모바일 기기에서의 그래픽 기반 디스플레이 미러링

이민우, 양윤식, 박 제임스, 김 진우, 한 탁돈
연세대학교 컴퓨터과학과
e-mail : hellomw@msl.yonsei.ac.kr

Graphic based Mirroring Technology in mobile device

Min-Woo Lee, Yoon-Sik Yang, James Park, Jin-Woo Kim, Tack-Don Han
Dept. of Computer Science, Yon-Sei University

요 약

디스플레이 미러링은 두 기기에 동일한 화면을 출력하는 기법으로, 작은 화면을 갖는 모바일 기기의 단점을 보완하기 위해 사용한다. 그래픽 기반의 디스플레이 미러링은 지연시간이 짧고 화면 품질의 저하가 없는 장점이 있다. 본 연구는 모바일 기기에서 추가적인 하드웨어 없이 수행이 가능한 그래픽 기반의 디스플레이 미러링을 제안한다.

1. 서론

스마트폰과 같은 모바일 기기의 활용도는 이미 PC 를 넘어선 상태이다. 모바일 기기는 다양한 PC 기반의 어플리케이션을 공간의 제약 없이 사용할 수 있을 뿐만 아니라, 카메라와 센서 등의 주변 장치를 사용한 어플리케이션의 활용도 가능하다. 또한 네트워크 기술의 발달로 다른 기기와의 상호 작용을 통한 융합 기술 연구에도 사용된다.

하지만 모바일 기기는 높은 휴대성 대신 제한된 화면 크기를 갖는다. 이러한 화면 크기의 제약은 가시성을 저하시켜 사용자의 모바일 기기 사용을 방해한다. 테블릿 PC 는 모바일 기기들의 화면 제약을 해결하기 위한 방법 중 하나지만, 무게가 증가하여 휴대성이 낮아지는 단점이 있다.

디스플레이 미러링은 휴대성의 저하 없이 모바일 기기의 화면 제약을 해결하기 위한 기법으로 작은 화면 크기를 갖는 기기의 컨텐츠를 큰 화면을 갖는 기기에 출력시키는 기술이다. 현재 디스플레이 미러링은 WiDi[1], Miracast [2], AirPlay[3]와 같이 상업적으로도 개발되고 있으나, 대부분의 기술들이 이미지 기반으로 이루어진다. 이미지 기반 디스플레이 미러링은 H.264 비디오 압축 포멧 기반으로 하드웨어 레벨에서 수행된다. 이와 같은 방식은 전력 소비를 줄일 수 있는 장점이 있지만, FPS 가 높을수록 지연 시간이 길어지고 이미지 티어링이 발생하는 단점이 있다. 뿐만 아니라 이미지 프레임을 압축하여 네트워크를 통해 전송하여 출력하는 방식이기 때문에, 해상도가 커질 경우 픽셀레이션이 발생하여 화면 품질을 저하시킨다.

이미지 기반의 디스플레이 미러링의 문제를 해결하기 위한 방법으로, 해상도의 변화와 관계 없이 항상 일정한 품질의 화면을 출력시키는 그래픽 기반의 렌더링 기술이 있다. 그래픽 기반의 디스플레이 미러링은 이미지 기반의 디스플레이 미러링보다 지연 시간이 짧고 티어링도 없다. 대표적인 그래픽 기반 디스플레이 미러링으로는 Xbounds[5]가 있다. Xbounds 는 추가 하드웨어를 사용한 그래픽 기반의 디스플레이 미러링 기술로, 상업적으로 개발된 대부분의 이미지 기반 디스플레이 미러링보다 성능이 좋다. 하지만 추가적인 하드웨어로 인해 비용이 발생하는 단점이 있다. 본 논문에서는 모바일 기기에서 추가적인 하드웨어가 필요 없는 그래픽 기반의 디스플레이 미러링 연구를 진행하였다. 또한 지연시간 최소화를 위해 명명된 파이프와 멀티스레딩을 사용하였다.

2. 관련 연구

디스플레이 미러링은 하나의 기기(서버)에 보여지는 컨텐츠를 다른 기기(클라이언트)에 동시에 출력시키는 기술로써, 주로 작은 화면크기를 갖는 기기의 컨텐츠를 큰 화면에 출력시키기 위해 사용한다. 디스플레이 미러링은 전송되는 데이터 정보에 따라 이미지 기반 미러링과 그래픽 기반 미러링으로 나누어 진다.

이미지 기반의 디스플레이 미러링은 완성된 이미지 프레임을 전송하여 출력하는 방식이다. 서버 기기는 완성된 프레임의 픽셀 정보를 클라이언트 기기로 전송한다. 전송된 프레임은 클라이언트 기기에서 다시 화면에 출력된다. 하지만 전송된 프레임의 해상도는 서버기기에 맞도록 설정되었기 때문에, 클라이언트 기기의 해상도가 서버 기기와 다를 경우 품질에 영향을 주는 단점이 있다.

그래픽 기반의 디스플레이 미러링은 서버에서 랜더링에 필요한 모든 정보를 클라이언트로 전송하여 다시 랜더링 하는 방법이다. 그래픽 기반의 디스플레이 미러링은 클라이언트 기기에서 다시 랜더링을 하기 때문에 해상도에 관계없이 항상 일정한 화질의 장면

을 미러링할 수 있다.

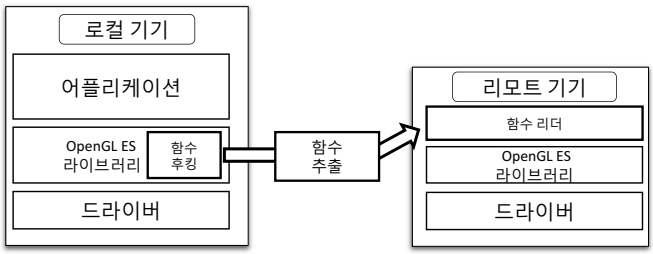
기존 그래픽 기반의 원격 랜더링 기술은 워크스테이션 환경에서 큰 장면을 실시간으로 랜더링하기 위해 사용되었다([4], [6]). 기존 방식들은 큰 장면을 랜더링하기 위해 타일 기반의 출력 방식을 사용하며, 하나의 서버에 다수의 클라이언트를 연결한다.

(그림 1)이미지 기반과 그래픽 기반 디스플레이 미러링

(그림 1)은 이미지 기반과 그래픽 기반 디스플레이 미러링의 품질 차이를 보여준다. 최근 이미지 기반 디스플레이 미러링은 해상도의 증가에 따른 품질 저하 문제를 보완하였지만, 여전히 이미지 티어링이 발생되어 화질을 저하시킨다. 반면 그래픽 기반 디스플레이 미러링은 해상도나 전송속도와 관계 없이 항상 일정한 화질을 유지한다.

그래픽 기반의 디스플레이 미러링은 이미지 기반 디스플레이 미러링보다 품질이 더 좋을 뿐만 아니라 지연 시간도 짧다. 그 이유는 이미지 기반 미러링 방식은 완료된 프레임을 인코딩하여 전송하는 반면, 그래픽 기반 미러링 방식은 서버 기기가 랜더링 됨과 동시에 그래픽 정보를 클라이언트로 전송하기 때문이다.

3. 전체 시스템 구성



(그림 2) 전체 미러링 시스템 구성

(그림 2)은 그래픽 기반의 디스플레이 미러링의 전체 시스템 구성을 개념적으로 묘사하고 있다. 본 기술의 구성은 크게 미러링의 대상이 되는 로컬 기기와 미러링된 영상을 출력하는 리모트 기기로 나눌 수 있으며, WiFi Direct 를 통해서 두 디바이스가 연결된다.

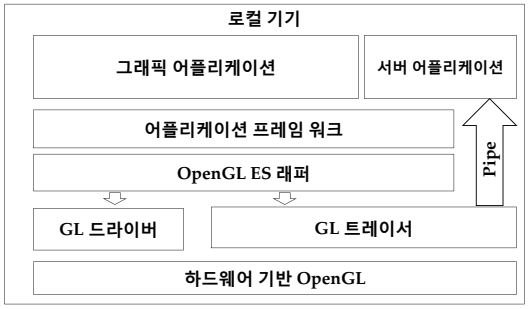
로컬 기기는 미러링에 대한 정보를 전송해야 하기 때문에 서버의 역할을 하며, 이를 위해 openGL ES 라이브러리에서 후킹한 데이터를 리모트 기기에 전송한다. 리모트 기기는 로컬 기기로부터 미러링된 정보를 받아 영상을 출력해야 하기 때문에 클라이언트의 역할을 하며, 이를 위한 별도의 그래픽 환경이 구축되어 있다.

서버는 미러링 인터페이스 관리와 클라이언트와의 연결을 위해 서버 어플리케이션을 사용한다. 클라이언트는 2 개의 스레드를 사용하여 Socket 통신과 영상 출력을 독립적으로 수행할 수 있도록 설계하였다.

4. 지연시간 최소화를 위한 최적화 기법

4.1 파이프 기반의 스레드 통신

서버-클라이언트간 데이터 통신은 그래픽 어플리케이션의 원활한 동작을 위해 독립적으로 수행해야 할 필요가 있다. 만약 그래픽 어플리케이션이 수행되는 스레드에서 데이터 통신이 이루어 진다면, 데이터 통신 환경에 따라 클라이언트뿐만 아니라 서버의 FPS 도 감소할 것이다. 따라서 본 기술에서는 서버 어플리케이션을 개발하여 그래픽 어플리케이션과 독립적으로 수행되도록 하였다.

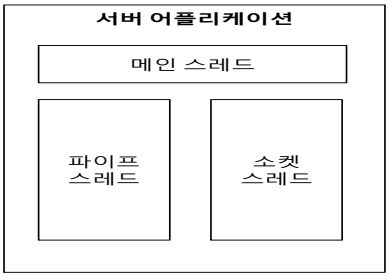


(그림 3) 파이프를 사용한 어플리케이션 간의 통신

(그림 3)은 파이프를 사용하여 어플리케이션 간의 데이터 통신이 어떻게 이루어지는지를 보여준다. 그래픽 어플리케이션을 수행하는 스레드는 GL 함수가 호출될 경우 GL 트레이서를 통해 함수 정보를 후킹한다. GL 트레이서에서 후킹된 GL 함수 정보는 명명된 파이프를 통해 서버 어플리케이션으로 전달된다.

서버 어플리케이션은 파이프에서 읽어온 GL 함수 데이터를 클라이언트에 전송하는 역할을 수행한다. 이때 클라이언트에 보낼 데이터는 데이터의 특성과 관계 없이 일정 크기 단위로 전송한다. 또한 서버 어플리케이션은 OpenGL 어플리케이션과의 독립적인 수행을 위해 백그라운드 서비스가 가능하도록 하였다.

4.2 멀티 스레딩을 사용한 어플리케이션 구현

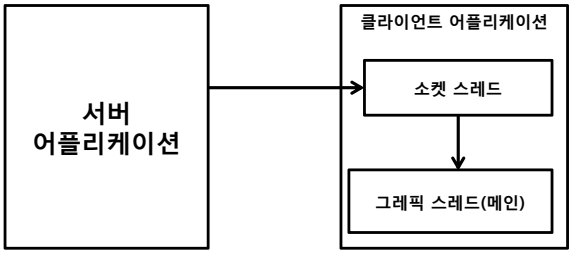


(그림 4) 멀티 스레딩기반의 서버 어플리케이션 구조

(그림 4)은 서버 어플리케이션의 내부 구조를 보여

주는 그림이다. 서버 어플리케이션은 3 개의 스레드로 구성되며 그 역할에 따라 메인 스레드와 파이프스레드, 소켓 스레드로 구분된다.

메인 스레드는 소켓 스레드와 파이프 스레드를 생성하거나, 기능 설정을 위한 UI 환경을 구축한다. 소켓 스레드는 클라이언트의 접속을 승인하는 역할을 한다. 파이프 스레드는 내부 통신을 통해 전달받은 GL 함수 정보를 소켓에 담아 클라이언트에 전달하는 역할을 한다. 클라이언트 어플리케이션은 크게 Wi-Fi 다이렉트를 통해 서버와 연결하는 부분과 서버에서 받은 OpenGL 함수들을 실행하여 서페이스 뷰에 그릴 수 있는 부분으로 나뉜다. (그림 5)은 클라이언트 어플리케이션의 구조를 보여준다.



(그림 5) 클라이언트 어플리케이션 구조

Wi-Fi 다이렉트를 통해 서버와 연결하는 부분은 소켓 스레드가 처리한다. 뿐만 아니라 서버에서 전송한 데이터를 추가 작업 없이 파이프를 통해 그래픽 스레드로 전송하는 역할을 한다.

GL 함수를 실행하여 서페이스 뷰에 그리는 부분은 그래픽 스레드에서 수행된다. 또한 그래픽 스레드는 소켓 스레드에서 전송받은 데이터를 다시 헤더와 변수, 포인터 데이터로 다시 변환한다.

4.3 서버 어플리케이션에서의 데이터 처리 간소화

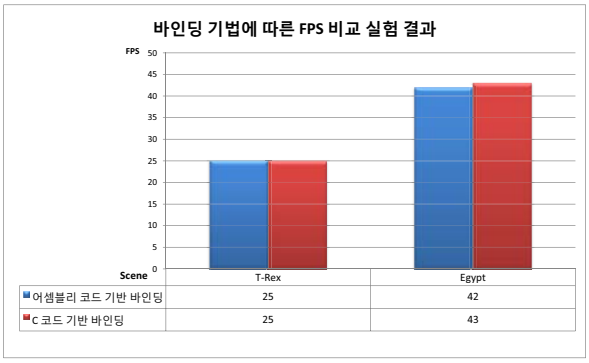
파이프를 통해 서버 어플리케이션으로 전송될 데이터는 FIFO 파일에 저장된다. 서버 어플리케이션은 FIFO 파일에 저장되어 있는 데이터를 읽을 때, 별도의 처리를 하지 않도록 하여 데이터 처리를 간소화하였다. 별도의 처리를 하지 않는 대신 서버 어플리케이션은 데이터를 일정 크기 단위로 클라이언트에 전송한다. 전송된 데이터는 클라이언트에서 디코딩하여 해당 함수를 호출한다.

5. 실험 및 분석

그래픽 기반의 디스플레이 미러링은 Android OS 환경에서 구축하였으며, 로컬 기기는 함수 후킹을 위해 OpenGL ES 라이프러리의 접근이 가능한 Google Nexus4 스마트폰으로 하였다. 무선 네트워크 연결은 WiFi 다이렉트를 사용하였으며, 서버와 클라이언트 어플리케이션은 Java 와 C 코드 기반으로 개발되었다.

5.1 바인딩 기법에 따른 랜더링 성능 실험

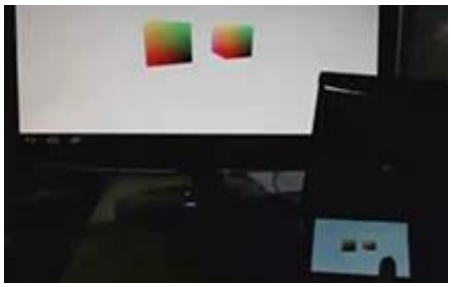
바인딩은 함수 포인터를 사용해 함수 포인터가 가리키는 드라이버 주소에 접근하는 방식으로써, 접근하는 방식에 따라 패스트 바인딩과 슬로우 바인딩 기법으로 나뉜다. 패스트 바인딩은 어셈블리 코드기반으로, 점프를 통해 원하는 함수를 드라이버에 전달한다. 점프 방식의 바인딩기법은 함수의 재호출이 생략되기 때문에 아규먼트 패싱이 가능하다. 점프 기반의 방식은 아규먼트 패싱으로 인한 성능 향상 효과가 있지만, 주소 접근 방식이라 후킹 함수의 적용이 어려운 단점이 있다. 슬로우 바인딩은 C 코드 기반의 함수 호출 방식으로 아규먼트 패싱이 중복되는 단점이 있다. 하지만 함수의 재호출이 발생하기 전 GL 함수 데이터를 쉽게 후킹할 수 있는 장점이 있다.



(그림 6) 바인딩 기법에 따른 랜더링 성능 실험

(그림 6)은 바인딩 기법에 따른 랜더링 성능을 실험한 결과이다. 이 실험은 후킹이 간편한 슬로우 바인딩과 패스트 바인딩의 성능 차이를 확인하기 위해 진행되었다. 실험 결과 모바일에서 사용하는 작은 데이터 크기의 장면들을 랜더링할 경우 두 바인딩 기법의 차이는 없었다.

5.2 디스플레이 미러링의 지연시간 실험



(그림 7) 지연시간 측정 실험 장면

이 실험은 이미지 기반의 디스플레이 미러링 방식과 본 논문에서 제안한 방식의 지연시간을 측정하기 위해 진행되었다. <표 1 >는 기존 이미지 기반의 디스플레이 미러링 기술들과 본 논문에서 제안된 방식의 최소 지연시간을 비교한 표이다. Miracast 의 경우, 모바일간의 미러링을 지원하지 않기 때문에, TV 를 사용하여 실험하였다. WiDi 는 Intel 에서 지원하는 기술로써 Intel 하드웨어가 포함된 장치에서 실험을 진행하였다. 실험에 사용된 어플리케이션은 **(그림 7)**과 같다.

<표 1> 기존 기술과 제안된 방식의 지연시간 비교

	Miracast	WiDi	AirPlay	제안된 방식
서버	스마트폰	노트북	테블릿 PC	스마트폰
클라이언트	TV	동글	노트북	모니터
지연시간	800ms	120ms	110ms	33ms

실험 결과 제안된 방식의 지연시간보다 Miracast 의 지연시간이 20 배 이상 길었다. 그 외 다른 기술들도 제안된 방식보다 3 배 이상 긴 지연시간을 보였다.

제안된 방식이 다른 기술들보다 지연시간이 짧은 이유는 지연시간이 서버 기기의 랜더링 후 전송되는 마지막 데이터의 크기에 영향을 받기 때문이다. 기존 기술들은 이미지 기반의 방식이기 때문에, 마지막으로 전송되는 데이터는 프레임 정보다. 제안된 방식은 그래픽 기반의 미러링 방식이므로, 한 프레임을 랜더링하기 위해 사용된 마지막 GL 함수 정보의 크기에 영향을 받는다.

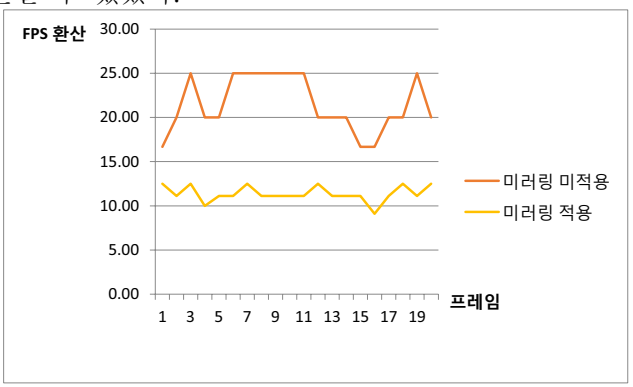
5.3 게임 환경에서의 디스플레이 미러링 실험

이 실험은 데이터크기가 큰 장면에서의 미러링 지연시간을 비교하고, 그래픽 기반의 미러링이 전체 성능에서 차지하는 비율에 대해 조사하기 위해 진행하였다. 실험을 위해 사용된 어플리케이션은 앱스토어에서 다운로드가 가능한 게임으로, 평균 버텍스 수는 1,500 에서 2,000 개다(그림 8).

(그림 8) 게임에서의 미러링 성능 실험 장면

(그림 9)은 미러링을 연결했을 경우와 연결하지 않았을 경우의 FPS 를 보여준다. 제안된 방식은 전송과 랜더링을 독립적으로 수행하기 위해 서버 어플리케이션을 구현하였다. 하지만 그래픽 어플리케이션에서 서버 어플리케이션으로 데이터를 전송하는 과정에서 크기가 큰 GL 함수 정보를 전송할 때, 오버헤드가 발생하였다. 또한 그래프의 굴곡이 디스플레이 미러링을 사용할 때 작아진다는 것은 디스플레이 미러링으로 인한 성능 저하가 전체 성능에 큰 영향을 미친다는 것을 의미한다.

해당 어플리케이션의 지연시간 실험 결과는 70ms - 135ms 이었다. Miracast 의 지연시간보다 약 10 배 이상 짧은 시간이다. 이 결과를 통해 그래픽 기반의 디스플레이 미러링이 모바일 기기에서 이미지 기반의 디스플레이 미러링보다 더 높은 효율을 보이는 것을 확인할 수 있었다.



(그림 9) 미러링 실험 결과

6. 결론 및 추후 연구 방향

본 연구를 통해 그래픽 기반의 미러링이 모바일 기기에서 추가적인 하드웨어 없이 수행이 가능하다는 것을 증명하였다. 또한 그래픽 기반 미러링 연구를 통해 장면을 화질의 저하 없이 두 기기에 출력하도록 하였다. 추후 진행할 연구로는 쉐이더가 포함된 어플리케이션 미러링과 FPS 를 높일 수 있는 압축방식 연구가 있다.

현재 본 연구는 OpenGL ES 1.0 의 함수 중 일부만 구현되어 있다. 하지만 대부분의 어플리케이션이 쉐이더를 포함한 OpenGL ES 2.0 을 사용하기 때문에 해당 어플리케이션을 미러링하기 위한 연구가 필요하다.

압축방식 연구는 두 기기의 FPS 를 높이기 위한 연구로써, 버텍스가 많은 장면을 미러링하기 위해 필요하다. 버텍스 압축은 지연시간 발생을 최소화하기 위해, 라이브러리 레벨에서 진행된다.

또한 본 연구의 실용적 활용을 위해, 이미지 기반 미러링 기술을 추가할 예정이다. 이미지 기반 미러링은 이미지 기반 어플리케이션을 미러링하는 역할을 한다. 위 기술을 실현시키기 위해서 두 미러링 기술을 호환시키기 위한 추가 정보를 클라이언트에 보내도록 한다.

7. 감사의 글

본 논문은 LG 전자의 지원을 받아 수행 되었음.

8. 참고자료

[1]https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/intel-wireless-display.html?, Intel WiDi

[2]http://www.wi-fi.org/wi-fi-certified-miracast%E2%84%A2, Wi-Fi ALLIANCE Miracast

[3] http://www.apple.com/airplay/, APPLE AirPlay

[4] Humphreys et al. “WireGL: A Scalable Graphics System for Clusters”, SIGGRAPH '01, 2001

[5]http://www.dreamchip.de/products/xbounds.html, Dream Chip xbounds

[6] Humphreys et al. “Chromium: a stream-processing framework for interactive rendering on clusters” ” *ACM Trans. Graph.*, vol. 21, no. 3, pp. 693–702, Jul. 2002.