연속적인 윤곽선 계산을 위한 최적 알고리즘

김 구진, 백 낙훈

경북대학교 컴퓨터공학과, 경북대학교 컴퓨터과학과

kujinkim@yahoo.com, bnhoon@yahoo.co.kr

An Optimal Algorithm for Computing a Sequence of Silhouettes

Ku-Jin Kim, Nakhoon Baek

Dept. of Computer Engineering, Dept. of Computer Science

Kyungpook National University

요 약

본 논문에서는 주어진 경로를 따라 이동하는 관측점으로부터 다면체 모델의 윤곽선을 계산하기 위한 최적 알고리즘 (optimal algorithm)으로서 점증적 갱신 알고리즘 (incremental update algorithm)을 제안한다. 전처리 과정에서는, 윤곽선을 계산할 프레임(frame)의 갯수가 f라 할 때 각 프레임 $j, 0 \leq j < f$,에 대해 그 전 프레임 $j-1$ 에서 이미 구한 윤곽선에 추가되거나 제거되어야 할 에지의 리스트를 계산한다. 이것은 모델의 각 에지가 윤곽선에 포함되기 시작하는 프레임과 윤곽선으로부터 제거되는 프레임의 범위를 계산함으로써 수행된다. 점증적 갱신 알고리즘은 전처리 과정에서 구성한 에지 리스트를 이용하여 f 프레임의 윤곽선을 효율적이고 안정적으로 추출할 수 있으며, 각 프레임마다 전 프레임의 윤곽선 에지 리스트와 여기에 추가 또는 제거되어야 할 에지의 리스트만을 검색하므로 시간 복잡도(time complexity)의 측면에서 최적 알고리즘이다.

1. 서론

물체의 윤곽선 정보는 비사실적 영상 렌더링, 충돌 감지, backface culling, 물체의 단순화 등의 응용분야에서 유용하게 사용된다. Isenberg [7] 등은 기존의 윤곽선 계산 방법들을 image space 알고리즘, object space 알고리즘, 그리고 hybrid 알고리즘으로 분류하고 있다.

Image space 알고리즘 [2] [5]은 depth buffer 또는 normal buffer 등을 이용해서 buffer 안에 저장된 영상의 불연속성 (discontinuity)을 발견함으로써 물체의 윤곽선을 계산한다. Depth buffer는 물체 표면 상의 점들과 관측점 사이의 거리 정보를 pixel intensity로 표현한다. 이러한 depth map 에 존재하는 에지를 추출하면 이것이 물체가 렌더링된 영상에서 물체와 배경을 구분하는 경계선이 된다. 비슷한 방법으로 normal buffer는 물체의 normal 정보를 표현한다. Hertzmann [5]은 depth map과 normal map을 결합시킴으로써 물체의 윤곽선과 C^0, C^1 불연속성을 찾는 방법을 제안하고 있다. Hybrid 알고리즘 [3] [12] [13]은 물체를 구성하는 face들을 수정하고 z-buffer를 이용하여 수정된 face들을 렌더링함으로써 윤곽선을 추출한다. Raskar and Cohen [12]은 물체를 구성하는 face 들 중에서 관측점을 기준으로 backface는 조금 확대하여 그리고 frontface는 정상적인 크기로 그림으로써 윤곽선이 렌더링되게 하였다.

Image space 알고리즘이나 hybrid 알고리즘은 그래픽스 하드웨어의 특성을 활용하여 실시간에 윤곽선을 추출할 수 있고, 윤곽선을 렌더링하는 동안 visibility culling이 자동적으로 해결된다는 장점을 갖는다. 그러나, 이들 알고리즘의 결과로 생성되는 윤곽선은 3차원 공간 상에서 윤곽선을 이루는 에지들의 기하학적인 정보를 포함하지 않으므로 윤곽선에 대해 stylization을 하거나 부가적인 처리를 하기가 어렵다. 또한 윤곽선이 *low precision*으로 표현되며 *antialiasing* 문제를 야기시킨다는 단점을 갖는다.

Object space 알고리즘 [1] [6] [9] [10] [11] [14]은 윤

곡선을 정확히 계산하여 analytic하게 표현할 수 있고, 윤곽선에 *stylization*이나 기타 다른 연산을 적용하기에 용이하다. 그러나, 윤곽선을 계산하는 동안 visibility culling이 자동적으로 해결되지 않으므로 보통 윤곽선을 계산한 후에 visibility culling을 위한 별개의 과정이 필요하다.

Benichou and Elber [1]는 다면체 모델에 대해 주어진 관측 방향에 대한 parallel silhouette을 계산하는 연구를 수행했다. 이들은 다면체 모델 에지들에 대해 normal을 구하여 Gaussian sphere 상에서 원호로 표현하고 관측방향을 대원 (great circle)으로 표현한 뒤, Gaussian sphere를 둘러싸는 욱면체를 정의하여 여기에 원호와 대원을 투사하였다. 원호는 선분으로, 대원은 직선으로 투사되는데, 직선과 교차하는 선분들에 해당하는 물체 에지들을 추출하여 윤곽선을 구성할 수 있다.

Gu et al. [4]은 다면체 모델을 coarse mesh로 단순화하면서 여기에 original model의 exact silhouette을 결합시킴으로써 단순화된 모델이 사실감있게 보이도록 하는 방법을 사용했다. 이때 모델을 관측하는 관측점은 모델을 둘러싸는 sphere 상에서 이동한다고 가정하였으며, sampling된 sphere 상의 관측점들로부터 모델의 정확한 silhouette을 미리 계산하여 silhouette map을 구성하였다. 임의의 관측점에 대한 윤곽선은 silhouette map으로부터 이웃한 윤곽선들을 선택하여 근사(interpolation)함으로써 계산한다.

Sander et al. [14]은 모델 상의 인접한 face들의 normal vector들을 포함하는 anchored cone을 계산하고 이들 이용하여 *hierarchical search tree*를 구성하였다. Tree를 이용하여 윤곽선이 될 가능성이 없는 edge들 중 많은 갯수를 search에서 제외함으로써 윤곽선 계산의 효율성을 높였다.

Markosian et al. [9]은 모델 에지 집합에서 적은 갯수의 에지들만을 검색하여 윤곽선을 구성하기 위한 최초의 시작 에지를 선택하고, 시작 에지로부터 출발하여 윤곽선 에지들을 연속적으로 추적함으로써 하나의 연결된 윤곽선 구성요소를 발견한다. 이 방법은 윤곽선을 빠르게 추출할 수 있으나 윤곽선에 속하는 모든 구성요소를 발견한다고 보장할 수는 없다.

Hertzmann [6]은 다면체 모델을 곡면의 근사로 취급하였다. 관측점이 점 $\mathbf{q}$라 할 때, 다면체 모델의 각 vertex를 곡면 상의 점 $\mathbf{p}_i$라 가정하고, 점 $\mathbf{p}_i$에서의 normal vector $\mathbf{N}_i$에 대해 $(\mathbf{q}-\mathbf{p}_i)\cdot\mathbf{N}_i$을 계산한다. 하나의 모델 에지를 공유하는 양 끝의 vertex에 대해 이와 같은 계산을 수행한 후 계산결과값의 부호가 양, 음과 같이 서로 다른 경우 에지 상에서 계산 결과값이 0이 되는 점을 수치적 방법을 이용하여 계산하면, 이 점이 윤곽선을 구성하는 한 개의 점이 된다. 인접한 모델 에지들에 대해 같은 과정을 반복하며 발견한 점들을 연결하여 윤곽선을 추출한다.

Markosian et al. [9], Hertzmann [6] 의 연구는 윤곽선 에지 간의 spatial coherence를 이용하여 윤곽선을 계산하는 접근방법을 사용하였다. 이들과 다른 관점에서 접근하는 방법으로는, 관측점이 고정되어 있지 않고 시간에 따라 이동하는 경우 temporal coherence를 이용하여 연속적인 윤곽선을 계산하는 문제를 고려할 수 있다. Northrup and Markosian [10]은 temporal coherence를 이용하여 interactive하게 이동하는 관측점에 대해 변화하는 윤곽선을 계산하였다. 윤곽선 계산은 Markosian [9]이 제시한 방법을 사용하지만, 윤곽선에 포함되는 하나 이상의 연결된 구성요소를 모두 발견하기 위하여 윤곽선의 시작 에지를 각 구성요소마다 한 개씩 계속 유지한다. 한 프레임에서의 윤곽선을 구성하기 위해서 그 전 프레임에서 시작 에지로 사용된 에지들 주변의 에지를 탐색하여 새로운 시작 에지를 찾아나간다. 이 방법은 효율적으로 object space 윤곽선을 계산할 수 있지만, 윤곽선의 구성요소가 새로 발생하거나 없어지는 경우에 대한 처리가 어렵고 시작 에지의 정보를 갱신시키는 비용이 필요하다. 또한, 각 프레임마다 전체 윤곽선을 새로 추적해야 한다는 단점이 있다.

Pop et al. [11]은 변화하는 관측점에 대하여 윤곽선 전체를 다시 계산하기 보다는 윤곽선에서 변화하는 부분만을 갱신하기 위한 방법을 제시하였다. 모델 에지들의 normal vector를 dual line segment로 매핑(mapping)하고 관측점은 dual plane으로 매핑하면 dual plane과 교차하는 dual line segment를 계산하여 윤곽선 에지들을 추출할 수 있다. 관측점의 변화는 dual plane의 변화로 대응되므로 한 프레임 f_1에서의 윤곽선을 일단 계산하면, 그 다음의 프레임 f_2에서는 그 전의 윤곽선에서 제거해야 할 에지와 추가해야할 에지만 발견하여 윤곽선을 점증적으로 수정할 수 있다. Pop et al.은 두 개의 dual plane 내부에 속한 dual line segment의 끝점을 발견함으로써 변동이 있는 에지들을 발견하고 이를 이용하여 윤곽선에서 변화하는 부분만을 갱신하는 방법을 제시했다. 이 방법을 이용하면 매 프레임마다 전체 윤곽선을 다시 계산할 필요 없

이, 필요한 부분만을 찾아내어 갱신하므로 연속적으로 변화하는 윤곽선을 효율적으로 계산할 수 있다. 그러나, 두 개의 dual plane 사이의 공간에 포함되는 점들을 효율적으로 찾기가 어렵기 때문에 heuristic을 사용해야만 하고, dual space를 사용하기 때문에 degenerate case들을 고려해야 하며 dual space의 구성 과정에서 에러가 누적될 수 있다는 단점이 있다.

만약 관측점이 interactive하게 주어지지 않고, 어떤 경로 상에서 이동한다는 제약이 가해진다면, 위의 방법들에 비해 좀더 효과적으로 윤곽선을 계산할 방법이 있을 것이다. 본 저자들은 논문 [8]에서 다음의 문제를 정의하고 알고리즘을 제시한 바 있다.

> 시간에 따라 변화하는 관측점의 궤적 $\mathbf{q}(t)$와 다면체 모델이 주어질 때, t의 변화에 따라 연속적으로 변화하는 윤곽선을 효율적으로 계산하는 방법은 무엇인가?

논문 [8]에서는 이 문제의 해결을 위하여 silhouette time interval, 즉 하나의 모델 에지가 윤곽선에 포함되는 시간 간격,을 계산하여 interval tree를 구성한 후, 주어진 시점 (time point)을 포함하는 모든 interval 들을 발견함으로써 윤곽선을 효율적으로 계산하였다.

본 논문에서는 silhouette time interval이라는 기본적인 아이디어를 이용하여 개발한 최적 알고리즘 (optimal algorithm)으로서 점증적 갱신 알고리즘을 소개한다. 점증적 갱신 알고리즘은 전처리과정에서 silhouette time interval을 이용하여 각 프레임마다 윤곽선에 추가하거나 제거해야 하는 에지 리스트를 구성한다. 전처리 과정을 거친 후, 점증적 갱신 알고리즘은 한 프레임의 윤곽선을 계산하기 위하여 그 전 프레임에서 이미 계산된 윤곽선과 그로부터 제거되어야 할 에지들, 그리고, 새로 추가해야 할 에지들을 한번씩만 방문하기 때문에 최적 알고리즘이라 할 수 있다.

본 논문의 구성은 다음과 같다. 제 2절에서는 점증적 갱신 알고리즘과 그 전처리과정을 제시한다. 제 3절에서는 점증적 갱신 알고리즘을 이용하여 윤곽선을 계산한 실험 결과를 보인다. 제 4절에서는 결론을 내린다.

2. 점증적 갱신 알고리즘

2.1 전처리과정

모델 에지의 집합을 $\{E_i \mid 0 \leq i < N\}$,라 할 때, E_i를 공유하는 두 개의 face는 F_1, F_2으로, 모델 외부를 향하는 이들의 normal vector를 각각 N_1, N_2로 표기한다. 이들 face를 포함하는 두 개의 평면을 P_1, P_2라 하면 (그림 1(a)), 두 개의 평면은 3차원 공간을 네 개의 영역으로 나눈다. 평면 P의 normal vector를 포함하는 half-space를 P^+, 반대 방향의 half-space를 P^-라 하면, 네 개의 영역은 $P_1^+ \cap P_2^+$, $P_1^+ \cap P_2^-$, $P_1^- \cap P_2^+$, 그리고, $P_1^- \cap P_2^-$이다 (그림 1(b)).

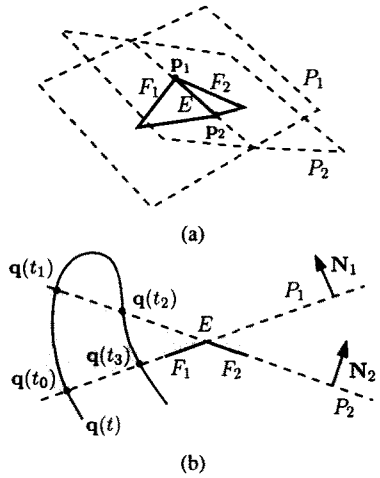

그림 1: 에지 E가 윤곽선에 포함되는 시간 간격.

관측점의 경로가 $\mathbf{q}(t)$, $t_{min} \leq t \leq t_{max}$,로 주어진다고 가정하면, 각 에지의 silhouette time interval은 다음과 같이 계산된다.

$$\{t \mid \mathbf{q}(t) \cap ((P_1^+ \cap P_2^-) \cup (P_1^- \cap P_2^+))\}$$

그림 1(b)의 경우, 회색영역이 $(P_1^+ \cap P_2^-) \cup (P_1^- \cap P_2^+)$에 해당하므로, 에지 E는 두 개의 silhouette time interval $(t_0, t_1), (t_2, t_3)$를 갖는다.

윤곽선을 구하기 위한 프레임의 갯수가 f로 주어지면, 관측점 $\mathbf{q}(t_j)$, $0 \leq j < f$,에서 모델에 대한 윤곽선을 구하게 되는데, 모델을 $\mathbf{q}(t_j)$로부터 관측하여 얻는 윤곽선을 $S(t_j)$라고 표기한다. 전처리 과정에서는 관측점을 결정하는 시점 t_j에서 윤곽선 $S(t_j)$를 계산하기 위하여, 그 전 프레임의 윤곽선, 즉 $S(t_{j-1})$,로부터 제거되어야 할 에지들과 새로 추가되어야 할 에지들을 계산하며, 이러한 에지들은 각각 $Delete[j]$와 $Add[j]$라는 에지 리스트에 저장된다. 각 에지의 sil-

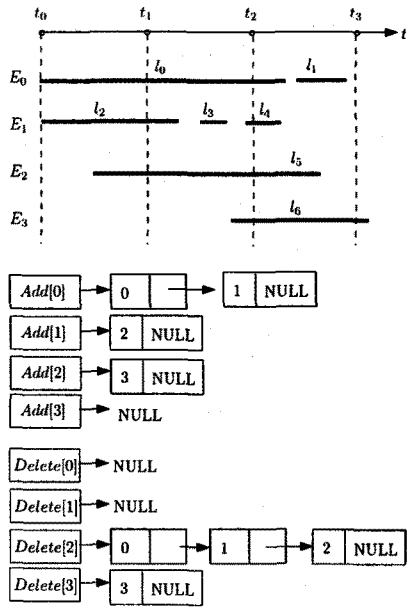

그림 2: Silhouette time interval과 *Add, Delete*의 구성.

houette time interval을 이용하면 *Add, Delete* 에지 리스트를 구성할 수 있다.

그림 2에서는 네 개의 에지 E_i, $0 \leq i < 4$,의 silhouette time interval들로부터 각 프레임에 대한 *Add, Delete* 리스트를 구성한 예를 보인다. 그림에서 t축을 따라 윤곽선을 계산할 시점 t_j가 표시되어있다. 에지 E_0의 경우 silhouette time interval은 l_0와 l_1인데, l_0를 이용하면, E_0가 프레임 0부터 2까지 윤곽선에 포함된다는 것을 알 수 있다. 따라서, E_0는 각각 *Add*[0]와 *Delete*[2]에 삽입된다. 또다른 silhouette time interval l_1은 연속된 두 개의 프레임 사이에 속하므로, *Add*나 *Delete*의 구성에 영향을 미치지 않는다. 에지 E_1과 같은 경우는 세개의 silhouette time interval l_2, l_3, l_4를 갖지만, 프레임을 기준으로 할 경우, fram 0부터 2까지 윤곽선에 포함된다는 것을 알 수 있다. 그러므로, E_1는 각각 *Add*[0]와 *Delete*[2]에 삽입된다.

2.2 알고리즘

각 시점 t_j에서 윤곽선 $S(t_j)$를 추출하는 점증적 갱신 알고리즘은 에지의 자료구조를 다음과 같이 구성하여 구현한다.

```
struct Edge {
    int vertex[2]; /* Edge의 양 끝점 */
    bool Sil; /* 에지의 윤곽선 포함 여부 */
    int link; /* 다음 윤곽선 에지의 인덱스값 */
} E[NumOfEdges];
```

임의의 시점 t_j에서 윤곽선 $S(t_j)$를 계산하기 위해서는 윤곽선 $S(t_{j-1})$이 미리 계산되어 여기에 속하는 윤곽선 에지들의 *Sil* field 값은 ON으로 setting되어 있고, *link* field는 다음 윤곽선 에지의 인덱스를 포함하고 있다고 가정한다. 그러면, $S(t_j)$를 추출하는 과정은 다음 알고리즘과 같다. 이 알고리즘에서 $S(t_j)$는 윤곽선 에지들의 리스트를 가리키는 pointer로 사용되고 있다.

Algorithm:Incremental_Update_Silhouette

Input: 다면체 모델, 관측점 경로 $\mathbf{q}(t)$, $t_{min} \leq t \leq t_{max}$, 프레임의 갯수 f

Output: 각 프레임의 윤곽선 $S(t_j), j = 0, 1, \cdots, f$

// 전처리과정

Step 1 *Add, Delete* 에지 리스트를 구성한다.

Step 2 각 에지의 *Sil* field 값에 OFF를 setting한다.

// 윤곽선 추출

Step 3 For $j := 0$ to f do,

(a) 만약 $j = 0$이면,

$S(t_j) := NULL$;

그렇지 않으면,

$S(t_j) := S(t_{j-1})$;

(b) *Add*[j]에 포함된 에지 리스트를 검색하면서 해당 에지의 *Sil* field 값을 ON으로 변경하고, *link* field가 *Add*[j]에 속한 다음 에지를 가리키도록 변경한다.

(c) *Add*[j]에 속한 마지막 에지의 *link* field가 $S(t_j)$의 첫번째 에지를 가리키게 한다.

(d) $S(t_j)$는 *Add*[j]의 첫번째 에지를 가리키게 한다.

(e) $S(t_j)$가 가리키는 에지 리스트에서 *Sil* field가 OFF인 것들을 삭제하면서 윤곽선을 추출한다.

(f) $S(t_j)$가 가리키는 에지 리스트에서 *Delete*[j]에 속한 에지들의 *Sil* field를 OFF로 변경한다.

3. 알고리즘의 성능분석 및 실험결과

Brute force 알고리즘은 각 프레임마다 모델의 모든 에지에 대해 윤곽선에 포함되는 지의 여부를 검사한다. Brute force 알고리즘, 논문 [8]에서 제시한 interval tree를 이용한 윤곽선 계산 알고리즘, 그리고 점증적 갱신 알고리즘의 시간과 공간복잡도를 분석한 결과는 표 1과 같다. 이 표에서 N은 모델 에지의 갯수, k는 silhouette time interval의 갯수, M은 각 프레임당 윤곽선에 속하는 에지의 평균 갯수, 그리고 f는 프레임의 갯수이다.

알고리즘들은 Pentium 4 (1.70 GHz), 512MB의 RAM을 가진 PC에서 C 프로그래밍 언어와 OpenGL을 이용하여 구현되었다. 실험한 관측점의 경로는 $\mathbf{q}_{close}(t) = (t, 0.5, -0.005t^2 + 350)$, $-300 \leq t \leq 150$, 와 $\mathbf{q}_{far}(t) = (t, 0, -0.005t^2 + 500)$, $-100 \leq t \leq 100$이다. 경로 $\mathbf{q}_{close}(t)$ 는 $\mathbf{q}_{far}(t)$보다 길고 모델과 더 가까운 거리에 있으므로 $\mathbf{q}_{close}(t)$의 경우에 비해 더 많은 갯수의 silhouette time interval 을 생성한다.

표 2는 네 개의 다면체 모델 bunny, hand, dragon, *Buddha*에 대해 관측점 경로 $\mathbf{q}_{close}(t)$, $\mathbf{q}_{far}(t)$ 를 적용할 때 각 모델이 갖는 silhouette time interval의 갯수를 보인다. **num_edges** 열은 각 모델의 에지의 갯수를 나타내고 **ave_num_sil_edge** 열은 각 프레임 당 윤곽선에 포함된 에지의 평균갯수, 그리고 **num_intv** 열은 모델의 silhouette time interval 갯수를 나타낸다.

표 3과 표 4는 각각 관측점 경로 $\mathbf{q}_{close}(t)$와 $\mathbf{q}_{far}(t)$에 대해 윤곽선 계산 알고리즘을 적용하여 계산한 결과이다. 전처리과정에 걸리는 시간과 실제 윤곽선을 추출하는 데 걸리는 시간은 초 단위로 표시하였고, 윤곽선 추출 시에 걸리는 시간을 프레임 rate로 환산한 결과가 표시되어 있다. Frame rate는 전처리과정에 걸린 시간과 추출한 윤곽선을 렌더링하는 데 걸리는 시간을 배제하고 윤곽선 추출에 걸리는 계산 시간만으로 측정되었다. 표 3과 표 4의 결과를 비교할 때, 전처리과정에 걸리는 시간이나 frame rate 면에서 점증적 갱신 알고리즘이 interval tree를 이용한 방법에 비해 효율적이라는 것을 볼 수 있다.

그림 3의 왼쪽 열에서는 각 모델을 렌더링한 결과를, 그리고 오른쪽 열에서는 $\mathbf{q}_{far}(t)$에 대해 모델의 윤곽선을 계산할 경우 한번이라도 윤곽선에 속하는 모든 에지들을 그리고 있다.

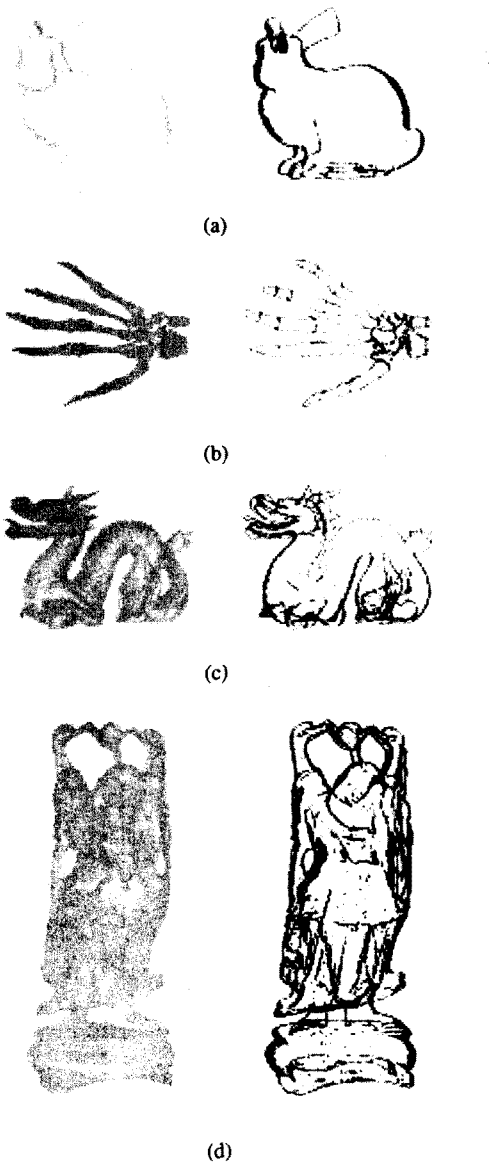

그림 3: 다면체 모델과 관측점 경로 $\mathbf{q}_{far}(t)$에 대해 윤곽선에 속할 수 있는 모든 에지 (위로부터 (a) Stanford bunny, (b) skeleton hand, (c) dragon, and, (d) happy Buddha).

Algorithm	공간복잡도	시간복잡도	
		Preprocessing	Silhouette Extraction
Brute Force	not required	not required	O(fN)
Interval Tree	O(k)	O($N + k \log_2 k$)	O($f \log_2 k$)
Incremental Update	O($N + k$)	O(N)	O($fM + k$)

표 1: 알고리즘들의 성능분석.

Model	num_edges	$\mathbf{q}_{close}(t)$의 경우		$\mathbf{q}_{far}(t)$의 경우	
		ave_num_sil_edge	num_intv	ave_num_sil_edge	num_intv
bunny	104,177	2,917 (2.80%)	78,109	2,472 (2.37%)	13,448
hand	981,999	19,677 (2.00%)	654,891	14,789 (1.506%)	70,800
dragon	1,307,102	28,281 (2.16%)	859,114	25,056 (1.917%)	124,954
Buddha	1,631,574	45,604 (2.79%)	1,073,063	43,150 (2.64%)	220,980

표 2: 관측점 경로 $\mathbf{q}_{close}(t)$, $\mathbf{q}_{far}(t)$에 대한 평균 윤곽선 에지의 갯수 및 silhouette time interval의 갯수 (600 frames).

Model	Performance	BruteForce	IntvTree	IncUpdate
bunny	preprocessing (sec)	0.00	0.82	0.30
	computation(sec)	9.34	0.36	0.07
	frame rate	64.22	1666.67	8571.43
hand	preprocessing (sec)	0.000	9.06	2.44
	computation (sec)	49.62	3.68	1.93
	frame rate	12.09	162.82	310.40
dragon	preprocessing (sec)	0.00	13.01	3.30
	computation (sec)	84.57	5.52	3.06
	frame rate	7.09	108.74	195.82
Buddha	preprocessing (sec)	0.00	16.68	4.01
	computation (sec)	112.83	8.99	4.95
	frame rate	5.32	66.72	121.29

표 3: 관측점 경로가 $\mathbf{q}_{close}(t)$인 경우 알고리즘의 비교.

Model	Performance	BruteForce	IntvTree	IncUpdate
bunny	preprocessing (sec)	0.00	0.19	0.16
	computation (sec)	9.50	0.13	0.03
	frame rate	63.13	4615.39	20000.00
hand	preprocessing (sec)	0.000	1.54	1.05
	computation (sec)	49.75	1.54	1.12
	frame rate	12.06	389.11	534.76
dragon	preprocessing (sec)	0.00	2.57	1.54
	computation(sec)	87.76	4.05	2.09
	frame rate	6.84	148.29	286.67
Buddha	preprocessing (sec)	0.000	4.07	2.09
	computation (sec)	111.10	7.13	3.70
	frame rate	5.40	84.14	161.94

표 4: 관측점 경로가 $\mathbf{q}_{far}(t)$인 경우 알고리즘의 비교.

표 1에 의하면 윤곽선 추출에 걸리는 시간에 가장 영향을 미치는 것은 silhouette time interval의 갯수 k이다. 그러므로, k의 값을 변화시키며 세 개의 알고리즘들의 성능을 비교해 볼 수 있다. k 값을 변화시키기 위해 $\mathbf{q}_{close}(t)$, $t_{min} \leq t \leq t_{max}$,의 길이를 변화시키는데, t_{min}은 -300으로 고정하고 t_{max}의 값을 변화시키며 실험을 수행했다. 600 개 프레임의 윤곽선을 다양한 길이의 관측점 경로를 이용하여 계산한 결과가 그림 4의 그래프에서 제시된다. 이 실험에서 점증적 갱신 알고리즘이 brute force 알고리즘이나 interval tree 알고리즘보다 월등하게 효율적인 결과를 얻음을 알 수 있다.

4.결론

본 논문에서는 주어진 경로 상에서 이동하는 관측점으로부터 다면체 모델의 윤곽선을 효율적으로 구하는 최적 알고리즘으로 점증적 갱신 알고리즘을 제안하였다. 관측점은 곡선 경로 $\mathbf{q}(t)$ 상에서 이동한다고 가정한다. 그러면 각 모델 에지에 대해서 구한 silhouette time interval 정보를 이용하여 각 프레임마다 그 전 프레임의 윤곽선으로부터 추가되거나 제거되어야 할 에지의 리스트를 구성할 수 있다. 전처리과정에서 얻어진 이러한 에지 리스트를 이용하여 변화하는 관측점에 대해 변화하는 윤곽선을 점증적으로 갱신시키며 계산할 수 있다.

참고 문헌

[1] F. Benichou and G. Elber. Output sensitive extraction of silhouettes from polygonal geometry. In *Proc. of Pacific Graphics 1999*, pages 60–69, october 1999.

[2] O. Deussen and T. Strothotte. Computer-generated pen-and-ink illustration of trees. In *Proc. of SIGGRAPH 2000, Computer Graphics*, volume 34, pages 13–18. ACM Press, 2000.

[3] B. Gooch, P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *Proc. of Symposium on Interactive 3D Graphics*, pages 31–38, 1999.

[4] X. Gu, S. J. Gortler, H. Hoppe, L. Mcmillan, B. Brown, and A. Stone. Silhouette mapping. Technical report, TR-1-99, Dept. of Computer Science, Havard University, march 1999.

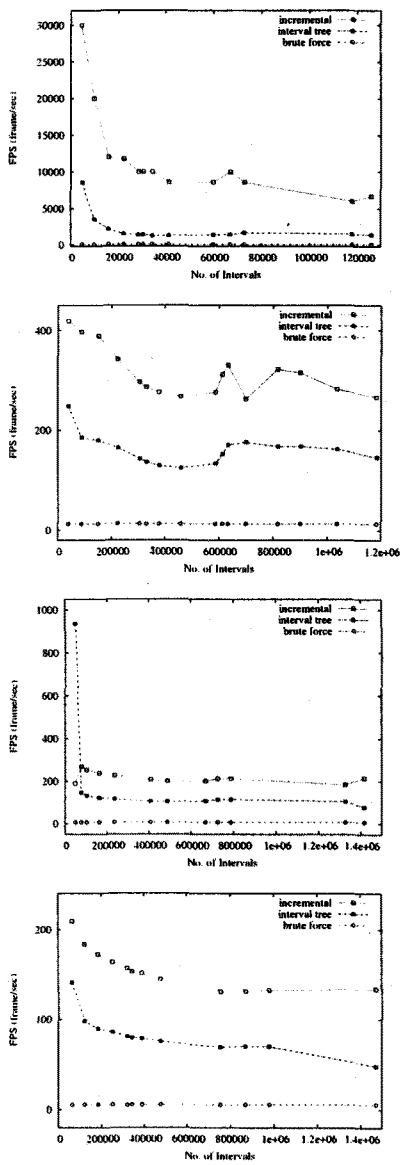

그림 4: 알고리즘들의 frame rate 비교 (위로부터 차례대로 bunny, hand, dragon, Buddha 모델에 대한 frame rate).

[5] A. Hertzmann. *Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines.* ACM Press, 1999.

[6] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 517–526. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

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

[8] K.-J. Kim and N.-H. Baek. Extracting silhouettes of a polyhedral model from a curved viewpoint trajectory (in korean). *Journal of the Korea Computer Graphics Society*, pages 1–7, June 2002.

[9] L. Markosian, M. A. Kowalski, S. J. Trychin, L. D. Bourdev, D. Goldstein, and J. F. Hughes. Real-time nonphotorealistic rendering. In T. Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, pages 415–420. ACM SIGGRAPH, Addison Wesley, Aug. 1997.

[10] J. Northrup and L. Markosian. Artistic silhouettes: A hybrid approach. In *Proc. of NPAR*, 2000.

[11] Pop, Barequet, Duncan, Goodrich, Huang, and Kumar. Efficient perspective-accurate silhouette computation and applications. In *COMPGEOM: Annual ACM Symposium on Computational Geometry*, 2001.

[12] R. Raskar and M. Cohen. Image precision silhouette edges (color plate S. 231). In S. N. Spencer, editor, *Proc. of the Conference on the 1999 Symposium on interactive 3D Graphics*, pages 135–140. ACM Press, Apr. 26–28 1999.

[13] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *Proceed. of SIGGRAPH 1990*, volume 24, pages 197–206. ACM Press, 1990.

[14] P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe, and J. Snyder. Silhouette clipping. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 327–334. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.