韓國컴퓨터情報學會 論文誌
第14卷 第2號, 2009. 2.

2009-14-2-2-1

모바일 3D 그래픽스를 위한 저전력 텍스쳐 맵핑 기법

김 현 희*, 김 지 홍**

A Low-Power Texture Mapping Technique for Mobile 3D Graphics

Hyunhee Kim*, Jihong Kim**

요 약

3차원 그래픽스에서 영상의 현실감을 높이기 위해 자주 사용되는 텍스쳐 맵핑 기법은 많은 연산량과 메모리 접근의 요구로 성능과 전력상의 병목점이 되고 있으며, 이러한 텍스쳐 맵핑 단계에서의 메모리 접근시간을 줄이기 위해 텍스쳐 캐시가 이용되고 있다. 그러나 점차 소형화 되고 있는 휴대용 기기의 특성과 배터리로 동작하기에 갖는 전력상의 제약으로 인해 텍스쳐 캐시가 차지하는 면적과 에너지 소모를 줄이는 노력이 필요하다. 본 논문에서 제안하는 기법은 텍스쳐 캐시의 크기가 줄어듦에 따라 발생하는 미스 율의 증가를 보완하기 위해 미리 읽기 기법을 사용한다. 또한 미리 읽기 버퍼에 텍스쳐 캐시에서 교체되는 블록을 임시로 저장해 둠으로써 충돌 미스를 줄이는 기법을 제안한다. 실험 결과, 1K bytes와 2K bytes의 캐시의 사용하면서 16K bytes 또는 8K bytes의 캐시를 사용했을 때와 비슷한 성능을 유지할 수 있음을 확인할 수 있었다. 또한 제안하는 기법의 사용으로 텍스쳐 캐시에서 소모되는 에너지 소모를 23%~60%까지 줄이고 70%정도의 면적을 감소 시킬 수 있음을 보여주었다.

Abstract

ETexture mapping is a technique used for adding reality to an image in 3D graphics. However, this technique becomes the bottleneck of the 3D graphics pipeline because it requires large processing power and high memory bandwidth. For reducing memory latency in texture mapping, texture cache is used. As portable devices become smaller and they have power constraint, it is important to reduce the area and the power consumption of the texture cache. In this paper, we propose using a small texture cache to reduce the area and the power consumption of the texture cache. Furthermore, we propose techniques to keep a performance comparable to large texture caches by using prefetch techniques and a victim cache. Simulation results show the proposed small texture cache can reduce the area and the power consumption up to 70% and 60%, respectively, by using 1~2K bytes texture cache compared to the conventional 16K bytes cache while keeping the performance.

▸ Keyword : (low-power),3차원 그래픽스(3D graphics), 텍스쳐 캐시(texture cache)

• 제1저자 : 김현희 교신저자 : 김지홍
• 투고일 : 2008. 11. 27, 심사일 : 2008. 12. 4, 게재확정일 : 2009. 2. 4.
* 서울대학교 전기,컴퓨터공학부 대학원 ** 서울대학교 전기,컴퓨터공학부교수
※ 이 논문은 2008년도 두뇌한국21 사업에 의해 지원되었으며, 2008년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구(No. R0A-2007-000-20116-0)입니다. 이 연구를 위해 연구 장비를 지원하고 공간을 제공한 서울대학교 컴퓨터연구소에 감사드립니다.

Ⅰ. 서 론

최근 휴대용 기기의 기능이 다양화되면서 휴대용 기기에서의 3차원 그래픽스 애플리케이션 사용이 보편화 되고 있다. 특히 이들 기기에서 3차원 게임이 가능해지면서, 휴대용 기기에서의 그래픽 하드웨어에 대한 관심이 증가하고 있다. 그래픽 처리 과정은 많은 양의 계산과 메모리 접근을 요구하는 작업이다. 그러나 배터리를 사용하는 이들 기기들이 가지고 있는 전력상의 제약 때문에, 휴대용 기기를 위한 그래픽 하드웨어에서의 저전력은 점점 중요한 설계 이슈가 되고 있다.

3차원 그래픽스는 화면에 보다 현실감 있는 이미지를 표현하기 위해 사용된다. 이러한 현실감 있는 영상이나 매우 복잡한 영상은 많은 수의 다각형 또는 삼각형으로 구성되며, 이로 인해 기하학적 연산 량이 크게 증가하게 된다. 이는 그래픽 시스템에서의 처리 속도를 저하시키며 에너지 소모도 크게 증가시키는 요인이 될 수 있다. 따라서 적은 기하학적 연산으로 더욱 현실감 있는 이미지를 표현하기 위해 텍스쳐 맵핑(texture mapping) 기법이 널리 사용되고 있다.

텍스쳐 맵핑은 2차원 상의 텍스쳐 이미지를 3차원 공간의 객체에 적용시키는 방법이다. 예를 들어 벽에 벽돌 이미지의 텍스쳐를 적용시키거나 나무 재질의 느낌을 표현하는 이미지를 바닥에 적용시킴으로써 적은 수의 다각형 또는 삼각형으로도 현실적이고 세밀한 영상을 표현할 수 있다. 그러나 화면의 픽셀(pixel)과 텍스쳐 이미지 상의 텍셀 (texel)이 항상 일대일 대응하는 것이 아니므로 삼각선형 보간 (trilinear interpolation) 또는 쌍선형 보간 (bilinear interpolation)의 필터링(filtering)을 수행하여 화면상의 픽셀에 대응하는 색상 값을 생성한다. 삼각선형 보간의 경우 8개의 텍셀을 필요로 하며 쌍 선형 보간의 경우는 4개의 텍셀을 필요로 하기 때문에 4~8의 텍셀을 얻기 위해 4~8번의 텍스쳐 메모리 접근이 요구된다. 이와 같은 연산 량과 메모리 접근 요구는 성능과 전력상의 병목 점이 된다.

텍스쳐 캐시의 사용은 텍스쳐 메모리의 접근을 줄임으로써 텍스쳐 맵핑 단계의 성능을 높일 수 있도록 한다. 지금까지의 텍스쳐 캐시에 관한 연구들은 대부분 메모리 대역폭과 텍스쳐 캐시의 성능 향상을 위한 연구들이 대부분이었다. 이러한 논문들에서는 16K bytes 정도의 큰 텍스쳐 캐시(texture cache)를 사용하여 1.5%~3%정도의 미스 율을 보이고 있다. 그러나 휴대용 기기에서의 면적상의 제약과 저전력의 요구로 인해 텍스쳐 캐시의 크기를 줄이는 것은 의미가 있다. 그림 1은 eCacti 1.0[1]을 이용하여 계산한 캐시 크기에 따른 정적 에너지이다. 1K bytes의 작은 캐시에서 소모되는 에너지와 비교하여 16K bytes에서 약 6배 정도의 정적 에너지가 소모됨을 볼 수 있다. 초당 30 프레임을 화면에 그린다고 가정했을 때 한 프레임을 그릴 때 캐시에서 소모되는 정적 에너지의 크기는 캐시 크기에 따라 좌우되기 때문에 캐시의 크기가 클수록 에너지 소모가 크게 증가하게 된다.

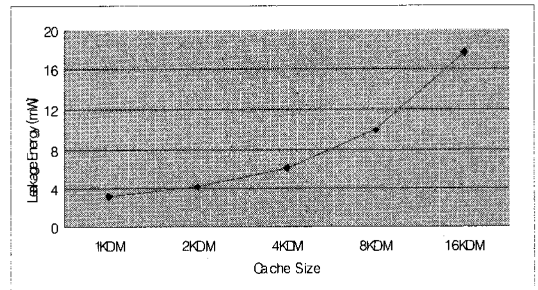


그림 1. 캐시 크기에 따른 정적 에너지
Fig.1 Static Energy Consumption of Various Cache Sizes

따라서 본 논문에서는 1K bytes 또는 2K bytes 정도의 작은 캐시의 사용을 제안하고, 캐시 크기를 줄임으로써 발생하는 미스 율의 증가를 보완하기 위해 미리 읽기(prefetch) 기법을 제안한다. 또한 미스 율을 줄이기 위한 텍스쳐 캐시 구조를 제안함으로써 캐시의 크기를 줄이면서도 큰 캐시를 사용하는 것과 같은 성능을 유지하도록 한다. 이러한 작은 캐시의 사용으로 텍스쳐 캐시에서 소모되는 에너지와 텍스쳐 캐시가 차지하는 면적을 줄일 수 있다.

본 논문은 다음과 같이 구성된다. 2장에서는 관련 연구를 소개하고, 3장에서는 3차원 그래픽스 파이프라인에 대해 간단히 설명을 한다. 4장에서는 제안하는 기법인 텍스쳐 맵핑이 갖는 지역성을 이용하여 복잡도와 정확도가 다른 세 가지 미리 읽기 기법을 소개한다. 5장에서는 제안하는 텍스쳐 캐시의 구조 및 동작 과정을 설명하며, 6장에서는 제시하는 각 미리 읽기 기법의 정확도와 작은 캐시와 미리 읽기 기법을 함께 사용했을 때의 미스 율 감소, 에너지 소모 감소 그리고 면적의 감소를 비교해 본다. 마지막으로 7장에서는 결론을 정리하고 앞으로 진행할 향후 작업에 대해 살펴본다.

Ⅱ. 관련 연구

텍스쳐 캐시의 성능에 관련된 여러 연구가 진행되어 왔다. 지금까지의 텍스쳐 캐시에 관한 연구들은 대부분 메모리 대역

폭과 텍스쳐 캐시의 성능 향상을 위한 연구들이 대부분 이었다. [2]의 논문에서 Hakura는 처음으로 텍스쳐 캐시의 사용을 제안하였으며 [3]의 논문에서는 메모리 접근 지연을 줄이기 위해 텍스쳐 캐시와 미리 읽기 기법을 결합하는 방법을 제시하고 있다. Cox는 [4]의 논문에서 L1 캐시와 함께 2-8 MB의 L2 캐시의 사용을 제안함으로써 프레임간의 텍스쳐 지역성까지 고려하고 있다. [5]의 논문에서는 텍스쳐 접근 방향에 따라 인덱스 비트를 바꾸는 방법을 제안하여 텍스쳐 캐시에서의 미스 율을 줄이고 있다. 이들 논문들에서는 16K bytes 정도의 비교적 큰 캐시를 사용하여 1.5%~3%정도의 미스 율을 보이고 있다. 그러나 앞에서 언급한 바와 같이, 휴대용 기기에서는 적은 전력 소모와 소형화가 요구되기 때문에 크기가 작은 캐시를 사용하면서 텍스쳐 캐시의 성능을 유지하는 방법이 요구된다.

[6]의 논문은 저전력 모바일 디바이스를 위한 프로그램 가능한 3차원 그래픽스 프로세서를 제안하였으며, [7]의 논문에서는 텍스쳐 캐시에서 소모되는 전력을 줄이기 위해 재 배열 가능한 텍스쳐 캐시 기법을 제안하였다. [8]의 논문에서는 휴대용 기기에서의 게이트 수의 제약과 전력소모를 고려하여 256~512 byte의 작은 텍스쳐 캐시의 사용을 제안하고 있다. 그러나 이 논문에서 주장하는 바와는 달리 텍스쳐 캐시의 크기를 줄였을 때 미스 율은 크게 증가하게 된다. 그림 2은 캐시 크기에 따른 미스 율을 보여준다. 실험에 사용된 애플리케이션은 Jelly Fish, Face model, Check Image이다. 캐시의 크기가 작아질수록 미스 율이 크게 증가함을 볼 수 있다. 특히 1~2K bytes정도의 캐시에서는 미스 율이 15%~20%까지 증가함을 보여준다.

본 논문에서는 1K bytes 또는 2K bytes 정도의 작은 캐시의 사용을 제안하고, 캐시 크기를 줄임으로써 발생하는 미스 율의 증가를 보완하기 위해 미리 읽기 기법을 제안한다. 또한 미스 율을 줄이기 위한 텍스쳐 캐시 구조를 제안함으로써 작은 캐시를 사용하면서도 큰 캐시를 사용하는 것과 같은 성능을 유지하도록 한다. 이러한 작은 캐시의 사용으로 텍스쳐 캐시에서 소모되는 에너지와 텍스쳐 캐시가 차지하는 면적을 줄일 수 있다.

3차원 그래픽스 파이프라인은 3차원 이미지를 화면상에 렌더링(rendering) 하는데 필요한 단계들로 크게 기하학적(geometry) 계산 부분과 래스터라이재이션(rasterization) 단계로 나눌 수 있다. 그림 3은 3차원 그래픽스에서의 파이프라인을 보여준다.

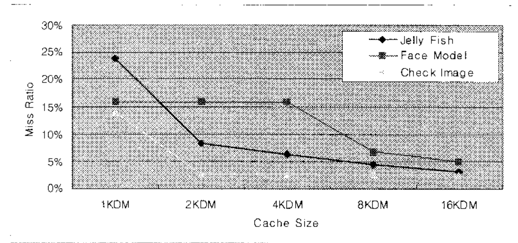


그림 2 .캐시 크기에 따른 미스율
Fig2. Cache Miss Ratio of Various Cache Sizes

III. 3차원 그래픽스 파이프라인

3.1 3차원 그래픽스 파이프라인

기하학적 계산 부분은 삼각형 또는 폴리곤으로 이루어지는 오브젝트들의 기하학적 데이터의 계산을 수행한다. 이 단계는 변환 (transformation) 단계, Trivial Rejection 단계, 라이팅(lighting) 단계, 뷰잉 변환 (viewing transformation) 단계로 이루어진다. 변환 단계에서는 평행 이동, 회전, 크기 변환 등을 통하여 오브젝트들을 오브젝트 좌표 계에서 월드 좌표 계(world coordinates)로 변환시킨다. 이러한 변환은 삼각형의 각 점에 4X4 변환 행렬을 곱함으로써 수행된다. Trivial Rejection단계에서는 뷰 박스의 밖에 위치하여 다음 단계에서 처리될 필요가 없는 오브젝트들을 제거한다. 라이팅 단계에서는 여러 라이팅 소스들에 대해 조도(illumination of intensity)를 계산하며 w 나누기 단계에서는 x, y, z 값을 동차 좌표 (homogeneous coordinate)의 w로 나눈다. 이는 원근 투영 (perspective prejection)후에는 w가 1로 남아있지 않기 때문이다. 마지막으로 뷰잉 변환 단계에서는 각 점에 4X4 행렬을 곱하여 월드 좌표로부터 정규화된 투영 좌표(normalized projection coordinates)로 변환시킨다.

래스터라이재이션 단계는 삼각형 셋업(triangle setup) 단계, 주사 변환 (scan conversion)단계, 텍스쳐 맵핑(texture mapping) 단계 그리고 프래그먼트(fragment) 연산 단계로 구성된다. 삼각형 셋업 단계에서는 기하학 처리 부분에서 입력된 삼각형 정보를 이용하여 여러 증분 값들(gradients values)을 계산한다. 주사 변환 단계에서는 삼각형 셋업 단계에서 계산한 증분 값들을 이용하여 스팬 라인(span line)상의 각 픽셀에 대응하는 값들, 즉 색상, 텍스쳐 좌표 등을 보간 한다. 텍스쳐 맵핑은 화면의 현실감을 증가시키기 위해 2차원 상의 이미지를 3차원 오브젝트에 적용시키

는 단계이다. 화면상의 각 픽셀에 대응하는 텍스쳐 이미지상의 값을 찾아 화면상의 픽셀 값을 대치 또는 조정한다. 마지막으로 프래그먼트 연산 단계에서는 α 블렌딩(blending), 안개, z 버퍼 테스트 등을 수행한다.

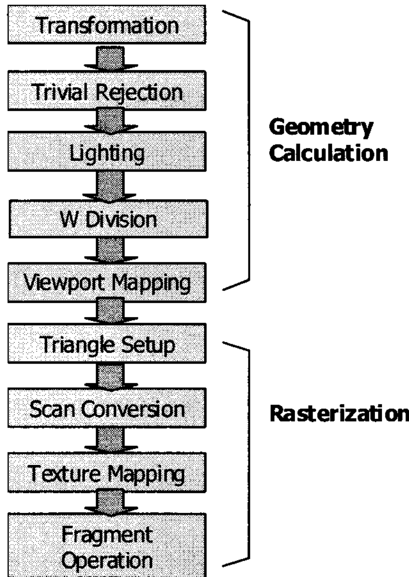


그림 3. 3차원 그래픽스 파이프라인 단계
Fig3. 3D Graphics Pipeline

3.2 텍스쳐 맵핑

텍스쳐 맵핑은 2차원 상의 이미지를 3차원 오브젝트에 적용시키는 방법으로 그래픽 처리에서의 복잡성을 감소시키면서 표면의 리얼리즘을 증가시키는 장점을 가지고 있다. 텍스쳐 맵핑 단계에서는 화면상의 각 픽셀에 대응하는 텍스쳐 이미지상의 값을 찾아 화면상의 픽셀값을 대치 또는 조정한다. 그림 4는 화면상의 픽셀과 텍스쳐 이미지 상의 텍셀의 관계를 보여준다. 그림과 같이 화면상의 스팬 라인 위의 하나의 픽셀은 텍스쳐 이미지 상의 텍셀에 대응된다.

그러나 화면상의 픽셀과 텍스쳐 이미지 상의 텍셀이 항상 일대일 대응하는 것은 아니므로 삼각선형보간(trilinear interpolation) 또는 쌍일차선형보간 (bilinear interpolation)의 필터링을 수행한다. 삼각선형보간의 경우 8개의 텍셀을 필요로 하며 쌍일차선형보간에서는 인접한 4개의 텍셀을 이용한다. 따라서 텍스쳐를 사용하는 한 픽셀의 값을 계산하기 위해서는 4 또는 8번의 메모리 접근을 요구하게 된다.

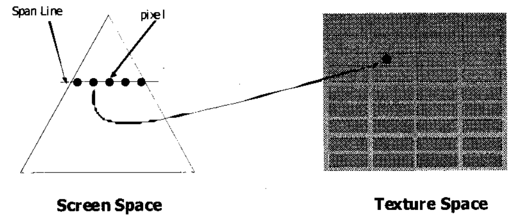


그림 4. 텍스쳐 맵핑 과정
Fig4. Texture Mapping

이러한 텍스쳐 맵핑에서의 메모리 접근 요구와 지연의 문제를 해결하기 위해 많은 그래픽스 하드웨어가 텍스쳐 메모리 이외에 텍스쳐 캐시를 사용하고 있다. 텍스쳐 캐시를 사용하는 텍스쳐 맵핑 단계의 구성은 그림 5와 같다.

주소 생성자(address generator)에서는 주사 변환 단계에서 계산된 텍스쳐 좌표인 u, v를 이용하여 텍셀의 주소를 계산한다. 텍스쳐 캐시에서 읽혀진 텍셀 값을 이용하여 필터링 단계에서는 쌍 일차 선형 보간 (bilinear interpolation) 또는 삼각 선형 보간 (trilinear interpolation)을 수행한다.

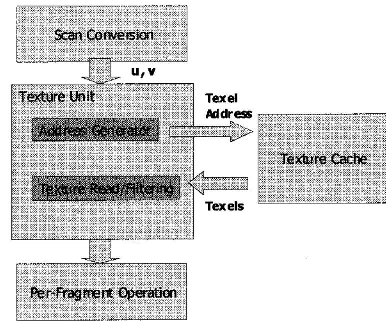


그림 5. 텍스쳐 맵핑 단계의 구성
Fig5. Texture Mapping Step

지역성과 시간적 지역성을 모두 갖고 있기 때문에 텍스쳐 캐시의 사용은 텍스쳐 맵핑의 성능을 크게 향상시킬 수 있다.

IV. 텍스쳐 맵 접근의 방향성을 이용한 미리 읽기 기법

본 절에서는 텍스쳐 캐시의 미스율을 줄이기 위한 세가지 미리 읽기기법을 소개한다. 각 미리 읽기 기법은 서로 다른 복잡도와 정확도를 갖는다.

4.1 변위 기반의 정확한 텍셀 예측 방법

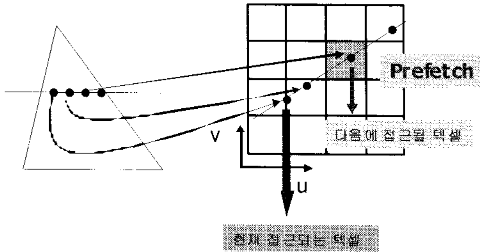


그림 6. 변위 기반의 정확한 텍셀 예측 방법
Fig6. Texel Prediction Method Based on Displacement

이 기법은 주사 변환 단계에서 보간(interpolation)시 이용하는 정보들을 이용하여 다음 접근될 텍셀을 정확하게 예측하는 방법이다. 이 기법에서는 텍스쳐 맵핑 단계에서의 텍셀 좌표를 계산하는 방법을 이용한다. 그림 6은 변위기반의 정확한 텍셀 예측 방법을 보여주는데, 그림과 같이 텍스쳐 이미지 상의 다음 접근될 블록을 미리 예측하여 미리 읽기를 하여 메모리 접근 지연 시간을 줄일 수 있다.

텍스쳐 맵핑 단계에서 텍스쳐 이미지상에서의 텍셀 좌표를 구하는 방법은 식 4.1과 같다. $\frac{\partial u'}{\partial x'}$와 $\frac{\partial v'}{\partial x'}$ 는 삼각형 셋업 단계에서 계산된다. 주사 변환 단계에서는 단지 현재 픽셀의 텍스쳐 좌표, u'(u/w), v'(v/w)에 이 변위 값을 더함으로써 다음 픽셀의 텍스쳐 좌표를 계산한다. 그러나 원근 보정을 위해 보간 된 w'(1/w)로 다시 나누어 주어야 한다.

$$u_{i+1} = \frac{u_i' + \frac{\partial u'}{\partial x'}}{w_i' + \frac{\partial w'}{\partial x'}}, \quad v_{i+1} = \frac{v_i' + \frac{\partial v'}{\partial x'}}{w_i' + \frac{\partial w'}{\partial x'}}$$ .... 식 4.1

이때 $\frac{\partial u'}{\partial x'}$, $\frac{\partial u'}{\partial x'}$, $\frac{\partial w'}{\partial x'}$에 적절한 값을 곱해줌으로써 다음 접근될 텍셀의 좌표를 미리 계산할 수 있다. 즉, ui+n과 vi+n의 좌표는 식 4.2와 같은 방법으로 계산될 수 있다.

$$u_{i+n} = \frac{u_i' + n * \frac{\partial u'}{\partial x'}}{w_i' + n * \frac{\partial w'}{\partial x'}}, \quad v_{i+n} = \frac{v_i' + n * \frac{\partial v'}{\partial x'}}{w_i' + n * \frac{\partial w'}{\partial x'}}$$ .... 식 4.2

본 논문에서는 블록 크기가 4X4임에 기반하여 예측된 텍셀이 현재 텍셀이 속한 블록과 다른 블록에 존재하도록 하기 위하여 n의 값을 4로 정하였다. 이 기법은 정확한 예측을 가능하게 하지만 나누기 연산을 수행하여야 하기 때문에 다음에 설명할 두 미리 읽기 기법에 비해 상대적으로 오버헤드가 크다.

4.2 변위 기반의 블록 예측 방법

4.1절에서 언급한 것처럼 정확한 텍셀 좌표를 예측하는 기법은 원근 보정으로 인해 나누기 연산이 요구되므로 오버헤드가 크다. 따라서 본 절에서는 텍스쳐 맵의 접근이 직선임을 이용하여 앞으로 사용될 텍스쳐 이미지의 블록을 예측함으로써 미리 읽기 주소 계산의 오버헤드를 줄일 수 있는 기법을 소개한다.

이 기법에서 제안하는 예측방법은 그림 7과 같이 다음 사용될 텍셀을 정확하게 예측할 수는 없지만 직선이 지나가는 블록들을 예측함으로써 미리 읽기 하는 기법이다. 이 기법에서는 주사 변환 단계에서 원근 보정이 수행됨을 가정한다. 따라서 원근 보정 이후의 텍스쳐 좌표인 u, v를 이용하여 u, v 방향으로의 변위, delta_u, delta_v를 계산하고 현재 접근되는 텍셀 좌표에 이 변위를 각각 더하여 다음 텍스쳐 좌표를 예측할 수 있다. 식 4.3과 4.4는 현재 텍스쳐 좌표인 current_u, current_v와 이전 텍스쳐 좌표인 previous_u, previous_v의 차를 이용하여 delta_u와 delta_v를 계산하는 방법을 보여준다.

$$delta_u = current_u - previous_u$$ .... 식 4.3

$$delta_v = current_v - previous_v$$ .... 식 4.4

식 4.5와 식 4.6에서는 식 4.3과 식 4.4에서 구한 delta_u, delta_v를 현재 좌표에 더하여 다음 접근될 좌표를 예측할 수 있다.

$$next_u = current_u + n * delta_u$$ 식4.5

$$next_v = current_v + n * delta_v$$ 식4.6

4.1절에서 제안하는 미리 읽기 기법 1과 다른 점은 원근 보정(Perspective Correction) 이후의 좌표를 이용한다는

점이다. 이 텍셀 좌표들은 직선 위에 존재하지만 delta_u와 delta_v의 값은 직선 위에서 달라지게 된다. 따라서 이 기법으로 예측된 텍셀의 좌표는 다음에 접근될 정확한 텍셀은 아니다.

그러나 이 텍셀이 속한 블록은 대부분 다음에 접근될 가능성이 크며, 실험을 통해 미리 읽기의 정확성이 큼을 확인 할 수 있다. 4.1절의 미리 읽기 방법과 마찬가지로 현재 텍셀이 속한 블록과 다른 블록을 예측하기 위해 블록 크기에 기반하여 각 delta_u와 delta_v값에 n을 곱한다.

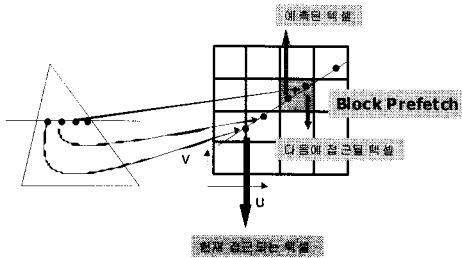


그림 7. 변위 기반의 블록 예측 방법
Fig7. Block Prediction Method Based on Displacement

4.3 접근 방향 기반의 블록 예측 방법

본 절에서 소개하는 기법은 블록 좌표에 기반한 예측 방법이다. 텍셀 주소를 구하기 위해서 주소 생성 (address generator)단계에서는 블록 좌표를 먼저 계산하게 된다. 이 때 계산된 블록 좌표를 이용하여 텍스쳐 맵 접근 방향에 따라 다음 블록을 예측한다. 텍스쳐 맵 접근 방향은 삼각형 시작시 한번만 계산되며 4.2절의 식 4.3과 식 4.4를 이용하여 계산된 delta_u와 delta_v를 이용한다.

삼각형의 시작점에서는 delta_u와 delta_v의 값을 계산할 수 없기 때문에 미리 읽기를 할 수 없다. 따라서 삼각형의 시작점을 제외한 점에서는 delta_u와 delta_v을 이용하여 텍스쳐 맵의 접근 방향을 다음과 같이 세 방향으로 나눌 수 있다. 그림 8은 45도보다 작은 접근 방향을 갖는 경우로 수평 방향의 인접한 블록을 미리 읽기를 한다. 그림 9는 45도 방향의 접근 방향을 갖는 경우로 대각선 방향의 인접한 블록을 미리 읽기를 할 수 있으며 그림 10과 같이 45도 이상의 접근 방향을 갖는 경우 수직 방향의 인접한 블록을 미리 읽기 할 수 있다. 이와 같은 방법은 접근 방향을 세 방향으로만 나누었기 때문에 정확한 예측을 하지 못하는 경우가 많다. 예를 들어 그림 10과 같이 접근 되는 경우 두 번째로 사용되는 블록은 예측이 되지만, 세 번째로 사용되는 블록은 두 번째로 사용되는 블록과 수직방향으로 인접하여 있는 블록이 되므로 예측할 수 없게 된다.

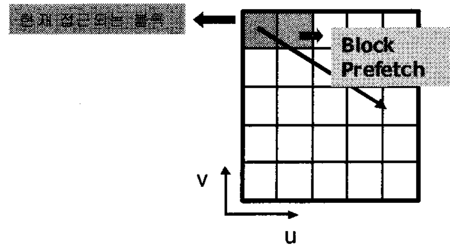


그림 8. 수평 방향의 인접한 블록 미리 읽기
Fig8. Horizontal Adjacent Block Prediction

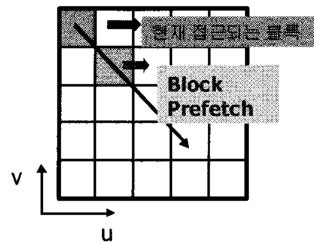


그림 9. 대각선 방향의 인접한 블록 미리 읽기
Fig9. Diagonal Adjacent Block Prediction

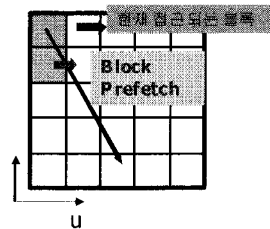


그림 10. 수직 방향의 인접한 블록 미리 읽기
Fig10. Vertical Adjacent Block Prediction

V. 텍스쳐 캐시 구조

본 절에서는 논문에서 제안하는 텍스쳐 캐시의 구조를 설명한다. 본 논문에서 제안하는 텍스쳐 캐시는 주 캐시와 미리 읽기 버퍼로 구성된다. 주 캐시는 직접 사상(direct

mapped)캐시 또는 2-way 캐시로 이루어 질 수 있다. [2]의 논문에서는 2-way 이상의 집합 연관 사상(set associativity)을 사용하여도 미스 율이 크게 줄어들지 않음을 보여주었다.

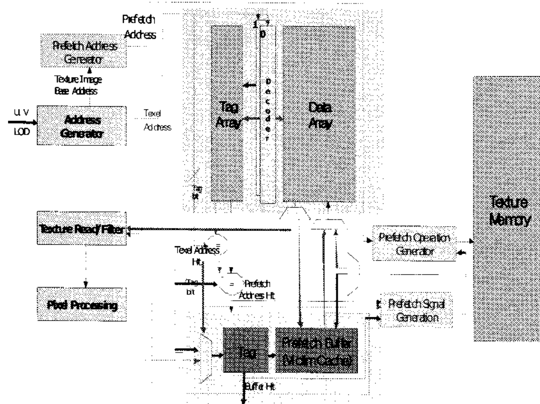


그림 11. 텍스쳐 캐시 구조
Fig11. Overview of Proposed Texture Cache

미리 읽은 블록을 위해 완전 연관 사상(fully associative) 캐시로 이루어진 미리 읽기 버퍼를 이용한다. 이를 버퍼를 이용함으로써 현재 주 캐시에서 접근되고 있는 블록이 미리 읽은 블록에 의해 교체되는 현상을 줄일 수 있다. 또한 작은 캐시를 이용함으로써 발생하는 캐시의 충돌 미스를 줄이기 위해 미리 읽기 버퍼를 희생 캐시(victim cache)로 동시에 이용한다. 이는 하나의 스팬 라인(span line)에 사용된 블록은 다음 인접한 스팬 라인에서도 사용될 가능성이 크기 때문이다. 그러나 캐시 크기가 감소하면 다음 스팬 라인에서 사용될 수 있는 블록이 교체될 가능성이 커진다. 따라서 주 캐시에서 교체되는 블록을 미리 읽기 버퍼로 복사함으로써 인접한 다음 스팬 라인에서 블록이 다시 이용될 때 텍스쳐 메모리에 다시 접근하는 대신 미리 읽기 버퍼에서 가져옴으로써 미스 패널티(penalty)를 줄일 수 있게 된다.

제안하는 텍스쳐 캐시의 구조는 그림 11와 같다. 앞에서 설명한 것과 같이 직접 사상 캐시 또는 2-way 캐시인 주 캐시와 완전 연관 캐시인 미리 읽기 버퍼로 이루어 진다. 미리 읽기 주소 생성 자(prefetch address generator)는 4절에서 설명한 미리 읽기 블록 예측 방법에 따라 주소를 생성하며 주소 생성자(address generator)는 현재 픽셀에서 필요한 텍셀 값들을 읽기 위한 텍셀 주소를 생성한다. 텍스쳐 읽기/필터(texture read/fileter) 단계에서는 텍스쳐 캐시에서 읽어온 텍셀 값을 이용하여 삼각선형보간 또는 쌍일차선형 보간의 필터링을 수행하며, 픽셀 처리(pixel processing) 단계에서는 필터링 된 값을 이용하여 안개 효과, 블렌딩, z 테스트 등의 픽셀 처리를 수행한다. 미리 읽기 주소 생성 자에서 생성된 미리 읽기 주소에 대한 요청이 텍스쳐 메모리로 보내지기 전에 미리 읽을 블록이 캐시에 존재하는지 확인하기 위하여 태그 룩업(tag lookup)을 수행해야 한다. 이때, 캐시에 없는 경우에만 텍스쳐 메모리로 요청을 보내게 된다.

텍스쳐 캐시에 대한 접근과 태그 룩업이 동시에 발생하는 경우 성능이 저하되는 문제가 발생하게 된다. 따라서 텍셀에 대한 캐시 접근과 미리 읽을 블록에 대한 태그 룩업을 동시에 할 수 있도록 하기 위해 주 캐시의 태그 부분에 읽기 전용 포트가 하나 더 추가된 SRAM cell[1]을 사용하였다. 텍셀 접근에 대해서는 decoder 0을 사용하고, 미리 읽을 주소의 태그 룩업을 위해서는 decoder 1을 이용하여 워드 라인을 선택한다. 메모리 셀을 읽거나 쓰는 데에는 텍셀 접근에 대해서는 읽기/쓰기 포트인 포트 0을 사용하고 태그 룩업에 대해서는 읽기 전용 포트인 포트 1를 사용하도록 한다. 이러한 포트의 추가는 추가적인 워드 라인과 비트 라인을 사용하여 셀(cell) 크기를 증가시키며, 비교자(comparator)와 sense amplifier와 같은 부가적인 구조들을 추가시킨다. 그러나 본 논문에서는 태그 부분에만 읽기 전용 포트를 추가시키므로 오버헤드는 크지 않다.

텍스쳐 캐시의 동작 과정은 다음과 같다. 주소 생성 자에서 생성된 주소는 먼저 주 캐시를 확인하고 히트가 발생한 경우 텍셀 데이터는 주 캐시에서 필터링 단계로 보내진다. 주 캐시에서 미스가 발생한 경우는 미리 읽기 버퍼를 확인하고 버퍼에 필요한 블록이 존재하면 텍셀 데이터가 필터링 단계로 보내지며 동시에 주 캐시의 내용이 업데이트된다. 이때 주 캐시에서 교체되는 블록은 미리 읽기 버퍼로 복사됨으로써 미리 읽기 버퍼의 블록과 주 캐시의 내용이 서로 바뀌게 된다. 주 캐시와 미리 읽기 버퍼에서 모두 미스가 발생한 경우는 글로벌 미스(global miss)로서 텍스쳐 메모리에 블록을 요청한다.

미리 읽기 주소 생성 자에서 생성된 미리 읽을 블록의 주소는 블록이 이미 캐시에 존재하는지 확인하기 위하여 태그 룩업을 수행해야 한다. 태그 룩업에 대해서는 decoder 1을 통해 워드라인이 선택되고 읽기 전용 포트인 포트 1으로 메모리 셀을 읽는다. 미리 읽기 버퍼에 대한 태그 룩업은 현재 주 캐시의 텍셀 접근이 히트(hit)일때만 수행하도록 한다. 이는 주 캐시에서 텍셀 접근이 히트인 경우 미리 읽기 버퍼의 접근을 수행하지 않기 때문에 미리 읽기 주소가 버퍼의 태그 룩업을 할 수 있게 한다. 미리 읽기 신호 생성자(prefetch signal

generation) 단계에서는 주 캐시에서의 텍셀 접근에 대한 결과, 주 캐시에서의 미리 읽기 주소의 태그 룩업 결과 그리고 버퍼에서의 태그 룩업에 대한 결과를 이용하여 미리 읽기 요청의 발생에 대한 결정을 내린다. 주 캐시에서 텍셀 접근이 히트이고 동시에 주 캐시와 미리 읽기 버퍼에서의 주소에 대한 태그 룩업의 결과가 미스일 때만 미리 읽기 요청이 메모리로 보내지게 된다. 그리고 현재 주 캐시에서의 텍셀의 접근이 미스인 경우에는 텍셀 접근이 미리 읽기 버퍼를 접근해야 하기 때문에 미리 읽기 주소는 버퍼에 대한 태그 룩업을 수행하지 않으며 따라서 주 캐시에서의 미리 읽기 주소에 대한 태그 룩업의 결과에 상관없이 미리 읽기 요청은 발생하지 않는다. 따라서 텍셀 접근이 주 캐시와 버퍼에서 모두 미스인 경우에는 미리 읽기 요청은 발생하지 않고, 글로벌 미스로 인해 요구되는 텍셀에 대한 요청만 메모리로 보내진다.

VI. 실험환경 및 결과

6.1 실험 환경

이 장에서는 앞에서 제안한 저전력 텍스쳐 캐시 기법을 실제 벤치마크 프로그램에 대해 적용해 보았을 때의 성능 향상과 에너지 소모의 감소를 시뮬레이션을 이용하여 확인한다.

본 절에서는 시뮬레이션 실험환경에 대해서 설명한다. 본 논문에서 제안하는 미리 읽기 기법과 텍스쳐 캐시구조를 실제 3차원 애플리케이션에 적용하여 실험하기 위해 트레이스(trace) 기반의 캐시 시뮬레이터인 Dinero IV[9]를 수정하였다. Dinero IV 시뮬레이터에 입력으로사용되는 메모리 트레이스(memory trace)는 OpenGL ES 라이브러리를 수정하여 2차원 텍스쳐 이미지에 대한 메모리 접근 트레이스를 추출함으로써 얻었다. 각 미리 읽기 기법은 OpenGL ES 라이브러리의 텍셀 주소를 생성하는 부분을 수정하여 구현하였으며, 각 미리 읽기 기법에 따라 텍셀 주소를 생성할 때 이용되는 정보들, 즉 텍스쳐 좌표 또는 블록 좌표, 텍스쳐 맵의 시작주소 등을 이용하여 미리 읽을 텍셀의 주소를 계산하고 메모리 트레이스에 추가하였다. 제안하는 텍스쳐 캐시에서 소모되는 동적 에너지와 정적 에너지를 계산하기 위해서 eCacti 1.0[1]을 이용하였으며, 주 캐시의 태그 부분에만 읽기 전용 포트를 추가한 캐시 구조의 에너지와 면적을 계산하기 위해 이를 수정하여 사용하였다.

표 1. 시뮬레이션 변수
Table1. Simulation Parameter

Parameters	Values
Fragment Generator Clock	100MHz
Memory Clock	100MHz
Memory Latency	13 cycles
Memory Access Energy	32.5 nJ

실험에 사용된 벤치마크는 jelly fish, face model 그리고 check image 의 세가지 애플리케이션이다. Jelly fish애플리케이션은 많은 수의 작은 삼각형으로 이루어져 있으며 여러 개의 텍스쳐 맵을 이용한다. 그리고 각 텍스쳐 맵에 대한 접근 방향이 다르다. Face model 애플리케이션의 경우도 많은 수의 작은 삼각형으로 이루어져 있지만 하나의 텍스쳐 맵을 이용하고 있다. 텍스쳐 맵에 대한 접근 방향은 대부분 수평 방향을 갖는다. Check image는 네 개의 큰 삼각형으로 이루어져 있으며 텍스쳐 맵에 대한 접근은 수평 방향과 45% 이하의 방향을 갖는다.

텍스쳐 캐시에서 소모되는 에너지는 식 6.1을 이용하여 계산한다. 미스 발생시 소모되는 에너지에는 텍스쳐 메모리에 접근할 때 소모되는 에너지를 포함하였다.

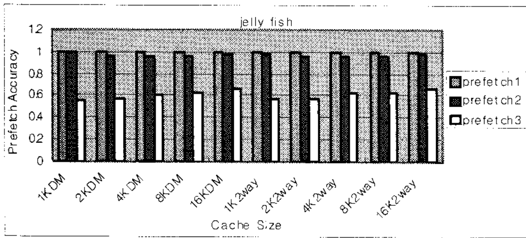
$$E_{total} = E_{dynamic} + E_{static} + E_{prefetch} \quad \cdots\cdots\cdots \text{식 } 6.1$$

캐시에서 소모되는 총 에너지 Etotal은 동적 에너지 Edynamic, 정적 에너지 Estatic 그리고 미리 읽기에 의한 에너지 Eprefetch의 합이다. Edynamic은 식 6.2를 이용하여 계산한다.

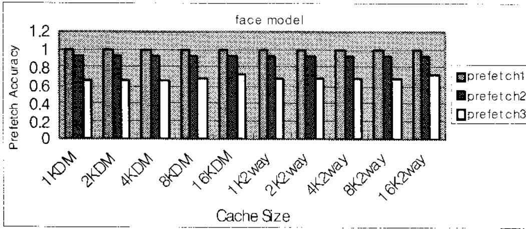
$$E_{dynamic} = N_{cache_hit} * E_{hit} + N_{buffer_hit} * E_{buffer_hit} + N_{global_miss} * E_{global_miss} \quad \cdots\cdots\cdots \text{식 } 6.2$$

Ncache_hit, Nbuffer_hit 과 Nglobal_mis 는 각각 주 캐시에서 히트가 발생한 횟수, 주 캐시에서 미스가 발생하였으나 버퍼에서 히트가 발생한 횟수 그리고 주캐시와 버퍼에서 모두 미스가 발생한 횟수이다. Ebuffer_hit 는 주 캐시에 접근하는 에너지, 미리 읽기 버퍼에 접근하는 에너지 그리고 버퍼와 주 캐시의 블록이 교체될 때 소모되는 에너지를 포함한

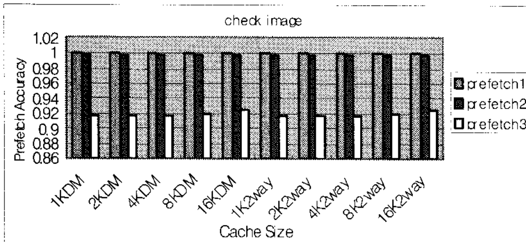
다. 또한 Eglobal은 캐시에서의 미스 에너지와 텍스쳐 메모리에 접근하는데 소모되는 에너지의 합이다. 식 6.3과 식 6.4는 각각 캐시의 정적 에너지와 미리 읽기로 인해 소모되는 에너지이다.



(a)



(b)



(c)

그림 12. 미리 읽기(prefetch) 기법에 따른 정확도 (a) jelly fish (b) face model (c) check image
Fig12. Accuracy of Prediction Methods (a) jelly fish (b) face model (c) check image

$$E_{static} = cycles * E_{static_per_cycle} \quad \text{식 6.3}$$

$$E_{prefetch} = E_{tag_lookup} + E_{memory_access_for_prefetch} \quad \text{식 6.4}$$

cycles는 한 프레임의 사이클 수로, 본 논문에서는 초당 30 프레임과 100MHz의 프레그먼트 생성기(fragment generator)를 가정하였으므로 약 3300000가 된다. Estatic_per_cycle은 한 사이클에 소모되는 정적 에너지이다. 식 6.4의 Etag_lookup 은 미리 읽을 블록이 이미 캐시에 있는지 확인하기 위해 태그 룩업을 수행하는데 소모되는 에너지이고, Ememory_access_for_perefetch는 태그 룩업의 결과 캐시에 없는 경우 블록을 미리 읽기 위해 메모리에 접근하는데 필요한 에너지이다. 표 1은 실험에서 이용한 변수들을 보여준다. 메모리 접근 지연 시간과 에너지 소모에 대한 변수들은 MICRON MT48V8M32LF SDRAM로부터 얻었다.

6.2 실험 결과 및 분석

본 절에서는 4장에서 제안된 세가지 미리 읽기 기법의 정확도를 보여주고 제안된 텍스쳐 캐시를 사용했을 때의 미스율, 에너지 소모 그리고 텍스쳐 캐시가 차지하는 면적을 비교해 본다. 1장에서 보여준 세 개의 애플리케이션에 대해 제안된 방법을 적용하여 실험하였다.

6.2.1미리 읽기기법에 따른 정확도

위의 그림 12는 4장에서 제안한미리 읽기 기법에 따른 정확도를 보여준다. 각 기법의 정확도는 식 6.5과 같은 방법으로 계산하였다.

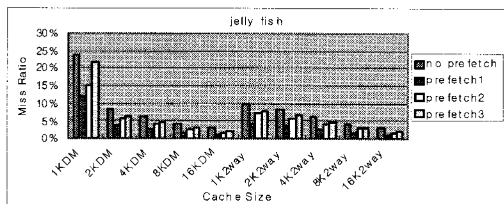
$$\frac{\text{미리 읽기 후 사용된 블록}}{\text{미리 읽은 블록}} \quad \text{식 6.5}$$

계산의 오버헤드가 가장 큰 미리 읽기1(prefetch 1)의 경우 정확도는 거의 100%에 가까웠고, 미리 읽기 2(prefetch 2)를 사용한 경우에도 90%~100%정도의 정확도를 보이고 있다. 미리 읽기 2 기법에서 예측된 블록은 대부분 사용되었기 때문에 90%~100%의 정확도를 보이고 있지만 실제 예측하지 못한 블록이 미리 읽기1에 비해 많기 때문에 미스 율에서는 미리 읽기 1 기법처럼 크게 향상을 보이고 있지 않다. 이러한 결과는 6.2.2 절에서의 미리 읽기 기법에 따른 미스율에서 볼 수 있다.

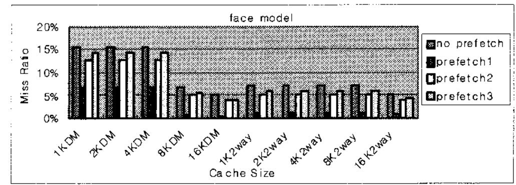
미리 읽기 3(prefetch 3) 기법은 가장 낮은 정확도를 보이고 있다. 또한 애플리케이션에의 텍스쳐 접근 방향 특성에 따라 다른 정확도를 보인다. Jelly fish와 face model의 경우 오브젝트를 구성하는 삼각형의 크기가 작거나 텍스쳐 맵의 접근이 임의의 방향으로 접근되기 때문에 미리 읽기 3을 이용한 경우 다음 사용될 블록의 예측이 잘못되어 불필요하게 미리 읽게 되는 블록이 많아 지게 된다. Check image의 경우는 텍스쳐 맵의 접근 방향이 거의 수평 방향에 가깝기 때문에 미리 읽기 3을 사용한 경우에도 90% 이상의 정확도를 보여주고 있다.

6.2.2 미리 읽기 기법에 따른 미스 율

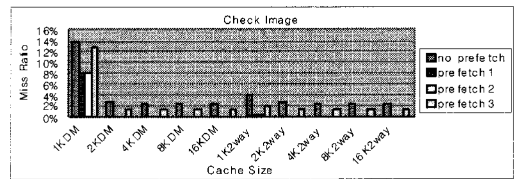
그림 13는 미리 읽기 기법을 이용하였을 때의 미스 율 감소를 보여준다. 미리 읽기 1 기법을 적용한 경우 1K bytes와 2K bytes의 작은 캐시를 사용했을 때 미스 율이 10%에서 최대 15%까지 감소함을 확인할 수 있다. 미리 읽기 2 기법에 대해서는 Jelly fish 애플리케이션에서 1K bytes의 직접 사상 캐시와 미리 읽기 버퍼를 사용하여 10%에 가까운 미스 율 감소를 보여주고 있으며, 2K bytes의 캐시를 사용한 경우에도 5%정도의 미스 율 감소를 보여주고 있다. Face model에서는 미리 읽기 2 기법을 적용한 경우 작은 캐시에서 5% 정도의 미스 율 감소를 보여주고 있다. 이는 face model이 많은 작은 삼각형으로 구성되어 짧은 스팬 라인에 대해서는 예측을 하지 못한 경우가 많기 때문이다. 또한 face model의 경우 텍스쳐 맵의 크기가 비교적 크고 접근 영역이 집중되어 작은 캐시를 사용한 경우 충돌 미스가 많이 발생한다. 이는 인한 캐시 미스는 희생 캐시의 사용으로 크게 감소시킬 수 있었다. 삼각형이 크고 수평 방향에 가까운 접근 패턴을 갖는 check image의 경우 미리 읽기 1과 미리 읽기 2의 방법을 이용하여 거의 0%에 가까운 미스 율을 얻었다. 그러나 1K bytes 직접 사상 캐시에서는 미리 읽기 1과 2를 이용한 경우 8% 정도의 미스 율이 나타남을 볼 수 있다. 이는 미리 읽기를 하기 위해 태그 룩업을 수행할 때는 캐시에 존재 하여 미리 읽기 요청이 발생하지 않았으나 미리 읽기 버퍼의 블록이 주 캐시로 옮겨 지면서 주 캐시의 블록이 교체되는 현상이 발생하였기 때문이다. 이러한 이유로 2-way 캐시에서는 주 캐시에서의 충돌 미스횟수가 적어 0%에 가까운 미스 율을 나타내고 있음을 볼 수 있다. 미리 읽기 버퍼를 희생 캐시로 이용한 경우 이러한 주 캐시에서의 충돌 미스 횟수를 없앨 수 있었다. 미리 읽기 3을 이용한 경우는 모든 애플리케이션의 경우에서 미리 읽기 1과 미리 읽기 2기법에 비해 미스 율 감소가 작았다.



(a)



(b)



(c)

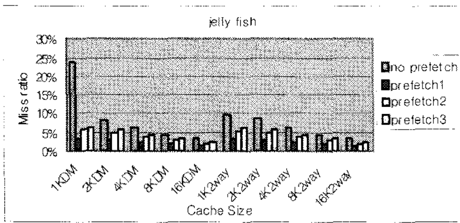
그림 13. 미리 읽기 (prefetch) 기법에 따른 미스 율 (a) jelly fish (b) face model (c) check image

Fig13. Miss Ratio of Prefetch Method율 (a) jelly fish (b) face model (c) check image

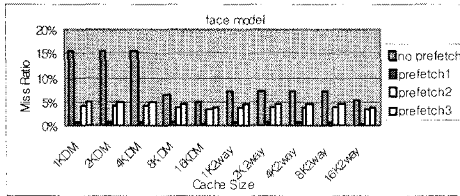
6.2.3 미리 읽기 버퍼를 희생 캐시로 이용한 경우 미스 율

본 논문에서 제안하는 텍스쳐 캐시 구조에서는 미리 읽기 버퍼를 희생 캐시로 동시에 사용하여 작은 캐시를 사용함으로써 발생하는 충돌 미스를 줄일 수 있다. 그림 14는 희생 캐시를 사용했을 때의 미스 율을 보여준다. 미리 읽기 기법과 희생 캐시를 결합하여 실험한 경우 1K bytes와 2K bytes의 캐시를 사용하여 8K bytes나 16K bytes의 캐시를 사용했을 때와 비슷한 미스 율을 보여준다.

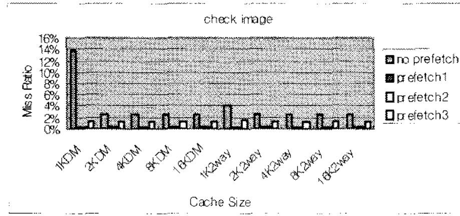
텍스쳐 맵이 2의 지수의 크기를 갖기 때문에 블록간의 충돌미스가 자주 발생한다. 특히 텍스쳐 맵이 큰 경우나 캐시의 크기가 작은 직접 사상 캐시의 경우에는 충돌 미스의 횟수가 더 많아진다. 또한 미리 읽기 요청을 위해 태그 룩업을 수행할 당시에는 주 캐시에 존재하여 미리 읽기 요청이 발생하지 않았지만, 미리 읽기 버퍼에서 주 캐시로 복사될 때 주 캐시의 블록을 교체시키는 현상이 발생하여 예측은 되었지만 미리 읽기 요청이 발생하지 않은 경우가 생긴다. 이러한 현상은 1K bytes의 직접 사상 캐시에서 많이 나타났으며 희생 캐시를 이용함으로써 이러한 현상을 줄일 수 있었다. 따라서 제안하는 텍스쳐 캐시를 이용한 경우 큰 캐시를 사용했을 때와의 비슷한 미스 율을 유지할 수 있음을 볼 수 있다.



(a)



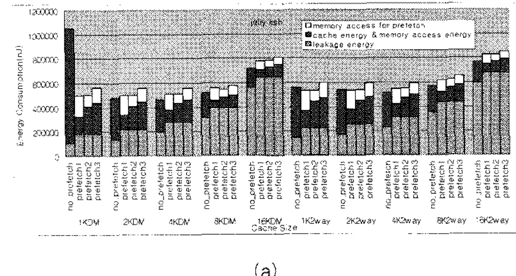
(b)



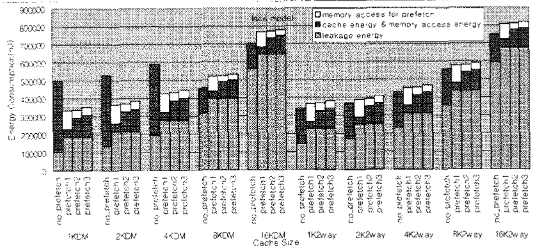
(c)

그림 14. 희생 캐시를 사용한 경우 미스 율 (a) jelly fish (b) face model (c) check image
Fig14. Miss Ratio using Victim Cache (a) jelly fish (b) face model (c) check image

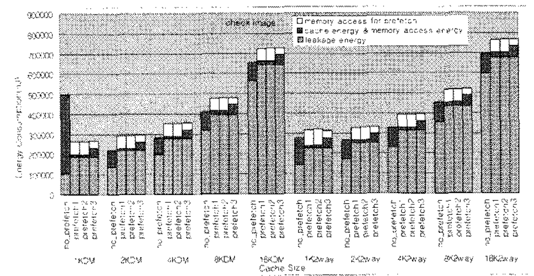
그림 15는 제안하는 텍스쳐 캐시를 사용하였을 때의 에너지 소모를 캐시의 크기에 따라 비교한 결과를 보여준다. 미리 읽기를 하지 않는 기법 (no prefetch)는 미리 읽기 기법을 사용하지 않은 일반적인 캐시에서의 에너지 소모를 보여준다. 미리 읽기 기법을 사용하는 캐시의 결과는 512 bytes의 완전 연관 캐시인 미리 읽기 버퍼를 추가적으로 이용하고 주 캐시의 태그 부분에 포트를 추가하여 에너지를 계산한 결과이다. 따라서 같은 크기의 캐시를 비교하였을 때는 일반적인 캐시를 이용한 경우 에너지 소모가 더 작음을 볼 수 있다. 그러나 캐시의 크기가 커질수록 정적 에너지가 증가하기 때문에 동적 에너지와 정적 에너지를 포함한 전체 에너지 소모는 캐시의 크기가 커질수록 크게 증가한다. 정적 에너지의 계산은 한 프레임을 마치는 동안의 에너지를 고려 하였기 때문에 초당 30프레임을 가정한 경우 약 33ms 동안에 소모된 에너지를 갖는다. 동적 에너지는 한 프레임 동안 캐시에 접근하는데 소모되는 에너지와 미스와 미리 읽기로 인해 메모리에 접근하는데 소모되는 에너지를 포함한다. 미리 읽기 2의 기법을 적용하여 실험한 경우 제안하는 텍스쳐 캐시를 사용했을 때 16K bytes 캐시를 사용했을 때와 비교하여 jelly fish 애플리케이션에서는 23%, face model 에서는 53%의 에너지 소모를 감소시킬 수 있었다.Check image 애플리케이션에서는 1K bytes의 캐시를 사용함으로써 16K bytes의 캐시를 사용하는 경우와 비교하여 60%의 에너지 소모를 감소시켰다.



(a)



(b)



(c)

그림 15. 텍스쳐 캐시에서의 에너지 소모 (a) jelly fish (b) face model (c) check image
Fig15. Energy Consumption of Texture Cache (a) jelly fish (b) face model (c) check image

6.2.5 캐시 크기에 따른 면적 비교

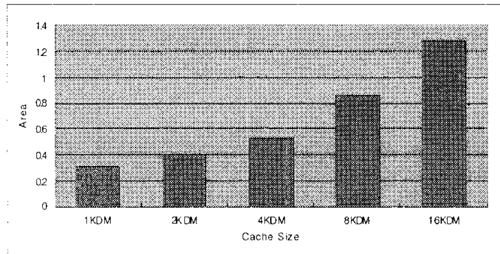


그림 16. 캐시 크기에 따른 면적 비교
Fig16. Area Comparison Varying Cache Size

본 논문에서는 1K bytes 또는 2K bytes의 캐시를 사용하여 16K bytes 캐시의 성능을 유지하고, 에너지 소모를 줄이기 위한 텍스쳐 캐시 구조를 제안하였다. 그림 16은 제안하는 텍스쳐 캐시의 면적을 일반적인 16K bytes의 캐시에 정규화한 값들을 보여준다. 제안하는 캐시의 면적은 태그에 포트를 추가하고 512 bytes의 미리 읽기 버퍼를 포함한 결과이다. 1K bytes와 2K bytes의 제안하는 텍스쳐 캐시를 사용하여 일반적인 16K bytes의 캐시에 비교하여 60%~70%의 면적을 줄일 수 있었다.

VII. 결론

휴대용 기기의 기능이 다양화되면서 휴대용 기기에서 3차원 그래픽을 사용하는 애플리케이션이 보편화 되고 있다. 특히 이들 기기에서 3차원 게임이 가능해지면서, 휴대용 기기에서의 그래픽 하드웨어에 대한 관심이 증가하고 있다. 휴대용 기기가 점점 소형화되고 있고 배터리로 동작하는 특성 때문에 이들 기기에서의 면적에 대한 제약과 전력 소모는 중요한 문제로 부각되고 있다.

따라서 최근 휴대용 기기를 위한 저전력 그래픽스 하드웨어에 대한 관심이 커지고 있다. 본 논문에서는 이러한 휴대용 기기에서의 그래픽스 하드웨어에서 사용되는 텍스쳐 캐시가 차지하는 면적과 에너지 소모를 줄이기 위한 기법을 제안하였으며, 미리 읽기 기법과 희생 캐시의 사용으로 성능을 유지하면서 텍스쳐 캐시에서 소모되는 에너지 소모를 23%~60%까지 줄이고 70%정도의 면적을 감소 시킬 수 있음을 보여주었다.

StrongARM 110과 같은 마이크로 프로세서에서 온칩 D-cache가 소모하는 에너지가 전체 에너지의 16% 정도를 차지한다는 사실에 기초해서 볼 때 같은 칩내에 있는 그래픽 하드웨어에서도 이와 동일한 크기의 큰 캐시를 사용한다는 것은 에너지 면에서나 면적 면에서 제약이 있음은 분명하다. 그러나 작은 캐시의 사용에서 얻는 면적과 에너지 소모 감소가 전체 그래픽스 하드웨어에서 차지하는 비중이 어느 정도인지는 아직까지 알 수 없다. 또한 그래픽스 하드웨어에서는 텍스쳐 캐시 이외에 픽셀 캐시가 사용되고 있다. 향후 연구로는 이러한 픽셀 캐시에서 소모되는 에너지의 감소를 위한 연구와 이들 캐시에서 에너지 소모의 감소가 전체에서 차지하는 비중을 알아보기 위한 연구들이 계속 진행될 수 있을 것이다.

참고문헌

[1] M. Mamidipaka and N. Dutt, "*eCACTI: An Enhanced Power Estimation Model for Onchip Caches,*" CECS Technical Report #04-28, 2004.

[2] Z. S. Hakura, and A. Gupta, "The Design and Analysis of a Cache Architecture for texture Mapping," Proc. of ISCA , pp. 108-120, 1997.

[3] H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a texture cache architecture," Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware, pp. 133-142, 1998.

[4] M. Cox, N. Bhandari, and M. Shantz, "Multi-level texture caching for 3D graphics hardware," Proc. of ISCA , pp. 189-196, 1998.

[5] C. H. Kim and L. S. Kim. "Adaptive Selection of an Index in a Texture Cache," Proc. of ICCD, pp. 295- 300, 2004.

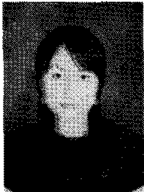
[6] J. H. Woo, J. H. Sohn, H. J. Kim, J. C. Jeong, S. J. Lee, and H. J. Yoo, "A 195mW, 9.1MVertices/s fully programmable 3D graphics processor for low power mobile devices," Proc. of ASSCC, pp. 372-375, 2007.

[7] J. S. Yoon, D. H. Kim, C. H. Yu, and L. S. Kim, "A 3D graphics processor with fast 4D vector inner product units and power aware texture cache," Proc. of CICC, pp. 539-542, 2008.

[8] I. Antochi, B. H. H. Juurlink, A. G. M. Cilio, P. Liuha, "Trading efficiency for energy in a texture cache architecture," Proc. of MPCS, pp. 189-196, 2002.

[9] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers," Proc. of ISCA, pp. 364-373, 1990.
[10] J. H. Lee and S. D. Kim, and C. Wems, "Application-Adaptive Intelligent Cache Memory System," TECS, Vol.1, No.1, pp. 56-78, 2002.
[11] J. Edler, and M. D. Hill, "*Dinero IV trace-driven uniprocessor cache simulator,*" available from Univ. of Wisconsin, 1997.
[12] The Khronos Group, "OpenGL ES," *http://www.khronos.org/opengles/*
[13] 김영옥, 노영섭, 오삼권, "모바일 벡터 그래픽 가속기 설계를 위한 OpenVG API 구현," 한국컴퓨터정보학회 학술발표논문집, 251-255쪽, 2008년 6월.
[14] 박동규, "블록크기가 서로 다른 Direct-Mapped 캐시 메모리의 Trace-Driven 시뮬레이션 알고리즘", 한국컴퓨터정보학회 논문지, 제2권 제1호, 39-50쪽, 1997년 3월.

저 자 소 개



김 현 희

2004: 중앙대학교 컴퓨터공학부 학사.
2006: 서울대학교 전기컴퓨터공학부 석사.
현재: 서울대학교 전기,컴퓨터공학부 박사과정.
관심분야: 멀티코어 임베디드 시스템, 저전력 시스템.

김 지 홍

1986: 서울대학교 계산통계학과 학사.
1988: University of Washington 컴퓨터과학과 석사.
1995: University of Washington 컴퓨터과학 및 공학과 박사.
1995년~1997: 미국 TI 선임연구원.
현재: 서울대학교 전기,컴퓨터공학부 교수.
관심분야: 임베디드 소프트웨어, 저전력 시스템.