Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 29 (2005) 393–402
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Fast extraction of polyhedral model silhouettes from moving viewpoint on curved trajectory

Ku-Jin Kim[a,*], Nakhoon Baek[b]

[a]Department of Computer Engineering, College of Engineering, Kyungpook National University, Daegu 702-701, Republic of Korea
[b]Department of Computer Science, School of EECS, Kyungpook National University, Daegu 702-701, Republic of Korea

Abstract

The efficient extraction of model silhouettes is essential in many applications, such as non-photorealistic rendering, backface culling, shadow computation, and computing swept volumes. For dynamically moving viewpoints, efficient silhouette extraction is more important for system performance. Accordingly, this paper presents an incremental update algorithm for computing a perspective silhouette sequence for a polyhedral model. The viewpoint is assumed to move along a given trajectory $\mathbf{q}(t)$, where t is the time parameter. As the preprocessing step, the time-intervals during which each model edge is contained in the silhouette, defined as silhouette time-intervals, are computed using two major computations: (i) intersecting $\mathbf{q}(t)$ with two planes and (ii) a number of dot products. If $\mathbf{q}(t)$ is a curve of degree n, there are at most $n+1$ silhouette time-intervals for an individual edge. The silhouette time-intervals are then used to determine the edges that should be added or deleted from the previous silhouette for each discrete viewpoint, thereby providing an optimal way to compute a sequence of silhouettes. A search-based algorithm is also presented that extracts the silhouette edges for each time point t_j by searching the silhouette time-intervals containing t_j. The performance of the proposed algorithms is analyzed and experimental results are compared with those for the anchored cone algorithm suggested by Sander et al. [In: Akeley K, editor. Siggraph 2000, Computer Graphics Proceedings. Annual Conference Series. New York/Reading, MA/New York: ACM Press/ACM SIGGRAPH/Addison-Wesley/Longman; 2000. p. 327–34.]

Keywords: Object space silhouette; Computer animation; Non-photorealistic rendering; Swept volume generation

1. Introduction

In computer graphics, researchers have been giving intensive attention to the efficient silhouette computation, as silhouettes are essential for a wide range of applications, such as non-photorealistic rendering, shadow computation, and backface culling. Since the silhouette of a model is the most important visual cue for recognizing its shape, even in the case of model simplification, the silhouette of the original model is used as the simplified model to look more realistic [1].

When silhouette extraction is used in computer animation, the camera position is considered as the viewpoint and camera path as the viewpoint trajectory. The camera path is usually given as a curve, so the corresponding viewpoint also moves along a curved trajectory, which makes the efficiency of the silhouette detection algorithm more critical than in the case of a fixed viewpoint. Fast silhouette extraction is also

*Corresponding author. Tel.: +82 53 950 7573; fax: +82 53 957 4846.
E-mail addresses: kujinkim@yahoo.com (K.-J. Kim), bnhoon@yahoo.co.kr (N. Baek).

0097-8493/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2005.03.009

important in geometric modeling systems, where swept volumes are often used to create objects. The silhouettes of a model from a moving viewpoint can be used as a starting point for the swept volume computation [2,3]. If sequential silhouettes of a model are extracted in real-time, the swept volume can then be computed in pseudo real-time.

Isenberg et al. [4] classified silhouette detection algorithms into object space algorithms [1,5–8], image space algorithms [9,10], and hybrid algorithms [11–13]. Among them, object space algorithms have the advantages of computing silhouettes in an analytic form, obtaining exact silhouettes, and convenient stylization or further processing of the silhouettes. While there are many efficient algorithms that compute silhouettes from a fixed viewpoint, there are relatively few algorithms applicable to a moving viewpoint. Accordingly, this paper presents a silhouette detection algorithm for a polyhedral model from a moving viewpoint in object space.

Depending on the viewpoint, there are two types of silhouette: a perspective silhouette and parallel silhouette. For the case of polyhedral models, both types of silhouette can be defined as a set of model edges with a special property. Let E be an edge of the polyhedral model, and F_1 and F_2 be two facets sharing E. As such, edge E is in a perspective silhouette from viewpoint $\mathbf{q}$ if and only if the facets F_1 and F_2 are at the same side of the plane containing viewpoint $\mathbf{q}$ and edge E. To compute a parallel silhouette, a view direction is given, which is an idealized viewpoint at an infinite distance from the model. As such, edge E is contained in the parallel silhouette with respect to the view direction $\mathbf{q}$ if and only if the facets F_1 and F_2 are at the same side of the plane containing edge E and parallel to vector $\mathbf{q}$. Usually, the computation of a parallel silhouette is easier than that of a perspective silhouette. Consequently, this paper considers the problems involved with computing perspective silhouettes, although the proposed algorithms can also be extended with minor modifications to compute parallel silhouettes. Thus, in the remainder of this paper, silhouette implies a perspective silhouette.

Let $\mathbf{q}(t)$ be the viewpoint trajectory, where t is the parameter denoting time. To compute sequential silhouettes from viewpoints on $\mathbf{q}(t)$, one intuitive way is to repeatedly apply a silhouette extraction algorithm from a fixed viewpoint, while changing the position of the viewpoint. However, this technique ignores the temporal coherence in sequential silhouettes. Few differences can be expected between silhouettes from two close viewpoints $\mathbf{q}(t_i)$ and $\mathbf{q}(t_i+\varepsilon)$. If one of the model edges is contained in the silhouette from viewpoint $\mathbf{q}(t_i)$, there is a high possibility that the same edge will also be in the silhouette from viewpoint $\mathbf{q}(t_i+\varepsilon)$. Thus, when assuming that the viewpoint moves with respect to time, a particular model edge should be contained in a silhouette for a time-interval rather than a set of discrete time points.

Two efficient algorithms are presented to extract a sequence of silhouettes for a polyhedral model from a moving viewpoint: (i) an incremental update algorithm and (ii) search-based algorithm. Both algorithms require the preprocessing of computing time-intervals for an edge to be included in the silhouette. Let the time-intervals be denoted as *silhouette time-intervals*. It is also assumed that the viewpoint moves along a trajectory $\mathbf{q}(t)$. Then, the end points of the silhouette time-intervals can be computed for each edge E by intersecting $\mathbf{q}(t)$ with the supporting planes of the two facets that share E. When the degree of $\mathbf{q}(t)$ is n, the end points of the time-intervals can be computed by solving two equations of degree n. For a model edge, the number of silhouette time-intervals is at most $n+1$.

The incremental update algorithm is considered first. When extracting f frames of silhouettes from a sequence of viewpoints $\mathbf{q}(t_j)$, $0 \leqslant j < f$, a data structure needs to be constructed that manages the information on the beginning and ending of edge inclusion in a silhouette from each viewpoint $\mathbf{q}(t_j)$. By using this data structure, a silhouette can then be extracted at each time point t_j by adding and deleting the appropriate edges from the previous silhouette. When computing the silhouette from viewpoint $\mathbf{q}(t_j)$, the edges in the previous silhouette $\mathbf{q}(t_{j-1})$ are visited, then the appropriate edges are added or deleted just once, making this an optimal algorithm.

In contrast to the above incremental update algorithm, the search-based algorithm directly uses the set of silhouette time-intervals with the corresponding edge information, such as (b, e, E), where b and e are the beginning and ending time points, respectively, for edge E to be included in the silhouette. Then, the silhouette from viewpoint $\mathbf{q}(t_j)$ can be extracted by searching the intervals containing t_j. To implement the search-based algorithm efficiently, an interval tree is used as the main data structure.

When the length of the viewpoint trajectory exceeds a threshold, the incremental update algorithm produces a better performance than the search-based algorithm. Thus, the current experiments are focused on showing the performance of the incremental update algorithm compared to previous work. The contributions of the proposed incremental update algorithm are summarized as follows:

- It is a novel algorithm that defines and uses silhouette time-intervals for computing the sequence of object space silhouettes. Based on preprocessed silhouette time-intervals, the incremental update algorithm can extract a silhouette sequence in an optimal way. Experiments show that the performance of the incremental update algorithm is better than that in the previous work.

- Various efficient silhouette extraction algorithms already exist, such as [1,8], yet they require heuristics that make the implementation difficult and take a relatively long time. Thus, when compared to such algorithms, the incremental update algorithm is analytic, plus it is simple and easy to implement.
- The algorithms in [6,7] do not require heuristics and are easy to implement. However, these algorithms do not always guarantee the extraction of exact silhouettes from a moving viewpoint, as they require the selection of appropriate starting points to trace the silhouette for each frame. Thus, when compared to these algorithms, the incremental update algorithm always provides accurate whole silhouettes.

The limitations of the proposed incremental update algorithm are mainly related to its preprocessing requirement:

- The algorithm requires the computation of silhouette time-intervals whenever a new viewpoint trajectory is given.
- The algorithm needs memory space in proportion to the number of silhouette time-intervals.

The remainder of this paper is organized as follows. Section 2 presents related work, then the computation of silhouette time-intervals is explained in Section 3. Section 4 outlines the proposed search-based algorithm with two data structures: an array and interval tree, while Section 5 introduces the proposed incremental update algorithm. The performance of both algorithms is analyzed in Section 6, plus the proposed incremental update algorithm is experimentally compared with the anchored cone algorithm proposed by Sander et al. [1] in Section 7. Some final conclusions and areas for future work are given in Section 8.

2. Related work

The silhouette extraction problem has already been investigated using many different approaches. Thus, according to the classification of Isenberg et al. [4], this paper discusses the trade-offs between existing silhouette detection algorithms based on the following categories: object space, image space, and hybrid algorithms.

Image space algorithms [9,10,13] extract silhouettes from a geometric buffer, such as a depth buffer or normal buffer, by detecting the discontinuities in the image. A depth buffer uses pixel intensities to represent the depth information on an object from a particular viewpoint. Thus, by detecting the edges in a depth buffer image, C^0 discontinuities in a scene can be detected. Similarly, a normal buffer contains the normal information on an object. Hertzmann [10] suggested an approach to obtain C^0,C^1 discontinuities and silhouettes in a scene by combining the depth map and normal map. Meanwhile, hybrid algorithms modify the faces of the model in the object space, and render them using a z-buffer. The extracted silhouette is then shown in the image space. For example, Raskar and Cohen [12] suggested a method for rendering image precision silhouettes by drawing a frontface polygon over an enlarged backface polygon, while Gooch et al. [11] computed silhouettes in a similar way, and applied it to technical illustration in manufacturing.

Image space algorithms and hybrid algorithms use the benefits provided by existing graphic hardware, so they have advantages as regards generating silhouettes in real-time or at an interactive frame rate. The visibility of the silhouette is solved when the silhouette is rendered. However, the resulting silhouettes do not contain geometric properties, so it is difficult to stylize or apply further processing to them. Such silhouettes also have a low precision and suffer from an aliasing problem.

In contrast, object space algorithms [1,6–8,10] have advantages as regards their analytic description, the production of exact silhouettes, and convenient stylization or further processing of the silhouettes. However, object space algorithms cannot solve the visibility-culling problem while the silhouettes are computed.

Markosian et al. [7] used a randomized algorithm for the silhouette detection problem, where the initial silhouette edge is identified by examining only a small subset of the edges in the model. The connected silhouette edges are then efficiently extracted from the initial edge, yet there is no guarantee that every connected component of the silhouette can be extracted. Meanwhile, Benichou and Elber [5] computed parallel silhouettes of polyhedral models using a Gaussian sphere, where the normal vectors for the edges of the model and the view direction are mapped onto a Gaussian sphere, at which point they become various great arcs and one great circle, respectively. This data is then projected from the Gaussian sphere onto a circumscribing cube, where the arcs and circle became straightlines, and the silhouette curve is finally computed by intersecting the lines.

Hertzmann [10] considered a polyhedral mesh as an approximation of a surface. After computing the normal vector $\mathbf{N}_i$ at each surface point $\mathbf{p}_i$, a normalized dot product of $\mathbf{N}_i$ and the viewing direction $(\mathbf{q}-\mathbf{p}_i)$, where $\mathbf{q}$ is the given viewpoint, is computed. For edges where the dot product results for the end points have different signs, the silhouette point on the edge with a potential zero dot product value is selected. The whole silhouette curve is then derived by connecting the silhouette points. Using the above silhouette finding technique, Hertzmann and Zorin [6] presented algorithms for the line-art rendering of smooth surfaces.

Gu et al. [14] rendered a polyhedral mesh with a coarse mesh and exact silhouette. In this case, the use of a coarse mesh produces an efficient rendering, while the use of an exact silhouette makes the coarse mesh look realistic. The viewpoint is assumed to move on a circumscribing sphere of the object. For each sampled viewpoint on the sphere, the silhouettes are computed in a preprocessing step. Then, for an arbitrary viewpoint on the sphere, the silhouette is computed by interpolating the preprocessed silhouettes from the neighboring viewpoints.

Sander et al. [1] hierarchically constructed search trees to speed up the silhouette detection. Each node in the tree contains two anchored cones for one face cluster. The two anchored cones are constructed to contain viewpoint positions that make the corresponding face cluster either front facing or back facing, respectively. As a result, the efficiency of the silhouette computation is improved by removing a large portion of the edges that are definitely not in the silhouette.

Although most previous research has focused on computing silhouettes from a fixed viewpoint, there are also several approaches that compute a sequence of silhouettes based on a temporal coherence. For example, Pop et al. [8] proposed several on-line silhouette finding applications, where, for a sequence of different viewpoints, rather than computing the complete silhouette each time, the changes in the silhouette of a polyhedral model are computed between consecutive frames. The normal vectors for the edges in the model are transformed into dual line segments, while the viewpoint is transformed to a dual plane. The silhouette edges then correspond to the dual line segments that intersect with the dual plane. Yet, the problem with this approach is that it is difficult to find such dual line segments efficiently. Thus, the problem was reduced to finding the end points of the line segments included in a double-wedge of two dual planes, corresponding to two close viewpoints, using heuristics. This algorithm is actually complementary to the algorithm proposed in a later section.

3. Computation of silhouette time-intervals

This section presents the derivation of equations to compute the silhouette time-intervals for an edge E in a model. It is assumed that E is shared by two facets F_1 and F_2, and the end vertices of E are $\mathbf{p}_1$ and $\mathbf{p}_2$. The outward normal vectors of F_1 and F_2 are denoted as $\mathbf{N}_1$ and $\mathbf{N}_2$, respectively. It is also assumed that $\mathbf{q}(t)$, $t_{min} \leqslant t \leqslant t_{max}$, is the trajectory of a moving viewpoint. Edge E is part of the silhouette from viewpoint $\mathbf{q}(t)$, if and only if $((\mathbf{q}(t)-\mathbf{p}_1)\cdot\mathbf{N}_1)((\mathbf{q}(t)-\mathbf{p}_1)\cdot\mathbf{N}_2) \leqslant 0$.

For an arbitrary plane P, the half-space with respect to P, which contains the normal vector of P, is denoted as P^+, while the other half-space is denoted as P^-. Then, P^+ and P^- can be represented as $P^+ = \{\mathbf{q} | (\mathbf{q}-\mathbf{p})\cdot\mathbf{N} \geqslant 0\}$ and $P^- = \{\mathbf{q} | (\mathbf{q}-\mathbf{p})\cdot\mathbf{N} \leqslant 0\}$, respectively, where $\mathbf{N}$ is the normal vector of P and $\mathbf{p}$ is an arbitrary point on P. Based on this property, the regions in the three-dimensional (3-D) space can be classified according to whether they contain viewpoints that include E in a silhouette. In Fig. 1(a), there are two faces F_1 and F_2 that share edge E. When the supporting planes of F_1 and F_2 are denoted as P_1 and P_2, respectively, the 3-D space can be classified into four regions: (i) region $P_1^+ \cap P_2^+$, (ii) region $P_1^+ \cap P_2^-$, (iii) region $P_1^- \cap P_2^+$, and (iv) region $P_1^- \cap P_2^-$. The necessary and sufficient condition for E to be included in a perspective silhouette from viewpoint $\mathbf{q}$ is that $\mathbf{q}$ is either in $P_1^+ \cap P_2^-$ or $P_1^- \cap P_2^+$. Fig. 1(b) is a view of the faces in Fig. 1(a) from an infinite distance along the line containing E. The shaded area corresponds to two regions: region $P_1^+ \cap P_2^-$ and region $P_1^- \cap P_2^+$. In Fig. 1(b), the viewpoint trajectory $\mathbf{q}(t)$ intersects with two planes P_1 and P_2 at $\mathbf{q}(t_d)$, $d = 0, 1, 2, 3$. In this case, the necessary and sufficient condition for edge E to be included in the perspective silhouette is that the viewpoint is on the curve segments $\mathbf{q}(t)$ for $t_0 \leqslant t \leqslant t_1$ or $t_2 \leqslant t \leqslant t_3$, where those segments are included in region $P_1^+ \cap P_2^-$. The silhouette time intervals for E are the intervals (t_0, t_1) and (t_2, t_3).

The end points of the silhouette time-intervals for E are computed by intersecting $\mathbf{q}(t)$ with P_1 and P_2, and

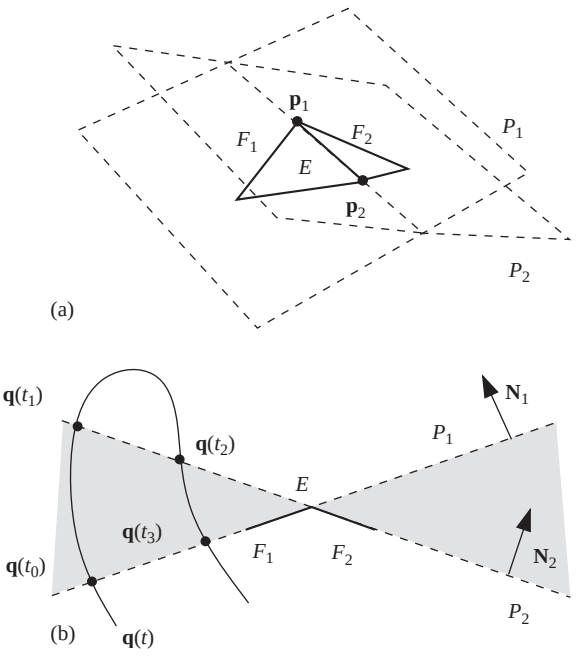

Fig. 1. Perspective silhouette edge E.

are calculated as the values of t as follows:

$$\{t|(\mathbf{q}(t)-\mathbf{p}_1)\cdot\mathbf{N}_1=0\cup(\mathbf{q}(t)-\mathbf{p}_1)\cdot\mathbf{N}_2=0 \text{ for } t_{min}\leqslant t\leqslant t_{max} \text{ and } t\in\mathbb{R}\}. \quad (1)$$

If the values of t in Eq. (1) are sorted in an ascending order and the sequence of $t_j, j=0,1,2,\ldots,n$, is derived, then the necessary and sufficient condition for edge E to be contained in the silhouette from viewpoint $\mathbf{q}(t)$, $t_j\leqslant t\leqslant t_{j+1}$, is as follows:

$$((\mathbf{q}(t_m)-\mathbf{p}_1)\cdot\mathbf{N}_1)((\mathbf{q}(t_m)-\mathbf{p}_2)\cdot\mathbf{N}_2)\leqslant 0,$$

where t_m is an arbitrary value between t_j and t_{j+1}.

When the degree of $\mathbf{q}(t)$ is n, the maximum number of silhouette time-intervals between t_{min} and t_{max} for each edge is $n+1$, because the number of intersections between the two planes and $\mathbf{q}(t)$ is at most $2n$. For efficient computation of the intersection, it is assumed that $\mathbf{q}(t)$ is given as a sequence of quadratic curves.

4. Search-based algorithm

This section presents a search-based algorithm for extracting a sequence of silhouettes from a moving viewpoint. Assume the extraction of silhouettes for f frames from a moving viewpoint on $\mathbf{q}(t)$, $t_{min}\leqslant t\leqslant t_{max}$, is considered. The silhouettes are then supposed to be computed from viewpoints $\mathbf{q}(t_j)$, where $t_j\in[t_{min},t_{max}]$ and $j=0,1,2,\ldots,f$. For the polyhedral model $\mathscr{P}$, the set of silhouette edges for $\mathscr{P}$ from viewpoint $\mathbf{q}(t_j)$ is denoted as $\mathscr{S}(j)$, where j is the frame identification number. The algorithm for computing $\mathscr{S}(j)$ can be summarized as follows:

Algorithm: Search_Based_Silhouette_Extraction.

Input: Polyhedral model $\mathscr{P}$ and viewpoint trajectory $\mathbf{q}(t)$, $t_{min}\leqslant t\leqslant t_{max}$.
Output: Sequence of silhouettes $\mathscr{S}(j)$, $j=0,1,2,\ldots,f$.
Step 1: For each edge E of $\mathscr{P}$, compute the silhouette time-intervals for E, then add the interval and edge information to set I.
Step 2: Construct an array (or interval tree) for the silhouette time-intervals in I.
Step 3: For each time point t_j,
(a) Search the array (or interval tree) to find intervals containing t_j, then add the corresponding edges to set $\mathscr{S}(j)$.
(b) Draw or proceed on $\mathscr{S}(j)$.

Assume that set I contains k intervals: $I=\{l_i|1\leqslant i\leqslant k\}$ and each silhouette time-interval l_i consists of three fields: $l_i=(b_i,e_i,E_i)$, where b_i and e_i are the time-interval endpoints and $b_i\leqslant e_i$. E_i is the edge that corresponds to the time-interval. Since the efficiency of a search-based algorithm depends on the search time for the time-intervals, two different data structures are introduced for storing the time-intervals: an array and interval tree. Theoretically, an interval tree is known as the optimal data structure for finding intervals containing a specific value. However, experimental results show that an array has a better performance than an interval tree when considering the preprocessing time.

4.1. Interval array

The intervals in set I are stored in an array structure with three fields for b_i, e_i, and E_i. Given the value t_j, a sequential search of l_i such that $b_i\leqslant t_j\leqslant e_i$ is applied to I.

4.2. Interval tree

An interval tree [15,16] is a useful data structure when searching for intervals containing a specific value. Thus, for the intervals in set I, a static interval tree is constructed as follows: Let (b_i,e_i,E_i) denote the silhouette time-interval (b_i,e_i) for edge E_i, and $(t_1,t_2,t_3,\ldots,t_{2k})$ be the sorted sequence for the endpoints of the k time-intervals $\{(b_i,e_i)|1\leqslant i\leqslant k\}$. Then, the interval tree T for set I is constructed as follows:

(1) The root w of T has a discriminant $\delta(w)=(t_k+t_{k+1})/2$ and points to two (secondary) lists $L(w)$ and $R(w)$, $L(w)$ and $R(w)$ contain the sorted lists of the left and right end points, respectively, of the members of I containing $\delta(w)$, and $L(w)$ and $R(w)$ are sorted in ascending and descending order, respectively.
(2) The left subtree of w is the interval tree for sequence $(t_1,t_2,\ldots,t_k)$ and the subset $I_L\subseteq I$ of the intervals whose right extreme is less than $\delta(w)$. The right subtree of w is defined analogously.

To construct secondary lists for nodes w, $L(w)$ and $R(w)$, an AVL tree was used for an efficient search. Only a static interval tree is needed, i.e. there is no need to insert or delete the nodes from the interval tree dynamically. A static interval tree for a collection of k intervals uses $O(k)$ space, since there are $4k-1$ nodes in the primary structure and at most $2k$ values to be stored in the secondary lists. The skeletal primary structure is constructed in $O(k\log k)$ time, since the endpoints of k time-intervals have to be sorted. When searching for an interval containing a specific value, the time complexity involved in searching the primary structure is $O(\log k)$, plus additional time is needed to visit the secondary lists.

5. Incremental update algorithm

The basic idea of the proposed incremental update algorithm is to determine when an edge begins or ceases to be a silhouette edge in the preprocessing step. The preprocessing of silhouette time-interval information can facilitate the extraction of sequential silhouettes, and data structures are used to keep the list of edges that need to be added or deleted from the silhouette for each time point t_j, $0 \leqslant j \leqslant f$: *Add*[$j$] and *Delete*[$j$].

```
struct EdgeList {
  int Eid;
  struct EdgeList *link;
} *Add [NUM_FRAME], *Delete [NUM_FRAME];
```

Fig. 2 presents an example of constructing *Add* and *Delete* for four edges E_i, $0 \leqslant i \leqslant 3$. The horizontal axis represents the values of the time parameter t, while the horizontal thick line segments are the silhouette time-intervals for each edge. For example, edge E_0 has two silhouette time-intervals l_0 and l_1. During the continuous time intervals l_0 and l_1, E_0 is included in the silhouette. Thus, since E_0 is included in the silhouette for four sequential time points t_0, t_1, t_2, and t_3, E_0 is attached to *Add*[0] and *Delete*[3]. Meanwhile, in the case of E_1, l_3 does not contain any time points, thus, only two intervals l_2 and l_4 contribute to constructing *Add* and *Delete*. Therefore, E_1 is attached to *Add*[1], *Delete*[2], *Add*[4], and *Delete*[4].

After constructing *Add* and *Delete*, the silhouettes can be incrementally updated for each time point using the following data structure of edges:

```
struct Edge {
  int vertex[2]; /* the adjacent vertices of the edge */
  bool isSilhouette; /* notifying whether this edge is
included in the silhouette */
  int link; /* containing the index of the next silhouette
edge */
} E[NUM_EDGES];
```

For each time point t_j, the edge list in *Add*[j] is added to $\mathscr{S}(j)$ by updating the *isSilhouette* and *link* fields for each edge. For each edge $E[i]$ in *Add*[j], $E[i]$.*isSilhouette* is updated as TRUE, while $E[i]$.*link* is updated to point to the next silhouette edge. Meanwhile, the edges in *Delete*[$j-1$], i.e. the edges with a *isSilhouette* field marked *FALSE* after computing $\mathscr{S}(j-1)$, are deleted while $\mathscr{S}(j)$ is extracted. For each time point t_j, the silhouette is extracted by visiting the edges in *Add*[j], *Delete*[j], and the previous silhouette $\mathscr{S}(j-1)$ only once. The details of the incremental update algorithm are presented in Appendix A.

6. Performance analysis

The time and space complexity of the brute force algorithm and proposed algorithms are presented in Table 1, where N is the number of edges in $\mathscr{P}$, k is the number of silhouette time-intervals, M is the average number of edges in the silhouette, and f is the number of frames.

The search-based algorithm using an interval tree has a better time complexity than that using an interval array, yet the preprocessing for the interval tree took considerably longer than that for the interval array. When the length of the viewpoint trajectory exceeded a threshold, the incremental update algorithm produced a better performance than the search-based algorithms.

7. Experimental results

This section compares three algorithms: the proposed incremental update algorithm, an anchored cone algorithm [1], and brute force algorithm. The incremental update algorithm was implemented using C programming and an OpenGL library, then the three algorithms were tested based on computing a sequence of perspective silhouettes for four polyhedral models: *bunny*, *hand*, *gargoyle*, and *parasaur* (see Fig. 4) in a PC with a Pentium 4 (2.80 GHz) processor and 1 GB RAM.

The models were normalized within a bounding box with a maximum edge length of 200, and the viewpoint trajectory $\mathbf{q}(t)$, which was composed so that the model

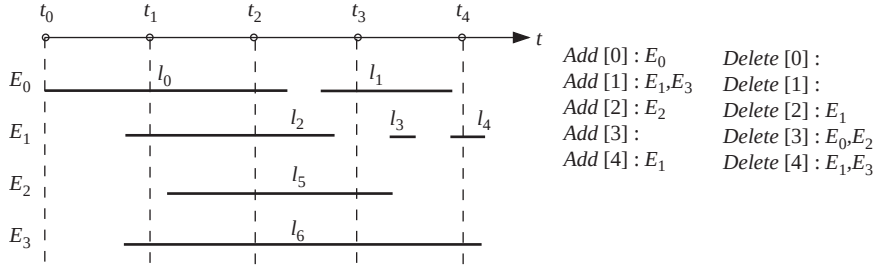

Fig. 2. Silhouette time-intervals.

Table 1
The performance analysis

Algorithm	Space complexity	Time complexity	
		Preprocessing	Silhouette extraction
Brute force	$O(N)$	Not required	$O(fN)$
Interval array	$O(N+k)$	$O(N)$	$O(fk)$
Interval tree	$O(N+k)$	$O(N+k\log_2 k)$	$O(f\log_2 k)$
Incremental update	$O(N+k)$	$O(N)$	$O(fM+k)$

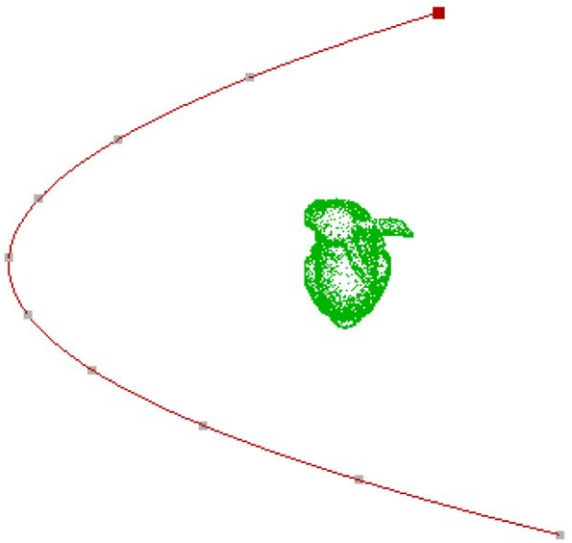

Fig. 3. Bunny model with viewpoint trajectory $\mathbf{q}(t)$.

edges would be included in at least one silhouette, was $\mathbf{q}(t) = (t, 0.5, -0.005t^2 + 500)$, $-350 \leqslant t \leqslant 400$, with different numbers of frames. Fig. 3 shows the path $\mathbf{q}(t)$ with one of the test models, *bunny*. The viewpoint trajectory used was a quadratic curve, as when the viewpoint trajectory is a curve with a degree higher than two, it can be approximated with a sequence of quadratic curves based on applying a slightly modified version of Chaikin's algorithm [17]. The preprocessing was then performed by intersecting the set of quadratic curves with the model faces Fig. 4.

Table 2 shows the features of the test models, including the number of model edges, average number of silhouette edges for each frame, and percentage of silhouette edges over the total number of model edges.

With the proposed incremental update algorithm, preprocessing is required with every new viewpoint trajectory, thus Table 3 shows the preprocessing time required when $\mathbf{q}(t)$ was given as the viewpoint trajectory for each model. In contrast, the brute force algorithm simply tests whether or not each model edge is contained in the silhouette from each viewpoint, so no preprocessing is involved. Meanwhile, although the anchored cone algorithm requires at least several minutes of preprocessing to construct the search tree structure, once this preprocessing is completed, it does not need to be repeated, thus no preprocessing time was considered for the experiments.

The frame rate for the incremental update algorithm, including its preprocessing time, was compared with that for the anchored cone algorithm, excluding its preprocessing time. The graphs in Fig. 5 show the frame rate for the three algorithms for the viewpoint trajectory $\mathbf{q}(t)$ with different numbers of frames for the four models. The average step size of the viewpoint movement for 1000, 2000, 3000, 4000, and 5000 frames was 1.66, 0.83, 0.55, 0.41, and 0.33, respectively. The number of frames was found to be inversely proportional to the step size of the viewpoint movement. The frame rate was the number of frames for which a silhouette could be computed within one second.

The frame rates for the proposed algorithms were also measured, excluding the rendering time. The frame rates labeled (c) and (p+c) in Fig. 5 represent just the computation time for the silhouettes and the combined preprocessing and computation time, respectively. Even though the proposed incremental update algorithm required the preprocessing steps of computing the silhouette time-intervals and constructing the *Add* and *Delete* structures, when the step size of the viewpoint movement was large, i.e. when the number of frames was 1000, the incremental update algorithm worked 9.37–15.03 times faster than the brute force algorithm and 2.67–4.85 times faster than the anchored cone algorithm. Also, when the step size of the viewpoint movement was smaller, the performance of the incremental update algorithm improved. For example, when the silhouettes for 5000 frames were extracted, the experimental results showed that the incremental update algorithm worked 34.45–41.13 times faster than the brute force algorithm and 8.55–13.63 times faster than the anchored cone algorithm.

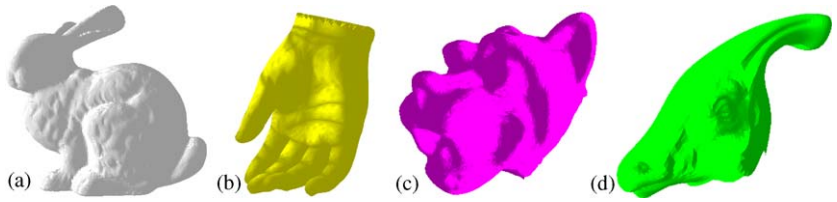

Fig. 4. Models: (a) *bunny*, (b) *hand*, (c) *gargoyle* and (d) *parasaur*.

Table 2
Model features

Model	Number of edges	Average number of silhouette edges
bunny	104,445	2942 (2.82%)
hand	91,284	2087 (2.29%)
gargoyle	25,083	1048 (4.18%)
parasaur	11,531	515 (4.47%)

Table 3
Preprocessing time for incremental update algorithm

Number of frames	*bunny* (ms)	*hand* (ms)	*gargoyle* (ms)	*parasaur* (ms)
1000	203.00	177.00	52.00	36.67
2000	203.00	177.33	62.67	36.33
3000	213.00	187.33	62.67	36.33
4000	213.67	182.33	62.67	31.33
5000	208.00	182.00	62.67	42.00

8. Conclusions

Algorithms were presented for the fast extraction of a sequence of perspective silhouettes for a polyhedral model from a moving viewpoint. The viewpoint is assumed to move along a trajectory $\mathbf{q}(t)$ that is a space curve of the time parameter t. As such, time-intervals are determined when each edge of the model is included in the silhouette based on two major computations: (i) intersecting $\mathbf{q}(t)$ with two planes and (ii) a number of dot products. If $\mathbf{q}(t)$ is a curve of degree n, then there are at most $n+1$ time-intervals for an edge to be included in the silhouette. Thus, two algorithms are proposed: a search-based algorithm and incremental update algorithm, which extract the silhouettes using the silhouette time-interval information.

The incremental update algorithm is an optimal solution for sequential silhouette extraction, and experiments demonstrated that the performance of the incremental update algorithm was better than that of other algorithms. With minor modifications the proposed algorithm can also be extended to compute parallel silhouettes. Future work includes the efficient computation of silhouette sequences for moving polyhedral models and solving the visibility problem.

Acknowledgements

The authors would like to thank Dr. Pedro Sander and his colleagues for providing the source code for the anchored cone algorithm. Dr. Kim was supported by the Korea Research Foundation Grant funded by Korean Government (MOEHRD) (R04-2004-000-10099-0). Dr. Baek was supported by the Kyungpook National University Research Fund, 2004.

Appendix A. Code of incremental update algorithm

```
void DrawIncUpdateSilhouette (int j)
// Input: j, which is a frame identification number
// Output: S(j), which is a pointer to the silhouette edge
list of frame j
{
 int SilPtr = IntNULL, SilPrePtr = IntNULL;
 // IntNULL was defined as −1
 struct EdgeList *ptr = NULL, *preptr = NULL;
 S(j) = S(j-1);


 // Attach the edge list pointed by Add[j] to S(j)
 ptr = Add[j];
 if (ptr != NULL) {
  if (j == 0) S(j) = ptr→Eid;
  if (SilEnd != IntNULL) E[SilEnd].link = ptr→Eid;
  while (ptr != NULL) {
   E[ptr→Eid].isSilhouette = TRUE;
   if (ptr→link != NULL) E[ptr→Eid].link =
   (ptr→link)→Eid;
   preptr = ptr;
   ptr = ptr→link;
  }
 SilEnd = preptr→Eid;
 }
 E[SilEnd].link = IntNULL;
```

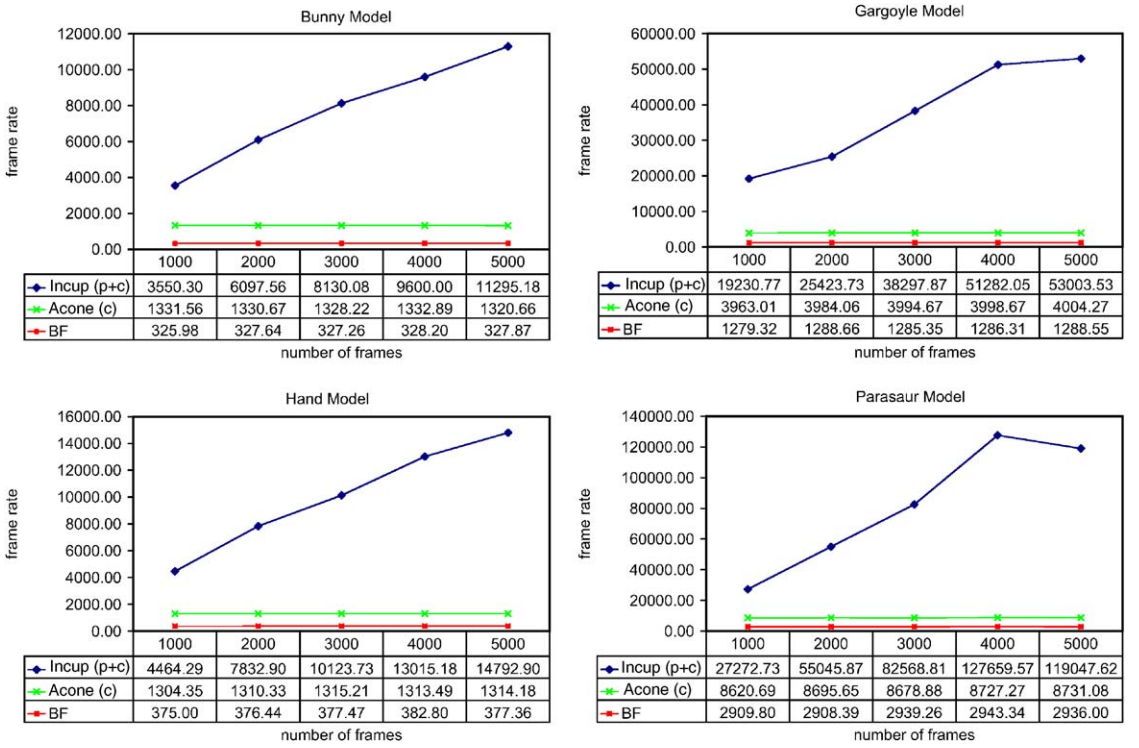

Fig. 5. Comparison of three algorithms.

```
//Draw silhouette edges and remove non-silhouette
edges from S(j)
SilPtr = S(j);
while (E[SilPtr].isSilhouette == FALSE && SilPtr != IntNULL) SilPtr = E[SilPtr].link;
if (SilPtr != IntNULL) S(j) = SilPtr;
SilPrePtr = SilPtr;
while (SilPtr != IntNULL) {
 if (E[SilPtr].isSilhouette == FALSE) {
  while (SilPtr != IntNULL && E[SilPtr].isSilhouette
  == FALSE)
                SilPtr = E[SilPtr].link;
  E[SilPrePtr].link = SilPtr;
 }
 if (E[SilPtr].isSilhouette == TRUE)
 DrawOneEdge(SilPtr); // Draw one silhouette edge
 if (SilPtr == IntNULL) break;
 SilPrePtr = SilPtr;
 SilPtr = E[SilPtr].link;
}
SilEnd = SilPrePtr;


//Set the isSilhouette field of the edges of S(j) those who are in Delete[j] as FALSE
ptr = Delete[j];
while (ptr != NULL) {
 E[ptr→Eid].isSilhouette = FALSE;
 ptr = ptr→link;
}
return S(j);
}
```

References

[1] Sander PV, Gu X, Gortler SJ, Hoppe H, Snyder J. Silhouette clipping. In: Akeley K, editor. Siggraph 2000, computer graphics proceedings. Annual conference series. New York/Reading, MA/New York: ACM Press/ACM SIGGRAPH/Addison-Wesley/Longman; 2000. p. 327–34.

[2] Martin R, Stephenson P. Sweeping of three-dimensional objects. Computer-Aided Design 1990;22(4):223–34.

[3] Weld J, Leu M. Geometric representation of swept volumes with application to polyhedral objects. The International Journal of Robotics Research 1990;9(5): 105–17.

[4] Isenberg T, Freudenberg B, Halper N, Schlechtweg S, Strothotte T. A developer's guide to silhouette algorithms for polygonal models. IEEE Computer Graphics and Applications 2003:28–37.

[5] Benichou F, Elber G. Output sensitive extraction of silhouettes from polygonal geometry. In: Proceedings of the pacific graphics 1999, 1999. p. 60–9.

[6] Hertzmann A, Zorin D. Illustrating smooth surfaces. In: Akeley K, editor. Siggraph 2000, computer graphics Proceedings. Annual conference series. New York/Reading, MA/New York: ACM Press/ACM SIGGRAPH/ Addison-Wesley/Longman; 2000. p. 517–26.
[7] Markosian L, Kowalski MA, Trychin SJ, Bourdev LD, Goldstein D, Hughes JF. Real-time nonphotorealistic rendering. In: Whitted, T, editor. SIGGRAPH 97 conference proceedings, annual conference series, ACM SIGGRAPH. Reading, MA: Addison-Wesley; 1997. p. 415–20.
[8] Pop M, Barequet G, Duncan CA, Goodrich M, Huang W, Kumar S. Efficient perspective–accurate silhouette computation and applications. In: COMPGEOM: annual ACM symposium on computational geometry; 2001. p. 60–8.
[9] Deussen O, Strothotte T. Computer-generated pen-and-ink illustration of trees. In: Proceedings of the SIGGRAPH 2000, computer graphics, vol. 34. New York: ACM Press; 2000. p. 13–8.
[10] Hertzmann A. Introduction to 3D non-photorealistic rendering: silhouettes and outlines. New York: ACM Press; 1999.
[11] Gooch B, Sloan P, Gooch A, Shirley P, Riesenfeld R. Interactive technical illustration. In: Proceedings of the symposium on interactive 3D graphics; 1999. p. 31–8.
[12] Raskar R, Cohen M. Image precision silhouette edges (color plate S. 231). In: Spencer SN, editor. Proceedings of the conference on the 1999 symposium on interactive 3D graphics. New York: ACM Press; 1999. p. 135–40.
[13] Saito T, Takahashi T. Comprehensible rendering of 3-d shapes. In: Proceedings of the SIGGRAPH 1990, vol. 24. New York: ACM Press; 1990. p. 197–206.
[14] Gu X, Gortler SJ, Hoppe H, Mcmillan L, Brown B, Stone A, Silhouette mapping. Technical Report, TR-1-99, Department of Computer Science, Harvard University, March 1999.
[15] H. Edelsbrunner, Dynamic data structures for orthogonal intersection queries. Technical Report, Technische Universitat Graz, Austria, 1980.
[16] Preparata FP, Shamos M. Computational geometry. New York: Springer; 1985.
[17] Chaikin G. An algorithm for high speed curve generation. Computer Graphics and Image Processing 1974;3:346–9.

Ku-Jin Kim is an Assistant Professor in the Department of Computer Engineering at Kyungpook National University, Republic of Korea. Her research interests include computer graphics, non-photorealistic rendering, and geometric/surface modeling. Prof. Kim received a B.S. degree from Ewha Womans University in 1990, an M.S. degree from KAIST in 1992, and a Ph.D. degree from POSTECH in 1998, all in Computer Science. She was a PostDoctoral fellow at Purdue University in 1998–2000. Prof. Kim also held faculty positions at Ajou University, Korea, and at University of Missouri, St. Louis, USA.

Nakhoon Baek is currently an assistant professor in the School of Electrical Engineering & Computer Science at Kyungpook National University. He received his B.A., M.S., and Ph.D. in Computer Science from the Korea Advanced Institute of Science and Technology in 1990, 1992, and 1997, respectively. His research interests include real-time rendering and computational geometry applications.