공학석사학위논문

투영 기하를 이용한 시점 의존적 그림자맵 방법

View-dependent Shadow Maps

Using Projective Geometry

2002 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

권 태 수

요약

그림자맵(shadow map)은 실시간 렌더링 같이 빛이나 물체가 자주 변하는 환경에서 좋은 방법이지만, 앨리어싱(aliasing)이 생긴다는 단점이 있다. 앨리어싱은 시점과 광원에 따라 그림자맵 화소를 화면에서 본 크기와 화면 화소의 크기가 다르기 때문에 생긴다. 이 논문에서는 그림자맵을 광원 시점에서 볼 때 시점에 따라 다른 비 균일한 격자(non-uniform grids)에서 샘플링(sampling)을 함으로써 좋은 화질을 얻는 방법을 제시한다. 이를 위해 물체를 대표 그림자 투영 평면으로 빛의 방향을 따라서 투영한 후 눈의 위치에서 렌더링 하는 투영 기하 방법(projective geometry)을 깊이 비교가 가능하도록 개선하고 이를 이용한 그림자맵 결과가 기존 방법에 비해 속도의 손해 없이 앨리어싱이 적어 질 수 있음을 보였다.

주요어 : 실시간 렌더링, 투영 기하, 그림자, 앤티앨리어싱

학번 : 2000-20715

목차

그림 목차

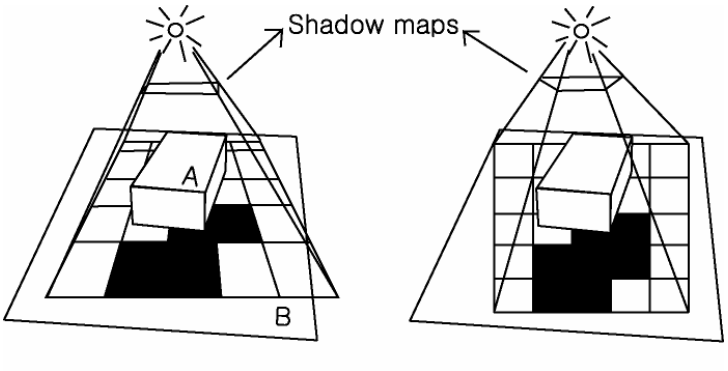
1. 서론

그림자는 화면을 실제적으로 보이게 하고 물체들 간의 상대적인 관계를 파악하도록 하는데 중요하다. 이런 그림자를 그리기 위한 방법은 많이 있지만 그 중 그림자맵은 그 장점으로 인해 자주 사용되는 방법 중 하나이다. 그림자맵 방법에서 그림자맵은 광원 시점에서 생성된 깊이 버퍼를 뜻한다. 그림자맵 방법을 사용해서 그림자를 그리고자 할 때는 화면의 각 화소에 대해서 그 화소의 광원시점으로 변환된 깊이와 그림자맵에 그려진 깊이를 비교하여 다르면 그림자로 판단한다.[2]

그림자맵 방법은 그림자맵 생성이 깊이버퍼를 이용한 렌더링으로 간단하고 일단 그림자맵이 생성되면 화면 화소의 그림자 여부 판단이 장면 복잡도에 영향을 받지 않기 때문에 효율적이다. 또한 하드웨어로 가속화 가능하여 실시간 렌더링에서 흔히 사용된다. 반면 그림자맵 알고리즘의 가장 큰 단점은 앨리어싱이다. 그림자맵의 한 화소가 광원시점에서 여러 화소 영역을 차지하는 경우 그림자 경계면에서 격자 모양의 앨리어싱이 생기게 된다. 다시 말해서 그림자맵 화소가 시점에서 가까운 곳에 그려지는 경우 그림자맵 화소의 확대로 인해 앨리어싱이 일어나게 되는데, 이를 없애려면 필요이상으로 고해상도인 그림자맵이 필요하게 된다.

(그림 1-1)은 물체 A 가 물체 B 에게 그림자를 드리운 것을 시점에서 본 그림이다. (그림 1-1-a)와 같이 기존의 광원시점에서 직사각형 격자로 그림자

맵이 만들어진 경우 B 에 생긴 그림자는 격자가 커지고, 이를 고치기 위해서 해상도를 높이면 시점에서 멀리 있는 부분이 지나치게 고해상도가 된다.



(a) (b)

그림 1-1 기존 그림자맵과 시점 의존적 그림자맵

본 논문에서는 (그림 1-1-b)와 같이 특정 평면에 대해서 그림자맵 격자가 화면에서 직사각형이 되도록 하는 방법을 제안 했다. 이는 물체들을 빛의 방향을 따라서 물체 투영 평면으로 투영 시킨 후 이 투영된 물체들을 눈의 시점에서 렌더링 해서 그림자맵을 생성하는 방법이다. 그 결과 물체 투영 평면에 위치하는 물체에 그림자가 생기는 경우 화면에서 그림자맵 격자가 직사각형이 되어 그림자맵 화소의 확대나 축소가 일어나지 않고 효율적으로 저장 된다.

2. 관련연구

실시간 렌더링에서 사용되는 그림자 알고리즘은 그림자 볼륨(shadow volume), 투영 그림자(projection shadows), 그림자 텍스쳐(shadow textures), 그림자맵(shadow map) 등이 있다.[7][14]

그림자 볼륨 방법[8][9]은 물체에 의해 빛이 가려지는 영역을 3 차원 다각형 볼륨으로 계산해서 그 안에 포함되는 부분을 그림자로 처리한다. 물체를 그린 후 그림자 볼륨의 앞면과 뒷면을 각각 그려 그려진 횟수의 차이를 가지고 그림자 여부를 판단한다. 스텐실버퍼(stencil buffer)를 사용하여 하드웨어로 가속화 될 수 있다. 화면 공간에서 화소마다 그림자 여부를 정확히 판단하므로 선명한 그림자를 얻을 수 있는 반면 모델이나 빛이 움직일 경우 그림자 볼륨을 다시 계산해야 한다는 단점과 다각형 개수가 많아지고 렌더링이 3 패스로 진행되는 단점이 있다.

투영 그림자 방법[10]은 다각형들을 그림자가 그려질 면에 투영시켜서 생기는 납작한 그림자 다각형을 생성해서 어둡게 렌더링 함으로써 그림자를 표현한다. 그림자 받는 면이 평면이어야 한다는 가정이 있고, 그림자 받는 면보다 그림자가 큰 경우나 그림자끼리 겹치는 경우 스텐실 버퍼를 사용하여 클립핑(clipping)을 해주어야 하는 단점이 있다. 이러한 단점에도 불구하고 간단히 구현될 수 있고, 화질이 좋고 빠르게 그려질 수 있어 게임처럼 제한된 상황에서 많이 사용된다.

그림자 텍스쳐 방법은 그림자 정보를 텍스쳐에 저장하여 그림자를 표현하는 방법이다. 그림자 텍스쳐는 모델이나 광원이 움직이지 않는 경우 다시 생성될 필요가 없다는 장점이 있다. 이러한 경우에는 래디오시티(radiosity) 같은 방법으로 고화질을 얻을 수 있다. 또는 투영 텍스쳐(projective texture) 방법[1]을 사용하여 빛의 시점에서 만들어진 텍스쳐를 그림자를 표현하는데 사용 할 수 있다. 오경수[6][11]는 제한된 형태의 모델에 대하여 빠르게 그림자 텍스쳐를 생성하는 방법을 제안 하였다.

그림자맵 방법[2]은 그림자 텍스쳐 방식과 마찬가지로 화상 기반 방법으로 그림자를 생성한다. 하지만 깊이 정보로부터 직접 그림자 정보를 뽑아내므로, 물체에 따라 다른 텍스쳐를 사용할 필요가 없이 공유가 가능하다. 이로 인해 그림자맵은 빛이나 물체가 움직였을 때 간단하게 그림자맵을 다시 생성하면 된다는 장점을 갖는다. Wolfgang Heidrich[13]에 의해 그림자맵 방법을 기존 하드웨어로 가속화 하는 법이 연구 되었다. 최근의 하드웨어에서는 직접 가속화 되어 그림자맵 생성과 렌더링이 각각 한 패스로 이루어 진다.[4]

이와 같은 장점을 가지는 그림자맵 방법은 앨리어싱이 생긴다는 큰 단점이 있다.

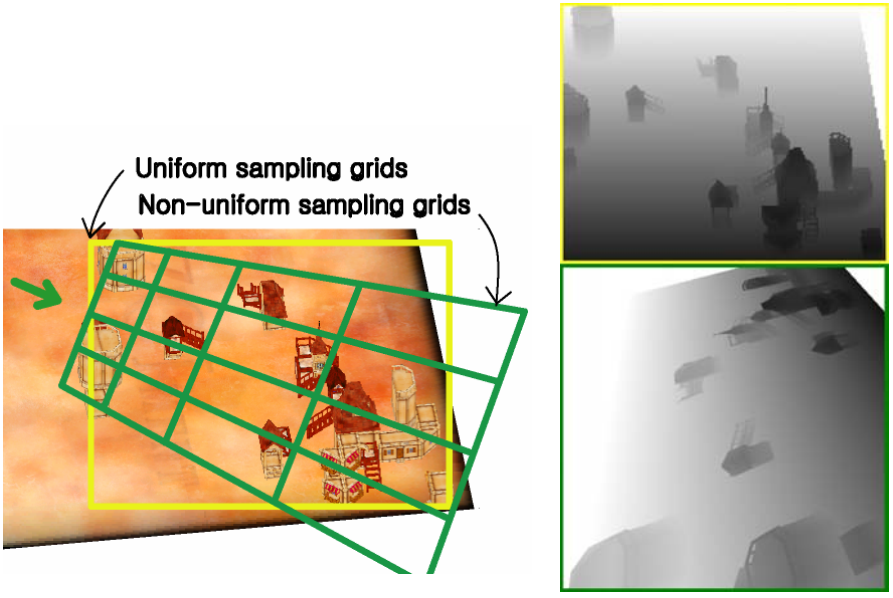
W. Reeves 는 텍스쳐에서의 바이리니어 필터링(bilinear filtering)에 대응 되는 퍼센티지 클로져 필터링(Percentage closer filtering) 방법을 제안하여 그림자맵의 화소가 화면공간에서 확대 되는 경우에 앨리어싱을 줄이는 법을 제

안 하였다.[3] 이 방법은 근사적인 의미의 필터링이지만 구현이 간단하여 현재 최신 하드웨어로 구현되어 사용되고 있다.

Adaptive shadow map 방법[5]에서는 보다 근본적으로 그림자맵 화소와 화면 공간 화소의 크기가 비슷해 지도록 하기 위하여, 계층적인 해상도의 격자 구조로 이루어진 그림자맵을 시점과 깊이에 따라서 적절히 사용하는 방법을 제안 하였다. 계층은 노드의 생성과 삭제로 표현되고 이때 LRU 방식으로 노드를 관리함으로써 시점이 고정되면 점진적으로 화질이 좋아지도록 했다. Deep shadow map 방법[12]에서는 기존 그림자맵이 그림자 각 화소에서 하나의 깊이 값 만을 저장하는 것과 비교해, 모든 깊이에 따른 빛의 투과율의 함수를 저장하도록 하는 방식을 사용했다. 이 방법으로 앤티앨리어싱(anti-aliasing)을 위한 필터링을 결과 화소에서 하지 않고 미리 할 수 있게 되어, 머리카락이나 반투명한 데이터에 사용가능하고 밉맵(mip-mapping)이 가능하다는 장점이 있다. 하지만 이런 시도들은 소프트 웨어로만 가속화가 된다는 단점이 있고, 하드웨어로 가속화 되는 범위에서 앨리어싱을 줄이려는 시도는 없었다.

3. 시점 의존적 그림자 맵

기존 방법의 그림자맵은 광원 시점에서 그림자맵을 생성하는 반면, 시점 의존적 그림자맵은 빛의 방향을 따라서 대표 그림자 평면으로 물체들을 투영시킨 뒤 이 투영된 물체를 눈의 시점에서 렌더링 해서 그림자맵을 생성한다. 그 결과 (그림 3-1)처럼 기존의 그림자맵은 광원 시점에서 직사각형 격자로 샘플링을 하는 반면(그림 3-1-b 위), 시점 의존적 그림자맵은 광원 시점에서 균일 하지 않게 샘플링 한다.(그림 3-1-b 아래) 이 방법을 통해서 대표 그림자 평면에 위치하는 물체가 그림자를 받는 모양대로 그림자맵이 생성된다.



(a) 광원 시점에서 본 장면 (b) 생성된 그림자맵

그림 3-1 기존 그림자맵과 시점 의존적 그림자맵 형태 비교

본 장에서는 투영된 물체를 눈의 시점에서 렌더링 해 그림자맵을 생성하기 위해 사용되는 평행 투영 기하 방법을 설명하고, 하드웨어에서 가속화되는 그림자맵 방법을 설명한 후, 이들을 이용해 구성된 전체 알고리즘을 설명한다.

3.1. 평행 투영 기하

본 논문에서는 구현을 간단히 하기 위해 평행광원(directional light source)을 가정하였다. 이 가정 하에서 필요한 물체를 투영면으로 투영하는 평행 투영 기하방법을 설명한다.

2 차원의 이미지(image)를 3 차원 상의 다른 곳에서 바라본 다른 이미지를 만들어 내는 변환을 호모그래피(homography) 변환이라 한다. 이때 어떤 이차원 이미지 A 가 3 차원 물체 집합 O 를 평행 투영해서 얻어진 결과라 하자. 이 A 이미지를 호모그래피 변환해서 이미지 B 를 얻고자 한다면, A 이미지 화소의 확대 축소로 인해 B 이미지에는 앨리어싱이 생기게 된다. 만약 B 이미지를 O 로부터 직접 얻어낸다면 이런 앨리어싱은 생기지 않는다. 이 과정이 그림자맵의 경우에도 유사하게 적용이 된다.

(그림 3-2)는 물체 투영 평면(geometry projection plane)에 평행 투영된 물체를 시점에서 보는 과정이다. 시점에서 보는 이미지를 물체로부터 직접 얻는 과정은 아래와 같다.

- 물체를 투영평면으로 투영한다.
  - 다각형의 각 정점을 투영평면으로 빛의 방향을 따라서 이동시킨다.

- 투영된 정점을 깊이 비교가 가능한 형태로 변환한다.
- 시점에서 Z-buffer 테스트를 하면서 투영된 물체를 렌더링 한다.

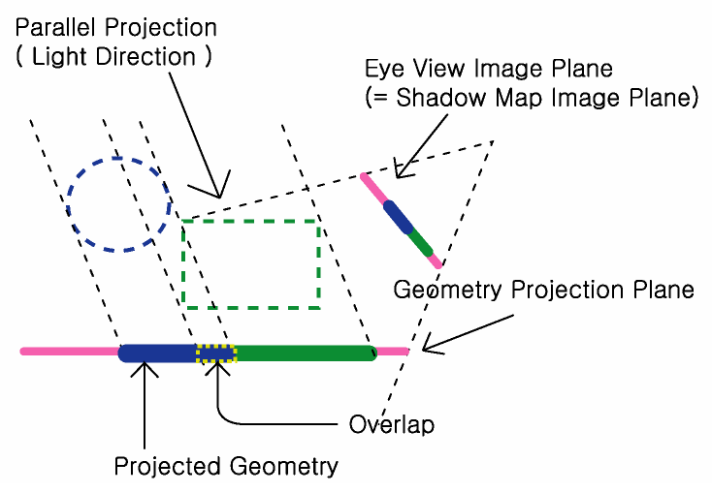


그림 3-2 평행 투영 기하

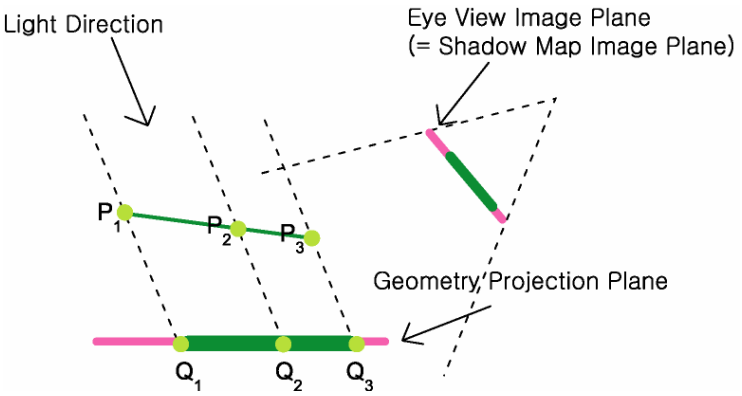


그림 3-3 투영된 물체의 깊이 보간

3.1.1. 좌표계

렌더링 파이프라인과 관련해서 본 논문 전체에 걸쳐 사용되는 좌표계 용어에 대해서 설명한다.

- 물체 좌표계(Object coordinate)

  - 물체의 입력 정점들이 정의된다. 정점을 P 라하자

- 세계 좌표계(World coordinate)

  - 화면에 보여주기 전단계의 좌표계이다. ($P * M_{World}$)

- 시점 좌표계(Viewing coordinate)

  - 시점 변환 후의 좌표계이다. ($P * M_{World} * M_{View}$)

- 클립 좌표계(Clip coordinate)

  - 투영 변환 후의 좌표계이다.

  $(x, y, z, w) = P * M_{World} * M_{View} * M_{Proj}$

- 정규 직교 좌표계(Normalized Device Coordinate)

  - w 로 나누어 준다. 시야 절두체(viewing frustum)의 x,y 영역은 [–1,1], z 영역은 [0,1]에 위치하게 된다.

  - $(X, Y, Z) = (x/w, y/w, z/w)$

- 화면 좌표계(Screen Coordinate)

  - 화면 위치로 평행이동, 확대 한다. $(X, Y, Z) * M_{Viewport}$

3.1.2. 평행 투영

물체 들을 빛의 방향을 따라서 물체 투영 평면으로 투영시킨다. 물체의 각

정점들이 세계 좌표계로 변환 된 후, 빛의 방향 D_L 으로 물체 투영 평면에 투영된 새로운 정점으로 변환된다.

물체 투영 평면을 식으로 나타내면

$$ax + by + cz + d = 0$$

(그림 3-3)에서 점 P,Q 를 세계 좌표계로 나타내면

$P_W = (x_W^P, y_W^P, z_W^Q)$ - 여기서 W 는 세계좌표계(world coordinate)를 뜻한다.

$Q_W = (x_W^Q, y_W^Q, z_W^Q)$

이 된다. 이때 세계 좌표계로 표현된 빛의 방향 벡터를 L_W 이라 하면, 점 P 와 Q 사이에는 아래의 식이 성립한다.

$$Q_W = P_W - L_W * \frac{P_W \cdot (a,b,c) + d}{L_W \cdot (a,b,c)}$$

이제 이 정점이 클립 좌표계(clip coordinate)에서 차지하는 위치는 아래와 같다.

$$Q_C = (x_C^Q, y_C^Q, z_C^Q, w_C^Q) = (x_W^Q, x_W^Q, x_W^Q, 1) * M_{EyeView} * M_{EyeProj}$$

3.1.3. 투영된 물체의 깊이 비교

위에서 구한 Q_C 는 물체 투영 평면상에 위치한다. 이로 인해 (그림 3-2)에 나타난 것처럼 시점에서 두 투영된 물체가 겹치는 부분의 경우 파란색이 선택되어야 하지만 이를 구별 할 수 없게 된다. 여기서는 가속기의 Z-buffer 기

능을 사용해 투영된 물체의 깊이를 비교하는 방법을 설명한다.

- 정점단위로 수행되는 알고리즘

$$P_W = P_O * M_{World}$$

$$P_L = (x_L^P, y_L^P, z_L^P, 1) = P_W * M_{LightView} * M_{LightProj}$$

$$Q_W = P_W - L_W * \frac{P_W \cdot (a,b,c) + d}{L_W \cdot (a,b,c)}$$

$$Q_C = (x_C^Q, y_C^Q, z_C^Q, w_C^Q) = Q_W * M_{EyeView} * M_{EyeProj}$$

Output $(x_C^Q, y_C^Q, \underline{\underline{z_L^P}}, w_C^Q)$

최종 결과에서 클립 좌표계로 계산된 결과의 z 값을 z_C^Q 대신 z_L^P 로 바꾸어 준다. 이 경우 최종 z-buffer 에는 z_C^Q / w_C^Q 대신 z_L^P / w_C^Q 가 저장되게 된다. 이때 화면상에서 같은 위치를 차지하는 정점끼리는 w_C^Q 가 동일하고 항상 $w_C^Q > 0$ 므로, 이 식은 빛에서 본 순서를 나타내는 데 적합하다. 이제 삼각형의 꼭지점이 아닌곳, 즉 interpolated depth 를 사용하는 곳에서 정확한지를 알아보자.

- 증명

$$P_2 = \alpha P_1 + (1-\alpha) P_3$$

$$Q_2 = \alpha Q_1 + (1-\alpha)Q_3 \quad \left(\because \overline{P_1Q_1} // \overline{P_2Q_2} // \overline{P_3Q_3}\right) \longrightarrow$$

이라 하자. 이때

$$P_L = (x_L^P, y_L^P, z_L^P, 1) = P_W * M_{LightView} * M_{LightProj}$$

$$Q_V = (x_V, y_V, z_V, 1) = Q_W * M_{EyeView}$$

$$Q_C = (x_C, y_C, z_C, w_C) = Q_V * M_{EyeProj}$$

라 하자. 이때 결과 좌표

$$O_C = (x_C^Q, y_C^Q, z_L^P, w_C^Q) \ ,$$

$$O_N = (X_N, Y_N, Z_N) = \left(\frac{x_C^O}{w_C}, \frac{y_C^O}{w_C}, \frac{z_C^O}{w_C}\right) = \left(\frac{x_C}{w_C}, \frac{y_C}{w_C}, \frac{z_L^P}{w_C}\right)$$

에 대하여, Q_2 위치에서의 보간(interpolate)된 Z_N^{O2} 은 아래와 같다.

$$\beta Z_N^{O1} + (1-\beta)Z_N^{O3} \longrightarrow$$

이제 α 와 β 사이의 관계를 구한다.

임의의 점 A^1, A^2, A^3 에 대해

$A_W^2 = \alpha A_W^1 + (1-\alpha)A_W^3$ 라 하자.

$\rightarrow \ A_C^2 = \alpha A_C^1 + (1-\alpha)A_C^3 \quad \left(\because A_C = A_W M_{View\,Proj}\right)$

양변을 w_C^{A2} 으로 나누면

$$A_N^2 = \frac{\alpha A_N^1 w_C^{A1} + (1-\alpha) A_N^3 w_C^{A3}}{\alpha w_C^{A1} + (1-\alpha) w_C^{A3}} \qquad \left(\because w_C^{A2} = \alpha w_C^{A1} + (1-\alpha) w_C^{A3}\right)$$

$$\therefore \beta = \frac{\alpha w_C^{A1}}{\alpha w_C^{A1} + (1-\alpha) w_C^{A3}} \quad \longrightarrow$$

, 에서

$$\beta Z_N^{O1} + (1-\beta) Z_N^{O3} = \frac{\alpha Z_N^{O1} w_C^{O1} + (1-\alpha) Z_N^{O3} w_C^{O3}}{\alpha w_C^{O1} + (1-\alpha) w_C^{O3}}$$

$$= \frac{\alpha z_C^{O1} + (1-\alpha) z_C^{O3}}{\alpha w_C^{O1} + (1-\alpha) w_C^{O3}} \qquad \left(\because Z_N w_C = Z_C\right)$$

$$= \frac{\alpha z_L^{P1} + (1-\alpha) z_L^{P3}}{\alpha w_C^{Q1} + (1-\alpha) w_C^{Q3}} = \frac{z_L^{P2}}{w_C^{Q2}} \qquad (\because \quad)$$

$$= Z_N^{O2}$$

따라서 이 방법은 삼각형의 정점 외에 보간 깊이가 사용된 곳에서도 깊이 비교가 동작한다.

3.2. 하드웨어 가속화된 그림자맵

그림자맵에는 광원시점에서 본 깊이 값이 저장되어 있다. 이때 화면에 그려질 각 화소마다 광원시점 깊이와 그림자맵에 저장된 깊이를 비교해서 그림자 여부를 판단할 수 있다. (그림 3-4) 다시 말해서 (P_x, P_y, P_z) 는 광원 시점 공간으로 변환된 화소의 위치라 하면, 깊이 비교는 아래의 식과 같이 표현 할 수 있다.

$$p_z \le shadow\_map(p_x, p_y)$$

최근의 가속기에서는 그림자맵 방법이 깊이 텍스쳐의 텍스쳐 맵핑 형태로 직접 지원이 된다. 광원 시점에서 렌더링을 해서 생긴 Z-buffer 가 그림자맵이고, 이 그림자맵이 texture 로 사용이 된 상태에서, 화소마다의 텍스쳐 좌표가 광원시점에서의 클립 좌표계 점 (P_s, P_t, P_r, P_q) 가 되도록 하면 투영 텍스쳐 맵핑과 유사한 방식으로 그림자 여부가 판단된다. 이 과정은

$$\frac{p_r}{p_q} \le texture\_2D\left(\frac{p_s}{p_q}, \frac{p_t}{p_q}\right)$$

으로 나타낼 수 있다. [4]

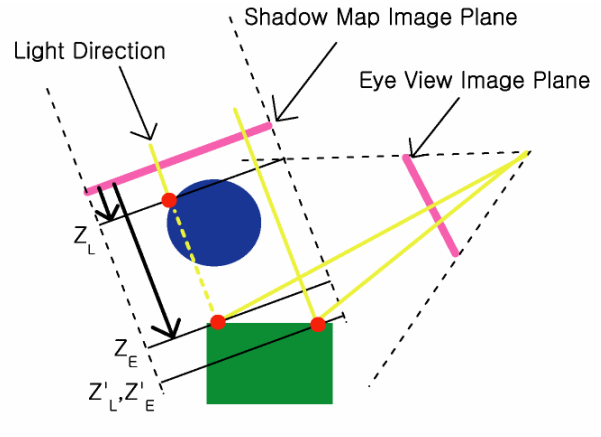


그림 3-4 그림자맵 방법

3.3. 평행 투영 기하를 이용한 시점의존적 그림자맵 방법

위에서 설명한 깊이 비교가 가능한 평행 투영 기하방법은 그림자맵 알고리즘을 위해 사용될 수 있다. 이는 평행 투영 기하 방법으로 모든 정점을

물체 투영 평면으로 투영 후 시점에서 깊이 비교가 가능하도록 렌더링하고 이때 생성된 깊이 버퍼(Z-buffer)를 그림자맵으로 사용하는 것이다. 이때 물체 투영 평면위치에 위치한 물체가 그림자를 받게 된다면, 그림자맵 격자가 화면격자와 평행하게 되어 그림자맵 화소와 화면 화소의 크기 비율이 일정하다. 반대로 그림자맵 화소와 화면 화소의 크기 비율은 그림자를 실제로 받은 위치와 물체 투영 평면이 멀어질수록 편차가 커지게 된다. 이러한 특징은 장면에 따라서 좋은 결과를 얻을 수 있는 물체 투영평면을 결정할 수 있다는 점에서 의미가 있다.

예를 들어 대부분의 물체들이 물체 투영 평면 근처에 넓게 퍼져서 분포하는 경우에 그림자맵이 가장 효율적으로 저장될 것이다. 그리고 일반적으로 물체 투영 평면은 그림자가 주로 보이게 될 평면과 같게 설정하는 것이 좋다. 이런 의미에서 물체 투영 평면을 화면상의 그림자를 대표하는 위치에 있어야 하므로 대표 그림자 평면(shadow plane)이라고 부르기로 한다.

이때 시점 의존적 그림자맵이 생성되는 과정은 (그림 3-5)에 나타나 있다.

이렇게 생성된 그림자맵을 사용하여 장면을 렌더링 하는 과정은 생성된 그림자맵을 깊이 텍스쳐로 사용한 상태에서 텍스쳐 좌표를 그림자맵을 생성할 때와 동일한 방법으로 설정하여 그리는 것이다. 즉 그려질 화소들의 화면위치와 그림자맵 공간 위치를 각각 윈도우 좌표와 텍스쳐 좌표로 정하면 앞에서 설명한 하드웨어 가속화된 그림자맵 방법이 사용되어 화면에 그림자가 있는 장면이 나오게 된다. (그림 3-6)

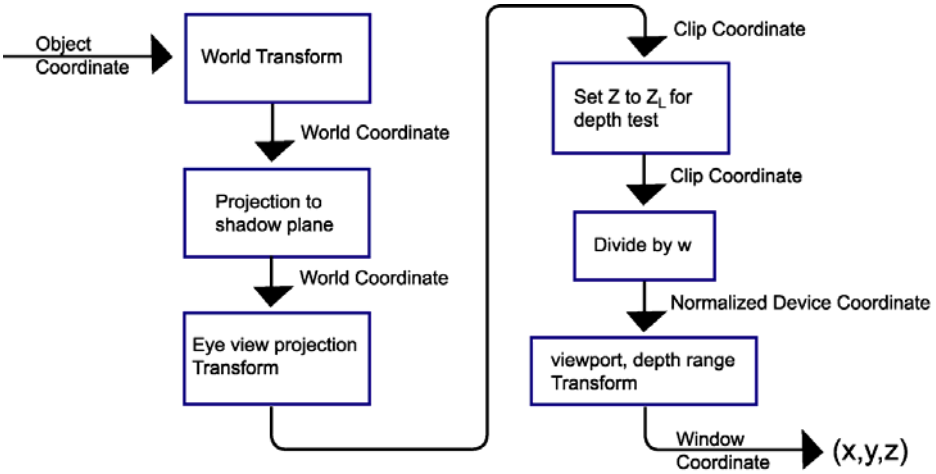


그림 3-5 그림자맵 생성 파이프라인(pipeline)

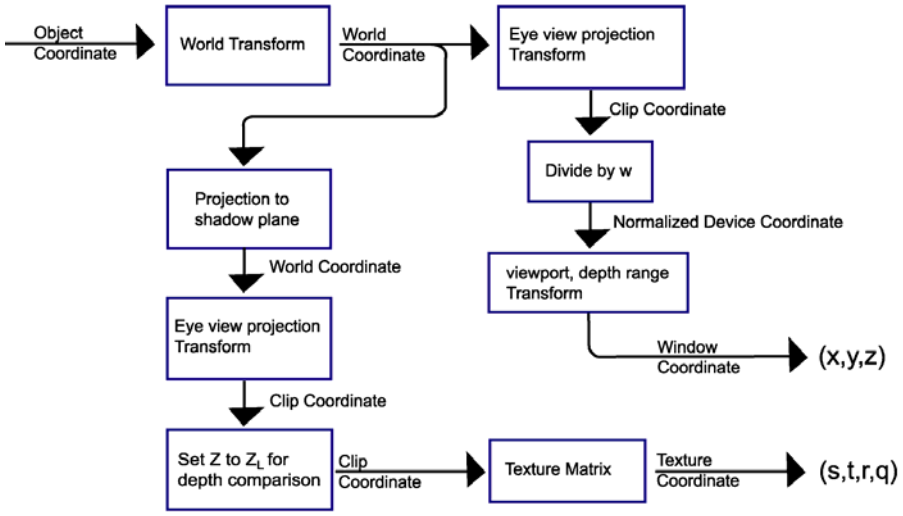


그림 3-6 렌더링 파이프라인

4. 구현

4.1. 그림자맵 영역 조정

그림자맵이 저장되는 영역이 작으면, 상대적으로 화면에서 그림자맵 한 화소가 차지하는 영역이 좁아져서 좋은 결과를 얻게 된다. 따라서 그림자맵 중 실제 화면에서 사용될 부분만을 확대해서 텍스쳐에 저장하도록 하면 앨리어싱이 최소화 될 수 있다. 이때 그림자맵 영역은 화면에서 보이는 모든 물체를 포함하는 범위에서 좁아 질 수 있다. 본 논문에서는 (그림 4-1)에서 나타난 것처럼 시야 절단체(view frustum) 전체를 저장하는 대신, 그림자맵 영역(shadow map region)부분만을 확대해서 저장하는 방법을 사용했다. 그 결과는 (그림 4-2)에 나타나 있다. 시점이 그림자 평면에 다가갈 때 시점 의존적 그림자맵이 변하는 모습을 순서대로 그렸다. (그림 4-2-a) 에 비해서 (그림 4-2-b) 가 더 확대되어 저장되는 것을 알 수 있다.

이 방법은 평행 투영이나 원근 투영에 상관 없이 사용 가능하다. 기존 그림자맵 방법에도 적용이 가능하다.

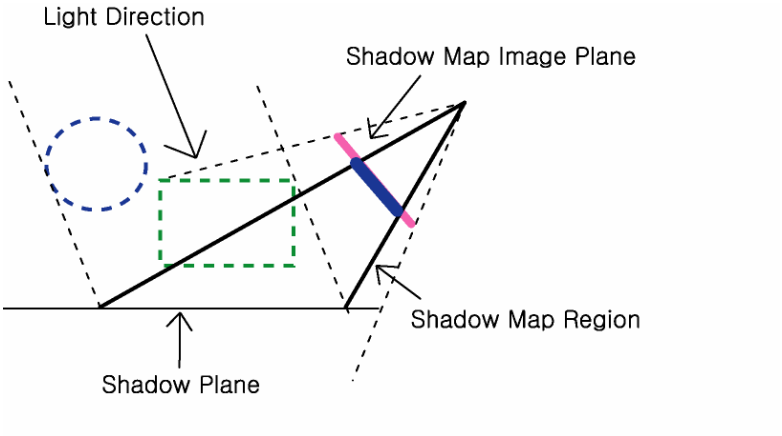
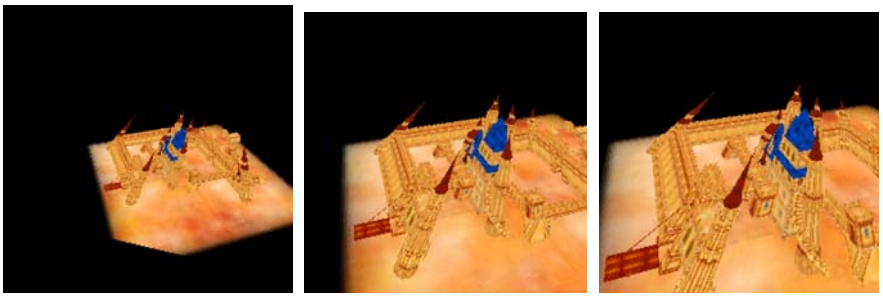
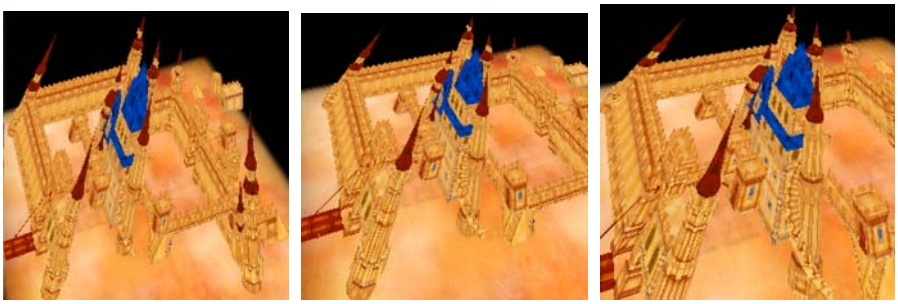


그림 4-1 그림자맵 영역



(a) 그림자맵 영역 조정을 사용하지 않은 경우



(b) 그림자맵 영역 조정을 사용한 경우

그림 4-2 사용 예

- 알고리즘

화면 영역을 바꾸어 주는 것은 정규 직교 좌표계에서의 평행이동(translate)과 확대축소변환(scale)으로 나타낼 수 있다. 하지만 클립 좌표계에서 정규 직교 좌표계로 바꾸어 주는 것은 하드웨어에서 행해 지므로 클립 좌표계에서 표현된 좌표를 미리 바꿔줌으로써 같은 효과를 얻을 수 있다. 이는 먼저 물체들의 경계상자를 이용해서 대표 그림자 평면으로 투영된 물체들이 화면에서 차지하는 영역을 구하고, 모든 정점단위로 이 영역이 화면에 전체를 차지하도록 평행이동, 확대축소변환 해주는 방식으로 동작한다.

정규 직교 좌표계에서 그림자맵 영역 $[(\min X, \min Y), (\max X, \max Y)]$를 구하면 그림자 맵은 위 영역만 저장되면 된다. 이 영역만이 화면에 나오도록 하는 변환은 아래를 만족해야 한다.

$$(\min X, \min Y) * Translate(t_x, t_y) * Scale(s_x, s_y) \rightarrow (-1,-1)$$

$$(\max X, \max Y) * Translate(t_x, t_y) * Scale(s_x, s_y) \rightarrow (1,1)$$

이를 만족하는 $(t_x, t_y), (s_x, s_y)$는 아래와 같다.

$$(t_x, t_y) = (-(\min X + \max X)/2, -(\min Y + \max Y)/2)$$

$$(s_x, s_y) = \left(\frac{2}{\max X - \min X}, \frac{2}{\max Y - \min Y} \right)$$

실제로 이는 모든 정점에서 변환이 끝난 클립 좌표계 좌표 (x_c, y_c, z_c, w_c) 를 평행이동과 확대축소 변환을 하는 것이 된다. 이 과정은 아래와 같다.

$$(x_c, y_c, z_c, w_c) = (x_w, y_w, z_w, 1) * M_{LightView} * M_{LightProj}$$

$$(x_o, y_o, z_o, w_o) = ((x_c + t_x w_c) * s_x, (y_c + t_y w_c) * s_x, z_c, w_c)$$

즉 그림자맵 생성시 정점의 최종 결과를 (x_o, y_o, z_o, w_o)로 설정하면 된다. 렌더링 시의 텍스쳐 좌표도 같은 방식으로 한다. 위 결과가 정규 직교 좌표계로 변환되면 w 로 나누어져서 $((X + t_x) * s_x, (Y + t_y) * s_y, Z)$ 가 된다.

4.2. 깊이 바이어스(Z-bias) 사용하기

그림자맵 알고리즘은 샘플링위치가 달라서 생기는 오차로 인해 자기 자신의 그림자를 받아 지저분해 지는 경우가 있다. 이를 개선 하기 위해 다각형 구분번호(polygon ID)를 사용하는 방법과 전면 제거(front face culling)를 이용하는 방법과 깊이 바이어스(Z-bias)를 이용하는 방법이 있다. 다각형 구분 번호 방법은 그림자맵 생성시 다각형 구분번호를 부여하여 같은 구분번호의 다각형에 의해 생긴 그림자는 받지 않도록 하는 법이다. 전면 제거 방법은 그림자맵 생성때는 후면을 사용하고 렌더링때는 전면을 사용하여 생기는 깊이 차이로 오차를 가리는 방법이다. 깊이 바이어스 방법은 그림자 비교시 깊이에 비례하는 오차값(Z-bias)를 빼주어서 그림자맵에 저장된 깊이와 어느 정도 이상 비슷해야지만 그림자로 판단하게 하는 방법이다. 어떤 방법이든

사용될 수 있지만, 본 논문에서는 깊이 바이어스를 사용했다. 이는 텍스쳐 행렬을 바꾸어 줌으로써 가능하다.[4]

4.3. 버텍스 셰이더(Vertex Shaders)

본 논문에서는 정점을 직접 변환하는 과정이 많이 사용된다. 이 과정은 CPU 를 사용하여 처리할 수도 있지만, 최근의 그래픽 하드웨어에서는 프로그래밍 가능한(programmable) 정점처리를 지원하므로 이를 사용하면 보다 빠른 속도로 렌더링이 가능하다. 본 논문에서는 DirectX 에서 지원하는 버텍스 셰이더를 사용하여 이 과정을 처리했다.[15] [16] 그림자맵을 생성하는 과정과 그림자를 렌더링 하는 과정에 각각 다른 버텍스 셰이더를 사용하여 정점을 변환한다. 각 버텍스 셰이더는 좌표계 변환 행렬들과 그림자맵 영역 조절을 위한 벡터들, 빛의 방향, 대표그림자 평면 방정식 등을 변수로 받아서 입력정점을 출력정점으로 변환하게 된다. 그림자맵 생성시의 출력정점 좌표 결정 부분과 그림자 렌더링 시의 텍스쳐 좌표 생성부분은 거의 동일하고 단지 마지막에 정점 좌표를 텍스쳐 좌표로 바꾸어 주기 위해서 텍스쳐 행렬이 곱해지는 과정이 추가된다.[4]

4.4. 렌더링

그림자 텍스쳐 맵핑 결과로 그림자 여부에 따라 1 또는 0 이 계산된다. 이를 shadow 라고 할 때 출력 화소의 결과는 아래와 같다.

$$\left(\text{Ambient term} + \left(\text{diffuse term} + \text{specular term}\right) * \text{shadow}\right) * \text{color texture}$$

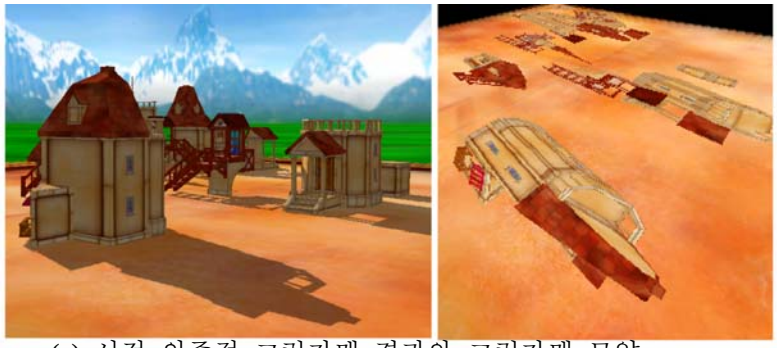
이는 최근의 하드웨어에서 픽셀 셰이더(pixel-shader)나 다중 텍스쳐 방법

(multi-texturing)을 써서 한 패스로 구현 가능하다. 또한 다중 렌더링(multi-pass rendering)이나 다중 텍스쳐 방법으로 여러 개의 여러 개의 광원으로 확장 될 수 있다.

5. 결과

모든 결과는 물체들의 평균정도 높이에 바닥면과 평행한 대표 그림자 평면을 설정하고 얻어진 것이다. 결과를 공정하게 하기 위하여 기존 방법도 그림자맵 영역을 가장 좁게 설정하였다. 펜티엄 III 800Mhz 기계에서 nVidia GeForce3 Ti 200 비디오 카드를 사용하여 렌더링 한 결과이다. 그림자맵은 256*256 을 사용했고 매 프레임 다시 생성되었다.

(그림 5-1)은 그림자맵 생성시 8069 개의 삼각형이 렌더링 되었고 모든 과정을 합쳐서 도합 16554 개의 삼각형이 프레임(frame)마다 렌더링 되었다. 결과를 보면 시점 의존적 그림자맵의 경우 그림자맵의 모양과 실제 바닥면에 생긴 그림자의 모양이 일치함을 알 수 있다. (그림 5-1-a-확대)에서 왼쪽은 시점에서 가까운 부분이고 오른쪽은 시점에서 먼 부분이다. 각각의 그림자맵 픽셀 크기가 비슷함을 알 수 있다. 반면 (그림 5-1-b-확대)에 나타난 그림을 보면 가까운 부분은 그림자맵의 확대로 인한 격자가 보이고, 멀리 있는 부분은 그림자맵이 축소되어 앤티앨리어싱이 동작하지 않는 것을 볼 수 있다.



(a) 시점 의존적 그림자맵 결과와 그림자맵 모양

(b) 기존 그림자맵 결과와 그림자맵 모양

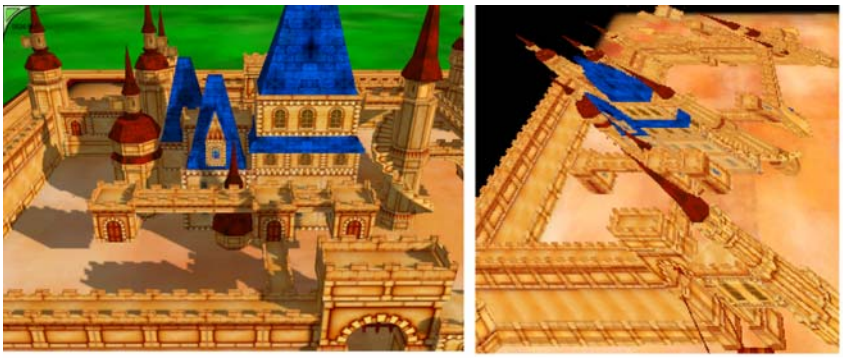
(a-확대) (b-확대)

그림 5-1 실험 1

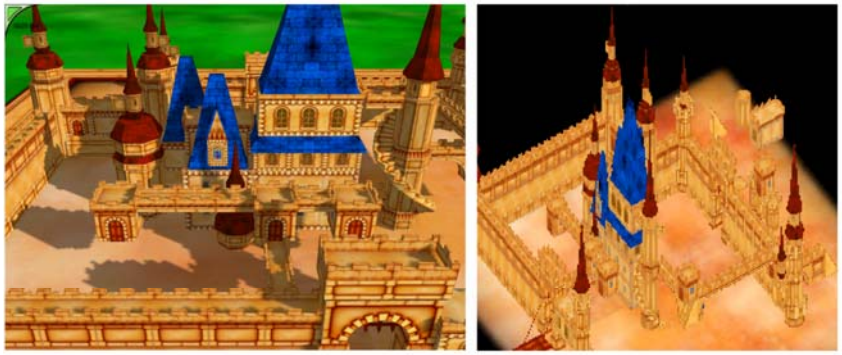
- 수행 속도

	프레임 율 (Frame rate)	그림자맵 생성 시간	그림자를 그리는 물체의 렌더링 시간
기존 방법	45FPS	6.5 ms	7.3 ms
시점 의존적 그림자맵	46FPS	6.2 ms	7.5 ms

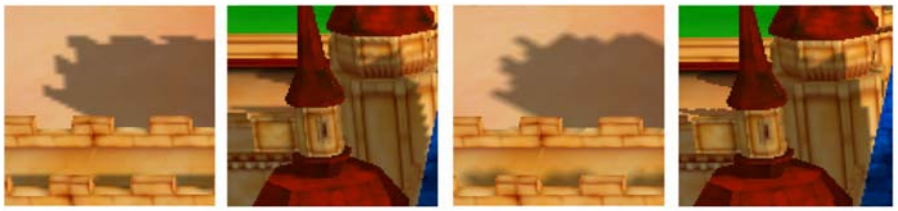
(그림 5-2)는 그림자맵 생성시 16257 개의 삼각형이 렌더링 되었고 모든 과정을 합쳐서 도합 31312 개의 삼각형이 매 프레임 렌더링 되었다. 위의 결과를 보면 시점 의존적 그림자맵에서는 투영된 물체의 모양이 바닥면에 생길 그림자의 모양과 거의 일치함을 알 수 있다. 이런 이유로 그림자맵 화면 상에서 빈 공간이 색이 많아 비효율적인 기존 방법에 비해 효과적임을 알 수 있다.



(a) 시점 의존적 그림자맵 결과와 그림자맵 모양



(b) 기존 그림자맵 결과와 그림자맵 모양



(a-확대) (b-확대)

그림 5-2 실험 2

- 수행속도

	프레임 율 (Frame rate)	그림자맵 생성 시간	렌더링 시간
기존 방법	25FPS	12.3 ms	11.4 ms
시점 의존적 그림자맵	25FPS	11.3 ms	10.6 ms

6. 결론

본 논문에서 우리는 대표 그림자 평면으로 물체를 투영하는 방법을 이용한 그림자맵 방법을 제안했다. 이 방법은 그림자맵 화소의 크기와 화면 화소의 크기의 비율을 일정하게 하기 위하여 제안 되었다. 그래픽 가속 하드웨어의 제한 범위 안에서 사용 가능하도록 하는데 주력하였고 그 결과 대표 그림자 평면으로 대표될 수 있는 넓고 복잡한 장면에서 속도의 손실 없이 좋은 결과를 얻었다.

본 논문에서는 그림자맵 카메라를 시점 카메라와 동일하게 설정하고 대표 그림자맵 평면을 조절하여 최적의 결과를 얻으려고 시도했다. 그 결과 본 논문의 가장 큰 문제는 대표 그림자 평면을 다양한 장면에서 적절하게 잡기가 어렵다는 것이었다. 이 단점을 개선하기 위해서 그림자맵 카메라를 시점 카메라와 분리하고, 그림자맵이 사용될 위치에 따라서 그림자맵 평면과 그림자맵 카메라를 여러 개 설정하는 방식을 생각중이다. 또 다른 개선점이라면 현재 그림자맵이 매 프레임마다 렌더링이 되는데, 이전에 생성된 그림자맵의 샘플링 격자가 화면 격자와 비교해서 크게 왜곡되었을때만 렌더링을 하도록 개선 할 수 있을 것이다.

7. 관련연구

[1] Mark Segal, et al. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH '92*, pages 249-252, 1992.

[2] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78*, pages 270-274, 1978.

[3] W. Reeves, D. Salesin, R. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of SIGGRAPH '87*, pages 283-291.

[4] Cass Everitt, Ashu Rege, Cem Cebenoyan. Hardware shadow mapping. www.nvidia.com

[5] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. *SIGGRAPH 2001 Conference Proceedings*, ACM SIGGRAPH, Addison Wesley, August 2001.

[6] Kyoung-Su Oh, Byeong-Seok Shin, Yeong Gil Shin. Linear Time Shadow Texture Generation Algorithm , In *Proceedings of GTEC 2001*, January 17-20, 2001

[7] A.Woo, P.Poulin, and A.Fournier. A Survey of shadow algorithms, In *IEEE Computer Graphics and Application*, 10(6):13-32, Nov. 1990

[8] Crow, Franklin C. Shadow Algorithms for Computer Graphics, Computer Graphics (SIGGRAPH '77), vol.11, no.2, pages 242-248, July 1977.

[9] T.Heidmann. Real shadows, real time, Iris Universe, 18:28-31, 1991, Silicon Graphics, Inc., 1991.

[10] J.F.Blinn. Me and my (fake) shadow, IEEE Computer Graphics and Applications, 8(1):82-86, Jan.1988.

[11] 오경수, 신병석, 신영길. 물체들간의 그림자를 표현하는 텍스쳐의 효율적인 생성방법, 한국컴퓨터그래픽스학회 논문지 2000 년 12 월호(제 6 권 제 4 호)

[12] Tom Lokovic, Eric Veach. Deep Shadow Maps. *SIGGRAPH 2000 Proceedings* , ACM SIGGRAPH, Addison Wesley, August 2000.

[13] Wolfgang Heidrich. High-quality Shading and Lighting for Hardware-accelerated Rendering. University of Erlangen, Computer Graphics Group, 1999

[14] 오경수. 깊이버퍼를 이용하는 효율적인 가시성 검사와 그림자 생성 방법. 서울대학교 컴퓨터 공학부 박사학위 졸업논문, 2001 년 3 월

[15] Microsoft. *DirectX 8.0 Programmer's Reference*, Oct. 2000.

[16] Chris Maughan, Matthias Wloka. Vertex Shader Introduction, NVIDIA Corporation.

Abstract

In real-time rendering, shadow maps are one of commonly used methods for shadow effects. Although shadow maps have many advantages especially in dynamic scenes, they have significant drawback that they produce aliasing artifacts. Aliasing artifacts occur because the size of a shadow map pixel mismatches the size of corresponding screen pixel. This paper introduces the view-dependent shadow map to reduce this problem. In this algorithm, objects are projected to shadow plane along direction of light so that shadow map can be made in eye space. As a result, the shadow map is view-dependent and has non-uniform sampling grids at the light source view. Also it can be accelerated by recent graphic hardwares that supports traditional shadow map algorithm. These features allow us to produce better quality pictures without loss of efficiency.

Keywords : real-time rendering, projective geometry, shadows, anti-aliasing

Sttudent number : 2000-20715