering is typically something that is considered to be purely 2D. We show how 3D scenes can be rendered and animated effectively using adequate algorithms for this particular type of non-photo-realistic rendering.

As graphics hardware constantly increases in performance, while simultaneously becoming cheaper, we also want to exploit its power to the maximum. This helps in reaching our additional goal: getting top real-time execution speeds. Current graphics hardware has been optimised to perform standard tasks for Gouraud shaded z-buffered rendering, and we demonstrate how it can be effectively put to use for non-photo-realistic rendering such as cartoon-style rendering.

We would like to emphasise that the goal of this paper is to get real-time cartoon rendering performance on standard graphical hardware. Some cartoon-like effects can be obtained using a colour quantisation scheme or applying texture-mapping in unusual ways, sometimes even built into high-end graphics hardware. The solution we propose in the current paper works real-time with much less demanding hardware requirements. Our techniques are more flexible assigning individual colours to different surfaces, and our approach lends itself to obtaining nicely smoothed transition borders instead of either the staircase effects or too fuzzy borders produced by methods based on quantisation or texture-mapping.

In addition, we introduce a new algorithm for quickly locating all silhouette edges. Furthermore, this algorithm can be adapted to locate the borders between the cartoon-colours.

This paper's sections are organised as follows. Section 2 describes related work in the field while Section 3 explains the details of our rendering process. In Section 4 we describe an alternative approach for locating silhouette edges. Section 5 shows some of our examples, while Section 6 presents the conclusions of this paper, some thoughts on our future research, and a few acknowledgements.

2. Related Work

Philippe Decaudin was the first to specifically look at 3D cartoon rendering. In his Ph.D. dissertation [2] he extensively describes the particular problems and shows possible solutions, however in a far from real-time implementation. A much speedier approach is shown by [8], who is juggling with on-the-fly calculations of texture-co-ordinates in a one-dimensional texture. We used an approach similar to theirs

to generate cartoon-style images and animations, and succeeded in speeding up the process while at the same time improving the quality of the images.

To find and render the silhouette edges in a polygonal model, two solution categories have been proposed. The first encompasses image space algorithms using several rendering stages, while either modifying depths [13] or scaling polygons [12]. The main advantage of image space based methods is that they also work on unconnected polygon meshes without the need for pre-processing, so they keep working well when the polygons are dynamically recreated or deformed for each individual animation frame. Drawbacks are that, due to the multiple rendering stages, the process is slowed down and also they lack the freedom to control the silhouette, such as adding texture or drawing a smoothing curve over it.

Therefore many NPR researchers opt for algorithms based in object space to determine the silhouette edges. In contrast with the image space based approaches, here the process is not necessarily $O(n)$ – with n being the number of edges in the input model. Nevertheless, many researchers still opt for an $O(n)$ approach – even including a pre-processing step – in order to work with simplified data structures [1, 8]. If the model is fixed during the animation and connectivity information is present, it is possible to work out algorithms that are faster than $O(n)$. [9] implemented a stochastic search to quickly find many – but not necessarily all – silhouette edges. [5] shows a method based on the Gauss map to find these edges in a model with n edges and k silhouette edges in a time proportional to $O(k \log n)$, but limiting themselves to orthographic views. In order to cope with perspective views, [14] and [7] use a hierarchical tree of cones. A conversion to homogenous coordinates in a 4D space and afterwards projecting this back to eight 3D cubes is a rather complicated extension to [5], proposed by [6]. We opt to do all calculations in 4D, generalising Gooch's algorithm in a straightforward and elegant way and furthermore making it suited for light calculations.

3. Cartoon-style Rendering

In order to get a typical cartoon-style look, some specific means of expression are used. Silhouette lines, border lines and sharp edges (creases) are drawn in a thicker line style. Also, objects are filled with a limited number of colours. Please refer to section 5 to view some example images.

3.1. Input

As geometrical input, a triangular mesh is used, which is a very common approach in current renderers.

3.2. Silhouette Lines

For a given view direction, silhouette lines are drawn everywhere on the border of the visible part of an object. The silhouette for a cylinder, for example, consist usually of two parts of a circle and two straight lines. Whenever the point of view changes, the silhouette lines will be on a different position on the surface of the object.

We use a method that's suitable for most existing polygonal input. Although the method is $O(n)$ for n input polygons, the algorithm uses only a marginal portion of the calculation time. We should stress that the rendering of the object – even in a constant colour – is already necessarily $O(n)$ and that the number of silhouette edges can be quite high for finely detailed objects that are not too smooth.

Nevertheless, we have also developed a faster algorithm, which is suitable for larger objects or for a variation of the rendering style in which $O(n)$ would not be required. This algorithm is explained in section 4.

Our current approach is the following: the unordered input faces are organised in an efficient topological data structure in a pre-calculation step. That is, we build a winged-edge data structure, giving us for every edge its vertexes, the adjacent faces and the neighbouring edges. We refer to [4] for properties and possible ways of implementing this kind of data structures.

In a polyhedral model, a silhouette line is an edge that is connected to both a front-face and a back-face. (Front-faces are those faces of which the normal points towards the viewer, while back-faces point away from the viewer). Everywhere they meet, a piece of the silhouette can be added. At this point, our data structure is very useful: whenever we find an edge lying between a front and a back-face, we mark this edge as being part of the silhouette. By traversing the edges in an order related to the neighbourhood of edges already found, we are able to get the silhouette as connected chains; this would not be possible with other approaches [1]. The connected chains of silhouette edges are useful to correct rendering problems where two edges touch, or to add special effects like drawing a spline or an artistic texture to further enhance them [9].

3.3. Sharp Edges

We analyse the given input to calculate which edges are sharp. When the dihedral angle between two neighbouring faces is smaller than a user-supplied maximum, the edge is marked as sharp. Also edges that only connect to one face in the currently investigated mesh are marked as being sharp. Again our topological model helps to speed up the necessary searches. While silhouette edges need to be calculated again for each individual rendering, the sharp edges can be efficiently collected in a pre-processing step.

3.4. Limiting the Number of Colours

The most typical feature in a cartoon-style drawing is the limited use of colours.

We use the Gouraud shading model (see e.g. [3] chapter 16.2). This is a standard shading model for polygonal input managing with ambient light, diffuse and specular reflection and realises a smooth transition of colour.

In our cartoon renderer, the user chooses the amount of colours (usually two or three) for a given surface to be rendered. If no specular highlights are requested, two colours are chosen: one for the darker parts that get no or only a little direct light and one for the main parts in the light. If the user also asks for specular highlights, a third colour (that is very close to white) is added. The pre-chosen colours are created by our system and are based upon the original material properties. We call these pre-chosen colours the cartoon-colours. For each material, the user can overwrite our default-calculated cartoon-colours as well as the values that limit the border between the colours. Additionally, the user has the option to associate a texture to a cartoon-colour region.

The cartoon rendering is a two-step process: the values that are calculated in the pre-processing step are used in the final rendering step. In the pre-processing step, the scene is rendered via standard Gouraud shading. For every vertex of every visible face, a colour value is calculated. These intermediate values are stored to be used in the rendering step.

In the final rendering step, the colour for every face is determined. The simplest approach is giving the cartoon-colour that most closely matches the mean colour of the face to the entire face.

A drawback of this approach is that a staircase effect pops up (see figure 2). [2] suggests using a model with more faces to improve the image quality. But that reduces performance significantly and only diminishes the staircase effect without really solving it. Another solution is to use texture-mapping [8], but that approach is either too aliased or has too fuzzy borders between the cartoon-colours (see figure 5(c)). Moreover, in our solution we keep the texture-mapping capabilities of the hardware free to be used for other effects.

We decided to subdivide the faces if necessary, so we only require extra processing for the polygons that have two or three colours. This subdivision process will be explained in detail in the next section.

3.5. Subdividing Faces Using Interpolation

We suppose that our input only consists of triangles. If not, the faces can be triangulated using standard software or by implementing an algorithm such as in [11].

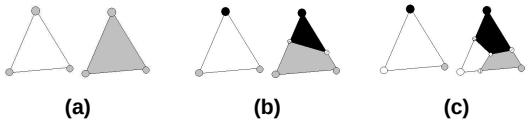


Figure 1. Three possible subdivisions.

For every vertex of the triangle, we determine which cartoon-colour most closely matches it. By comparing the colours for the vertices, three cases can arise (see figure 1).
a) The three colours are all equal. No splitting is needed: the face gets the same colour as the points.
b) Two colours are equal and the other one is different. On two edges in-between points are calculated. The face will be divided in two via the line connecting these in-between points. Each piece will get the colour that matches the points it is closest to.
c) The three colours are all different. In that case in-between points will be calculated on each edge. A fourth point m will also be determined. This point m is defined as the centroid of the three in-between points. By connecting m to each in-between point, the original face will be cut into three pieces. The colour of each piece is again determined by the colour of the nearest original vertex.

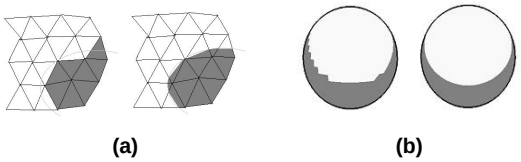


Figure 2. Without versus with subdivision.

Figure 2(a) illustrates the need for subdividing. Here the grey-coloured arc marks the form that ideally would divide the light and the dark area on the polygon mesh. In the drawing at the left, no polygons are subdivided, while on the right the subdivisions are calculated using the interpolation of intensity values in each vertex. As can be noticed, the dark area in the left picture is very rough, while the corresponding area on the right is much smoother. Both dark areas have straight edges and use the same input data, but look totally different.

Figure 2(b) is a concrete illustration of our technique. The sphere at the left is rendered without the subdivision, while the sphere at the right uses the subdivision scheme that is described in the following sections. The image with the subdivisions looks much smoother and also behaves much more fluently in animations.

3.6. Calculating the In-between Points

The in-between points are calculated by linear interpolation of the colours.

Let a and b be the two points for which an in-between point r is needed. Let ia and ib be the respective intensities of the colour in these points, and Ia and Ib the respective cartoon-colours. The intensity in r, ir, has to be equal to the threshold between the two cartoon-colours, Ia and Ib. This threshold can either be chosen freely by the animator, or it can be some simple derivation of Ia and Ib. An example is to choose ir as the mean between Ia and Ib, thus:

$$ir = (Ia + Ib)/2 \tag{1}$$

The fact that r lies on the line between a and b allows us to express r as a linear combination:

$$r = t * a + (1 - t) * b \tag{2}$$

In the Gouraud model of interpolating colour inside a face, the same interpolator t also governs the relation between the intensities:

$$ir = t * ia + (1 - t) * ib \tag{3}$$

Rewriting equation 3, we get a formula for t:

$$t = (ir - ib)/(ia - ib) \tag{4}$$

If we now use the value of ir obtained in equation 1, we get a value for t. Substituting that value of t, equation 2 will deliver us the desired in-between point r:

$$r = b + (ir - ib)/(ia - ib) * (a - b) \tag{5}$$

3.7. Overview of the Rendering Process

In this section, we show a simplified overview of the cartoon rendering process:
Pre-processing step:

- Read the objects into memory, building up a winged-edge data structure.
- Create a display list for each surface in the objects.
- Calculate and mark sharp and border edges.

For each individual rendering:

- Get the view transformation from the underlying animation program.
- Render the display list of each surface into the feedback buffer[1] using Gouraud shading.
- The feedback buffer is processed, so the actual rendering can start, using the calculated vertex colours to assign cartoon-colours and optionally subdivide the polygons. In this step the cartoon polygons are rendered into the screen buffer.
- The sharp edges are drawn.
- Calculate silhouette lines: the list with non-sharp edges is processed, drawing all edges that connect a front-face with a back-face (depending on the view direction).

The pre-processing step, which is executed only once per session, takes several seconds to complete. But each rendering step can be performed in real-time, at least until a certain complexity of the scene (about 10,000 polygons with our current hardware).

The use of the feedback buffer turns out to be especially beneficial. An initial remark is that not all the rendering is done twice. In the first rendering (the one to the feedback buffer), all objects are transformed from object co-ordinates to screen space. In the second rendering (the actual rendering of the adapted mesh), all co-ordinates are already in screen space without the need for any transformation. Furthermore, in order to make optimal use of the fact that the processor on the main machine and the processor on the graphical board can perform tasks in parallel, the rendering to the feedback buffer is started in a separate thread, each time with a different part of the object, while the main processor does the subdivision and colouring of the previous polygons.

4. A Faster Silhouette Detection Algorithm

Analogous to [5] and [6], we convert the silhouette edge detection to a dual problem. Each face in the primal space is represented as a 4D point in the dual space via its plane equation. After normalisation, these points form a sphere, known as the Gauss map of the faces in the primal space. This allows us to convert problems in the primal space to easier to attack problems in a dual space.

The older approaches represent the edge between two adjacent faces as the arc joining the dual points that represent those faces. We – on the contrary – simple look at the dual edge as a pair of points, one for each of the faces.

The camera used in the rendering can either be a 3D point or a 3D direction vector. The camera is represented in the 4D dual space by converting it to homogenous co-ordinates, i.e. having the 4th co-ordinate equal to 1 for a point in the scene, and having it equal to 0 for a direction vector.

To check whether a face with the plane equation $Ax + By + Cz + D = 0$ is a front face, we have to substitute the camera (Vx, Vy, Vz) in its plane equation and look for a

[1] When in feedback mode, no pixels are produced by rasterisation. Instead, information (such as screen co-ordinates and colour values) about primitives that would have been rasterised is fed back to the application.

positive value. In the dual 4D space, this is just a dot product: $(A, B, C, D).(Vx, Vy, Vz, Vw)$ to be evaluated. To find silhouette edges, we have to find edges in the dual space for which the corresponding faces give different signed results for the dot product.

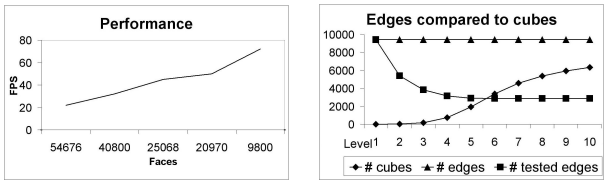
In order to rapidly search for silhouette edges, all the edges of the input model are stored in a 4D variant of an octtree. At the highest level, one 4D cube – ranging from $(-1, -1, -1, -1)$ to $(1, 1, 1, 1)$ – encloses the Gaussian sphere. By splitting the space for each co-ordinate in two, this 4D cube can be divided into 16 smaller 4D cubes. This process of dividing into sub-cubes can be repeated recursively until some small enough limit level is reached. In a pre-processing step, each edge is recursively put into an as small as possible sub-cube that encloses both points representing the edge. Only sub-cubes that contain at least one edge are represented.

In a final pre-processing step, we assign two bounding boxes to each sub-cube. The first bounding box encloses all edges belonging to the sub-cube's children while the second one encloses the sub-cube's own edges.

So far for the pre-processing. Now at runtime, for the silhouette edge detection from a given camera position, the 4D octtree is traversed recursively, only looking at those sub-cubes for which at least one of the two bounding boxes has both a positive and a negative value for the dot product. During the traversal of the 4D octtree, all edges stored on each visited level are tested for being a silhouette edge.

If we suppose – as in the papers referenced in the related work section – that the number of silhouette edges for a "usual" object is relatively small compared to the number of total edges, our algorithm will have to test only a small fraction of the input edges. The main difference with the referenced papers is a simpler setup adapted to perspective views, while using bounding boxes that are much tighter than a hierarchy of cones.

Figure 3 shows the performance of our algorithm and the number of edges tested in relation to the number of cubes created.



(a) (b)

Figure 3. a) Performance of our algorithm. b) Tested edges compared to created cubes (based on the fish of figure 4(a)).

The proposed algorithm can be adapted to find the polygons that contain the border between cartoon-colours.

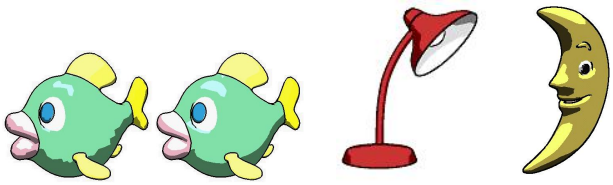
Therefore, we have a closer look at the light equation, for simplicity limiting ourselves to an ambient term A and one diffuse light. The light intensity I is then:

$$I = A + max(0, N.L) \tag{6}$$

Here N is the surface normal and L is the normalised vector from the surface towards the light. Note that the clamping of the diffuse light to be at least 0 has no effect for $N.L$ sufficiently large, as is the case for the area where the borders between the cartoon-colours are defined. This way the light formula can be rewritten to only depend on the normal plane on the surface and be compared as larger or smaller than 0, depending on a value derived from the chosen border between the colours.

So, instead of the equations of the planes of the faces, in the 4D octtree, this time we store the normal plane of the points of the model. We can then quickly find those points, for which our winged-edge data structure supplies the polygons that are to be divided, as they have to be coloured by more than one cartoon-colour.

5. Examples



(a) (b) (c)

Figure 4. a) A fish (without/with subdivision). b) A desk lamp. c) A moon shaped toy.

The fish of figure 4(a) contains 6,280 faces and 3,164 vertices. Both the version using subdivision and the version without subdivision get rendered at a speed of 30 frames per second. The desk lamp of figure 4(b) and the moon (6,320 faces) of figure 4(c) are both rendered at 26 frames/sec. Note that there are two light sources used to illuminate the moon-shaped toy.

The deer of figure 5 consists of 10,396 faces and gets rendered at a speed of 18 frames per second. Figure 5(c) shows what happens when texture smoothing is turned on: the smoothing region is too large, making the border between the bi-tonal regions very fuzzy. Figures 5(d, e) give examples of how texture-mapping can be used to obtain different rendering styles.

For figure 6(b) we turned on the "Multi-sampling for High-Resolution Anti-aliasing (HRAA)", an anti-aliasing

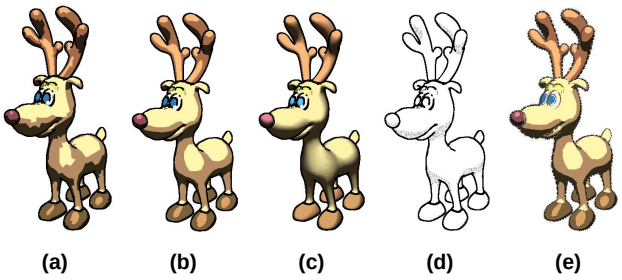


Figure 5. a, b) A deer rendered without/with subdividing. c) Smoothing the texture as suggested by [8]. d) Replacing a cartoon colour with a texture and (e) texturing the silhouette.

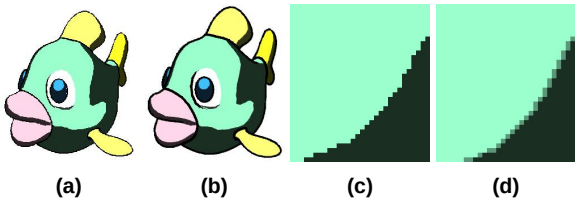


Figure 6. A standard rendering without anti-aliasing (a) versus anti-aliasing turned on (b). (c, d) show zoom-ins on (a, b).

method implemented in hardware on the GeForce3 graphics board. The anti-aliasing is supported without any loss of speed. Figures 6(c, d) show zoom-ins on a detail of figures 6(a, b). If we reverted to the texture-mapping approach suggested by [8], the smoothing would only be able to smooth the silhouette edges but would keep rendering the border between the cartoon-colours with a staircase effect.

6. Conclusions and Future Research

In this paper we describe our approach to successfully implementing a cartoon-rendering. Furthermore, we explain the techniques that are used to boost this rendering to real-time performance using currently available off-the-shelf hardware. We also elucidate the techniques to use to get the quality of the images as high as possible, given the real-time constraint.

In addition, we present a compact new algorithm to rapidly search for silhouette edges and for the polygons containing the borders between the cartoon-colours.

In our future research, we would like to investigate other applications of our silhouette edge detection algorithm, such as in the field of computer vision [10]. The quick location of silhouettes and shading borders could be especially helpful in detecting the orientation of known objects.

Acknowledgements

We would like to express our thanks to Bart Claes and Marc Flerackers for their help in the implementation.

We also wish to express much gratitude to PAVR (a european TMR project) and the Flemish Government, which are kindly funding part of the research reported upon in this paper.

References

[1] J. Buchanan and M. Sousa. The edge buffer: A data structure for easy silhouette rendering. *NPAR 2000: Symposium on Non-Photorealistic Animation and Rendering*, pages 39–42, June 2000.

[2] P. Decaudin. *Modélisation par Fusion de Formes 3D pour la Synthèse d'Images – Rendu de Scènes 3D imitant le Style "Dessin Animé"*. PhD thesis, Université de Technologie de Compiegne (France), 1996.

[3] J. Foley and A. Van Dam. *Computer Graphics: Principles and Practice*. Addison Wesley Pub Co, ISBN: 0201848406, July 1995.

[4] A. Glassner. Maintaining winged-edge models. *In James Arvo, editor, Graphics Gems II*, pages 191–201, 1991.

[5] B. Gooch, P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. *ACM Symposium on Interact. 3D Graphics '99*, pages 31–38, 1999.

[6] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, pages 517–526, 2000.

[7] D. Johnson and E. Cohen. Spatialized normal cone hierarchies. *Proceedings 2001 ACM Symposium on Interactive 3D Graphics*, pages 129–134, March 2001.

[8] A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3d animation. *NPAR 2000: Symposium on Non-Photorealistic Animation and Rendering*, pages 13–20, June 2000.

[9] L. Markosian, M. Kowalski, S. Trychin, L. Bourdev, D. Goldstein, and J. Hughes. Real-time non-photorealistic rendering. *Proceedings of SIGGRAPH '97*, pages 415–420, August 1997.

[10] W. Matusik, C. Buehler, R. Raskar, S. Gortler, and L. McMillan. Image based visual hulls. *Proceedings of SIGGRAPH 2000*, pages 369–374, 2000.

[11] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, ISBN: 0521445922, 1994.

[12] R. Raskar and M. Cohen. Image precision silhouette edges. *Proceedings of ACM Symposium on Interactive 3D Graphics*, pages 135–140, 1999.

[13] T. Saito and T. Takahashi. Comprehensible rendering of 3D shapes. *Proceedings of SIGGRAPH '90*, pages 197–206, 1990.

[14] P. Sander, X. Gu, S. Gortler, H. Hoppe, and J. Snyder. Silhouette clipping. *Proceedings of SIGGRAPH 2000*, pages 335–342, 2000.