http://dx.doi.org/10.5392/JKCA.2013.13.08.044

3차원 그래픽스 기하 파이프라인 기반의 래스터 파이프라인 구현

Raster Pipeline Implementation based on 3D Graphics Geometry Pipelines

백낙훈
경북대학교 IT대학 컴퓨터학부

Nakhoon Baek(oceancru@gmail.com)

요약

래스터 연산은 트루 컬러 이미지(픽스맵)나, 단색 이미지(비트맵)을 표현하기 위해서 광범위하게 사용된다. 이 기능은 이미지 프로세싱 기능이나, 폰트 출력 시에 강하게 요구된다. 반면에, OpenGL ES 하드웨어 등을 포함하는, 현재의 모바일 그래픽스 플랫폼들에서는 이 기능을 직접 제공하지는 않는다. 모바일 그래픽스 플랫폼들에서 이러한 래스터 연산을 완벽히 제공하기 위해서, 본 논문에서는 그래픽스 이미지들을 3차원 점들의 집합으로 해석하고, 풀-소프트웨어 구현 방식으로, 이들 3차원 점들을 전형적인 3차원 기하 파이프라인으로 처리하게 했다. 구현 결과는 충분한 실행 속도를 보였고, 정확도를 증명하기 위한 공식 검증 테스트(conformance test)들을 모두 통과하였다.

■ 중심어 : | 래스터 연산 | 기하 파이프라인 | 소프트웨어 구현 |

Abstract

Raster operations are widely used to display full-color graphics images (or pixmaps) and single-color images (or bitmaps). These features are strongly needed for image processing applications and font output. However, current mobile graphics platforms, including OpenGL ES hardware implementations, do not directly support these features. To fully support those raster operations on the mobile graphics platforms, we interpreted the graphics images as a set of 3D points, and processed those 3D points through the typical 3D geometry pipelines, in a full-software implementation. Our implementation shows sufficient execution speeds, and passed the official conformance tests to show its correctness.

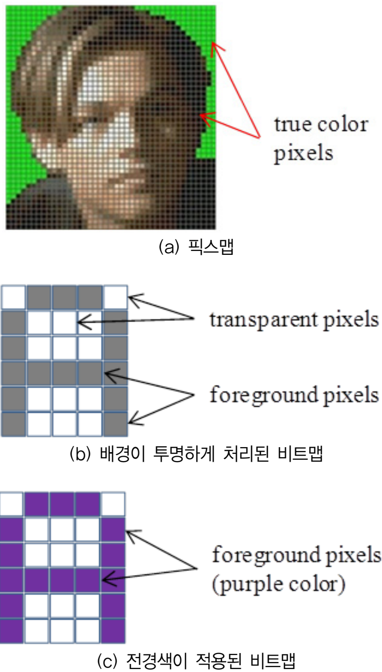
■ keyword : | Raster Operation | Geometry Pipeline | Software Implementation |

I. 서 론

컴퓨터 그래픽스 분야에서는 [그림 1]에서와 같이, 래스터(raster) 형태로 저장된 이미지(image)를 화면에 그대로 보여주는, 래스터 출력 기능이 반드시 요구된다. 래스터 이미지는 일반적으로 픽셀(pixel) 값들의 2차원 사각형 배열로 표현된다. 이러한 이미지들을 화면에 그대로 표현하는 방식은 웹 브라우저(web browser)나 워드 프로세서(word processor)등의 일반적인 프로그램들에서도 제공되는 기능이다. 따라서, 래스터 출력 기능들은 대부분의 그래픽스 플랫폼들에서 전통적으로 제공되어 왔다[1-3].

* 본 연구는 문화체육관광부 및 한국콘텐츠진흥원의 2013년도 문화콘텐츠산업기술지원사업의 연구결과로 수행되었음.
접수일자 : 2013년 05월 16일
수정일자 : 2013년 07월 15일
심사완료일 : 2013년 07월 23일
교신저자 : 백낙훈, e-mail : oceancru@gmail.com

자주 사용하는 래스터 이미지들은 정보의 내부 저장 방식에 따라, 크게 픽스맵(pixmap)과 비트맵(bitmap)으로 구분한다. 픽스맵은 보통 [그림 1](a)에서와 같이, 픽셀마다 RGBA(red, green, blue, alpha) 또는 RGB(red, green blue) 형태의 트루 컬러(true color) 색상 값을 저장한다. 화면에는 저장된 이미지가 그대로 재생되는 방식으로 사용된다.



(a) 픽스맵

(b) 배경이 투명하게 처리된 비트맵

(c) 전경색이 적용된 비트맵

그림 1. 픽스맵과 비트맵의 예

비트맵의 경우는 픽셀 별로 0 또는 1의 1비트 값만을 저정한다. 비트맵을 출력할 때는 [그림 1](b)에서와 같이, 그대로 흑백 이미지(black-white image)로 재생할 수도 있지만, 별도의 전경색(foreground color)을 설정하여, 비트 값이 1인 부분은 전경색으로 나오게 하고, 비트 값이 0인 부분은 출력하지 않아서 투명하게 처리하는 방식을 주로 사용한다. 이러한 투명 처리가 가능한 비트맵 출력 방식은 [그림 1](c)에서와 같이, 폰트(font) 출력에 유용하다.

래스터 이미지를 처리하는 래스터 파이프라인(raster pipeline)은 워크스테이션이나 PC에서 사용하는 전통적인 그래픽스 하드웨어에서는 반드시 필요한 기능으로 분류되지만, 스마트 폰, 태블릿 PC 등을 대상으로 하는 모바일 그래픽스(mobile graphics) 분야에서는 조금 다른 관점을 가지고 있다. 모바일 그래픽스 분야는 스마트 폰과 태블릿 PC 시장의 급격한 확대로, 그 영향력이 커지고 있고, 현재는 완전한 3D 그래픽스 출력 기능을 가지고 있다[4].

현재의 모바일 그래픽스용 그래픽스 표준들로는 OpenGL ES 표준들이 주로 사용되고 있다. 이들은 표준의 제정 당시부터 상대적으로 열악한 저수준 하드웨어에서의 사용을 상정하였다. 그 결과로, 이들 표준에서는 불필요하거나, 상대적으로 사용 빈도가 떨어지는 기능들을 제거하여, 생산비를 낮추고, 제한된 자원을 효율적으로 사용하도록 하였다. 이 선택 과정에서, 래스터 파이프라인은 의도적으로 제거되었다[5]. 당시에는 다른 고급 3차원 그래픽스 기술의 발전으로, 래스터 기능들에 대한 요구가 점차 줄어들 것으로 예상하였으나, 비트맵 폰트의 출력이나, 이미지 프로세싱 연산들에서는 지금도 여전히 래스터 연산들이 요구되고 있다.

모바일 그래픽스 분야에서는 현재 OpenGL 계열의 라이브러리들이 대세를 이루고 있다. OpenGL 계열은 현재 워크스테이션, PC, 임베디드 시스템 등의 거의 모든 컴퓨터 플랫폼들에서 가장 널리 사용되는 그래픽스 라이브러리이다. 원래의 OpenGL은 워크스테이션과 PC 용으로 개발되었고, 하드웨어 상에 별도의 래스터 파이프라인을 설치하도록 하여, 다양한 래스터 연산을 제공했다[6].

반면에, 모바일 그래픽스 분야를 위해 별도로 제정된 OpenGL ES (embedded system) 버전 1.0, 1.1, 2.0, 3.0 등에서는 일체의 하드웨어 래스터 파이프라인이나, 이를 사용하기 위한 API 함수들을 제거하였다. 이에 따라, 픽스맵이나 비트맵 이미지를 직접적으로 출력할 수 없는 상황이다[5][7][8].

OpenGL ES 계열의 초기 설계 당시에는 3D 그래픽스 처리의 핵심이 되는 기하 파이프라인(geometry pipeline)에서 제공하는 텍스처 매핑(texture mapping) 기법으로 픽스맵이나 비트맵을 간접적으로 처리할 수 있다고 판단하였다[5]. 반면에, 기존의 다양한 OpenGL

응용 프로그램들에서는 픽스맵을 이용한 트루 컬러 이미지 출력이나, 비트맵을 이용한 문자 출력 등이 예상외로 빈번하게 사용되고 있다. 또한, 텍스처 매핑 기법을 사용하려면, 화면에 단순한 2차원 이미지를 출력하려고 해도, 3차원 기하 파이프라인에서의 처리를 위해, 3차원 물체를 구성하고, 카메라와 조명을 새로 설정하고, 텍스처를 입력하는 단계를 모두 거쳐야 한다는 점은 상당히 번거로운 작업이 된다.

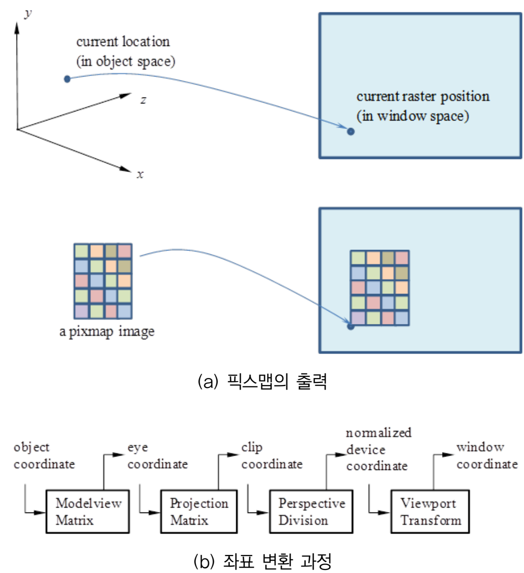
가장 먼저 나타난 기술적인 문제는 기존 프로그램들의 문자 출력이 대부분 비트맵 방식으로 구현되었다는 것이다. 예를 들어, 많은 플랫폼에서 사용하는 GLUT (OpenGL Utility Toolkit) 라이브러리에서는, 초기 OpenGL 의 영향으로, glutBitmapCharacter(...) 함수를 사용하여 비트맵 방식의 문자를 출력하는 것이 일반적이다[9]. 문제는 모바일 그래픽스 분야의 표준인 OpenGL ES 에서는 더이상 이러한 비트맵에 대한 처리를 지원하지 않는다는 것이다. 이것은 기존 프로그램들의 포팅(porting) 시에 심각한 문제가 될 수 있다.

본 논문에서는 래스터 파이프라인이 별도로 제공되지 않는 모바일 그래픽스 표준들에서, 픽스맵과 비트맵의 출력을 비롯한 래스터 파이프라인의 기능들을 제공하는 방법을 제시한다. 기존의 문헌에서는 텍스처 매핑 기법으로 처리가 가능하다고 간단히 언급되어 있었지만[10], 저자들이 아는 한, 실제적 구현 방법이나, 상세한 처리 내용들이 출판된 적은 없었다. 특히, 우리는 텍스처 매핑으로 래스터 파이프라인을 대체할 수 있을 것이라는 기존의 기대들에 반하여, 실제 구현 시에 의외로 문제가 생길 수 있는 상황들을 발견하였다. 우리는 픽스맵과 비트맵을 대응되는 점 집합(point set)으로 보고, 3차원 기하 파이프라인을 통하여 정확한 최종 출력을 얻는 방법을 제시한다[11].

2장에서는 래스터 연산의 세부 사항들과 이에 연관된 기술적 문제점들을 보인다. 3장에서는 본 논문에서 제시하는 구현 전략에 따른 알고리즘들을 제시한다. 구현 결과와 이에 대한 분석은 4장에서 다루고, 마지막으로 5장에서 결론과 향후 과제를 보인다.

II. 문제 분석 및 구현 전략

래스터 파이프라인은 이미지(image)나 폰트(font) 정보를 담고 있는 2차원 색상 배열(color array)을 주어진 그대로 화면에 출력하는 기능을 담당한다. 간단한 2차원 구현들에서는 2차원 화면 상의 좌표를 주면, 대응되는 2차원 이미지가 주어진 크기 그대로 화면에 출력되는 방식을 취하기도 한다[12]. 반면, 3D 그래픽스 라이브러리들에서는 주된 처리 대상이 되는 3차원 기하 물체(geometry object)와의 융합적인 사용을 위해, 처리 방법이 보강되어 있다[6]. 여기서는 OpenGL 계열을 기준으로 각각을 설명하되, 설명의 편의상, 픽스맵 출력 방식을 먼저 설명하고, 비트맵 출력 방식을 보이겠다.



(a) 픽스맵의 출력

(b) 좌표 변환 과정

그림 2. 픽스맵 이미지의 처리

픽스맵의 출력은 출력 위치를 설정한 후에, 출력할 픽스맵을 보내는 방식이 된다. 이를 위한 raster 함수는 다음과 같다:

RasterPos(float *x*, *y*, *z*) : 래스터 출력이 나와야 할 위치인 *current raster position*을 3차원 좌표로 설정한다. [그림 2](a)에서와 같이, 이 3차원 좌표는 통상 현재 설정된 3차원 기하 파이프라인의 설정을 그대로 사용

하여, 대응되는 3차원 기하 좌표를 기준으로 출력 위치를 설정할 수 있게 한다.

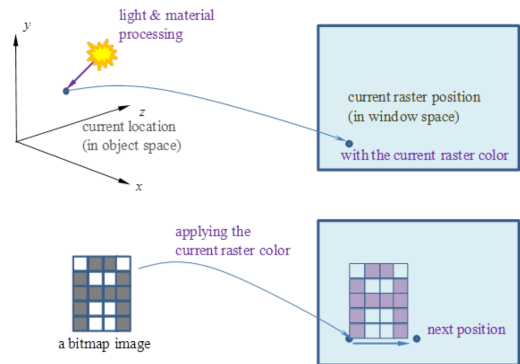
DrawPixels(int *width, height*, enum *format, type*, void* *data*) : 픽스맵 1개를 *data* 영역에 저장된, width height 크기를 가지는 2차원 색상 배열로 정의하고, 이 픽스맵을 *current raster position*에 출력한다. 색상 정보의 저장 형태는 *format*과 *type*에서 설정하는데, 주로 RGB (red, green, blue) 또는 RGBA (red, green, blue, alpha) 형태를 사용한다.

픽스맵의 처리에 있어서는 좌표의 설정 과정에서 기술적인 문제가 발생한다. RasterPos(...) 함수는 물체 좌표계(object coordinate system)에서 좌표를 설정하지만, 픽스맵의 출력 시에는 윈도우 좌표계(window coordinate system)에서의 2차원 좌표를 필요로 한다. 본 논문의 구현에서는 최종 위치를 계산해 내기 위해서, OpenGL의 좌표 변환 과정을 그대로 에뮬레이션했다. 즉, [그림 2](b)에서와 같이, modelview matrix, projection matrix, perspective division, viewport transformation 등의 연산을 각 단계별로 차례로 적용하여, 최종적인 윈도우 좌표를 구하였다.

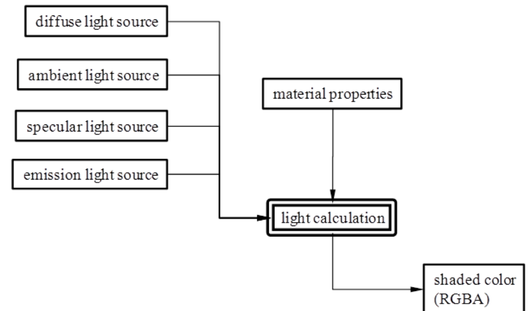
비트맵의 출력을 위해서는 전경색(foreground color)을 설정해야 하고, 0/1로 구성된 비트맵 정보를 보낼 필요가 있다. 다음 함수들이 그러한 기능을 수행한다.

Color(float *r, g, b, a*) : 비트맵은 색상이 출력되어야 할 부분을 비트 1로 표현하는데, 이 때, 실제 사용될 색상을 지정하는 함수이다. 2가지 모드의 설정이 가능한데, 단순한 경우는 해당 색상을 그대로 사용하지만, 그래픽스 시스템의 렌더링 부분이 L&M(light and material, 광원과 물성) 계산을 사용하도록 설정되어 있는 경우라면, 해당 색상을 가지는 점(point)이 3차원 상에서 *current raster position*에 주어졌을 때, 실제 3차원 렌더링 파이프라인에서 계산해 낸 최종 픽셀 색상을 사용하도록 한다. 이 색상은 *current raster color*로 저장된다.

Bitmap(int *width, height*, float *xorg, yorg*, float *xinc, yinc*, void* *data*) : 비트맵은 *width* × *height* 크기를 가지되, 각 색상 정보가 0 또는 1로 설정된 2차원 배열 형태로 *data* 영역에 저장된다. 이 기능은 문자(character font) 출력에 자주 사용되는데, 이를 위해, 실제 출력되는 배열을 (*xorg, yorg*) 픽셀만큼 shift 시켜서 사용하고, 출력이 끝나면, current raster position을 (*xinc, yinc*) 픽셀만큼 이동시켜서, 다음 글자의 출력에 대비하도록 한다. 또한, 색상 정보가 1인 부분은 current raster color로 출력하되, 0인 부분은 전혀 출력하지 않아서, 투명하게 처리되어야 한다.



(a) 비트맵의 출력



(b) L&M 처리

그림 3. 비트맵 이미지의 처리

비트맵 출력에서 발생하는 또다른 기술적 문제는 전경색(foreground color)의 계산이다. L&M 기능이 켜져 있는 경우에는 [그림 3]에서와 같이, 결국 광원과 재질에 대한 하드웨어 계산 과정을 에뮬레이션 해야 한다. 본 논문에서의 구현은 이 전체 과정을 소프트웨어로 에뮬레이션 해서, 동일한 결과를 구하도록 했다.

픽스맵과 비트맵의 연산을 구현하는 방법들 중에서, 가장 널리 알려진 것은 텍스처 매핑 기능을 사용해서

간접적으로 구현하는 방식이다. 이 방식은 [그림 4]에서와 같이, 출력하려는 픽스맵이나, 비트맵을 텍스처 이미지로 저장한 후에, 출력을 원하는 위치에 2차원 평면 직사각형을 출력시키면서, 텍스처 매핑을 적용하면, 저장된 이미지가 출력된다.

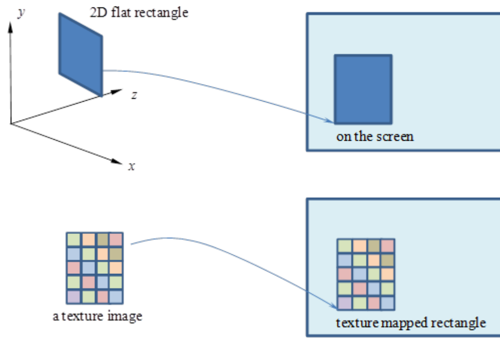


그림 4. 텍스처 매핑 방식의 처리

이러한 방식은 몇 가지 문제점을 가지는데, 이들은 다음에 설명하는 것과 같이, 텍스처 매핑 기능의 기술적 한계 때문에 발생한다.

- **텍스처의 크기에 제한이 있다:** 전형적인 모바일 그래픽스 파이프라인에서는 하드웨어의 성능 문제로, 텍스처의 크기가 반드시 2의 승수가 되도록 제한을 가한다. 일반적인 직사각형 형태를 모두 허용하려면, 스텐실 매스킹(stencil masking)과 같은, 좀더 고급의 그래픽스 기법도 함께 적용해야 한다.
- **텍스처 매핑은 많은 계산량을 필요로 한다:** 대부분의 모바일 그래픽스 시스템에서는 텍스처 매핑 하드웨어가 제한된 성능만을 가지는 경우가 많다. 따라서, 픽스맵, 비트맵의 출력과 같이, 상대적으로 단순한 작업에 텍스처 매핑 기법을 적용하면, 실제로 필요한 고급 기능들에 영향을 미쳐, 전체 성능의 급격한 저하를 가져올 수도 있다.
- **비트맵에서는 투명한 출력이 가능해야 한다:** 비트맵의 비트 0값은 투명한 픽셀에 대응된다. 이것을 텍스처 매핑에서 구현하려면, 대응되는 부분의 alpha 값이 0이 되는 완전히 새로운 텍스처 이미지 를 실시간으로 생성해 내거나, 스텐실 매스킹과 같은 고급 기능을 사용해서 처리해야 한다.

이러한 문제 때문에, 텍스처 매핑을 이용하여 간접적으로 이미지를 출력하는 방식은 알파-블렌딩(alpha blending)이나, 스텐실 매스킹과 같은 좀더 고급 기법들이 함께 사용되어야 한다. 이렇게 되면, 프로그래머는 다른 특수 효과에 이들 기법들을 사용하지 못하게 되는, 치명적인 제약을 받게 된다. 또한, 원활한 처리를 위해서, 픽스맵, 비트맵 이미지들의 크기를 2의 승수값이 되도록 재조정해야 한다는 제약 조건도 있다.

이것들을 해결하는 방법으로, 본 논문에서는 픽스맵, 비트맵에서 출력이 필요한 픽셀들에 대해서만 각 픽셀에 대응하는 점(point)을 출력하는 방식을 택하였다. 즉, [그림 5]에서와 같이, 주어진 2차원 이미지를 픽셀들의 집합으로 보고, 대응되는 윈도우 좌표 상에, DrawArrays(…)와 같은 3차원 그래픽스 출력요소(3D graphics primitive)를 사용해서, 점집합(point set)을 출력하게 했다.

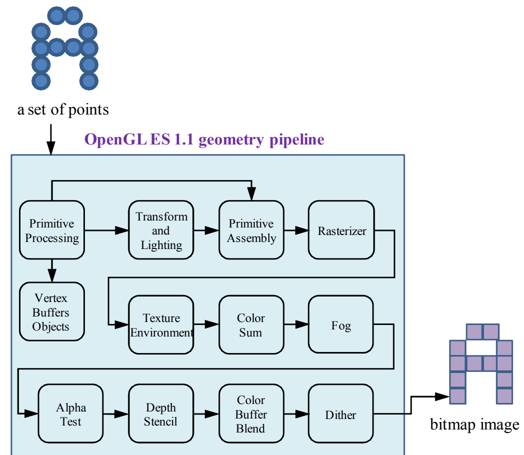


그림 5. 점집합 처리 방법

이러한 방식의 처리에서는 어떠한 크기를 가지는 이미지들도 처리가 가능하고, 더 이상 알파-브렌딩이나 스텐실 매스킹 등의 다른 처리 기법들이 추가로 요구되지 않는다. 또한, 반대로, 알파-브렌딩이나 스텐실 매스킹 등의 고급 기법들이 지원되지 않는, 저수준의 그래픽스 하드웨어에서도 본 논문이 제안하는 방법을 그대로 사용할 수 있다. 3차원 상의 점을 출력하는 기능만 필요하다.

본 논문이 제안하는 방법이 가지고 있는 잠재적인 문제점은 전송해야 하는 자료의 양이 늘어날 수 있다는 것이다. 이미지는 보통 픽셀의 색상값들만 전송하면 되지만, 본 논문이 제안하는 방법에서는 점의 3차원 좌표와 색상값들이 모두 필요하고, 그래픽 카드로 전송해야 하는 자료양이 증가한다. 하지만, 비트맵의 경우는 투명한 픽셀들의 존재로 인해, 실제 늘어나는 자료양은 그렇게 많지 않다.

이 절에서 제시한 처리과정에 대한 분석들에 기초해서, 다음 절에서는 구체적인 알고리즘들을 제시하겠다.

III. 알고리즘의 제시 및 구현

이 절에서는 래스터 파이프라인의 구현을 위한 알고리즘들이 설명된다. 설명의 편의를 위해, 픽스맵 처리 과정을 먼저 보이고, 다음으로 비트맵 처리 과정을 보이겠다.

1. 픽스맵의 처리

RasterPos(…) 함수에서는 주어진 물체 좌표(object coordinates)로부터 대응되는 윈도우 좌표(window coordinate)를 계산해야 한다. 본 논문에서는 OpenGL 좌표 변환 파이프라인을 소프트웨어로 그대로 구현해서, 윈도우 좌표를 정확히 계산하도록 했다. 다음 단계로, DrawPixels(…) 함수는 다음 단계들을 수행한다.

입력: RasterPos(…) 함수에서 미리 설정한 current raster location (x, y, z)

입력: RGBA 또는 RGB 색상값으로 저장된 픽스맵 정보

단계1: OpenGL 상태 변수들(state variables)을 저장하고, 기하 파이프라인을 끈다. 현재의 좌표 변환 설정을 나중에 그대로 복원할 수 있도록, OpenGL에서 좌표 변환 파이프라인에 관련된 모든 상태 변수들을 저장한다. 픽셀에 부여되는 색상 값이 그대로 저장될 수 있도록, 텍스처 매핑, 쉐이딩 기능 등을 끈다.

단계2: 점들의 집합을 출력한다. 모든 변환 행렬을 단위 행렬로 설정해서, 필요한 윈도우 좌표가 그대로 나오게 한다. 픽셀 좌표 (x, y, z)를 버텍스 어레이(vertex array)에 설정하고, RGBA 형태의 색상 값들을 컬러 어레이(color array)에 설정한 후, 출력 프리미티브(output primitive)로 이미지를 출력한다.

단계3: 원래의 기하 파이프라인을 복원한다. 저장해 놓았던 모든 상태 변수들을 복원한다. 좌표 변환, 텍스처 매핑, 쉐이딩 기능 등을 다시 켠다.

2. 비트맵의 처리

current raster color를 설정하기 위해서는 glColor4f(…) 등의 함수를 사용한다. 2절에서 설명한 바와 같이, 필요로 하는 색상 값을 얻기 위해서는 쉐이딩 기능 전체를 거쳐야 할 수도 있다. 우리는 OpenGL의 쉐이딩 기능 전체를 에뮬레이트 해서 정확한 계산을 하게 했다.

비트맵 처리 알고리즘은 다음과 같다.

입력: current raster location (x, y, z)와 색상값 (R, G, B, A)

입력: 각 픽셀마다 0 또는 1이 설정된 비트맵 정보

단계1: OpenGL 상태 변수들(state variables)을 저장하고, 기하 파이프라인을 끈다. 현재의 좌표 변환 설정을 나중에 그대로 복원할 수 있도록, OpenGL에서 좌표 변환 파이프라인에 관련된 모든 상태 변수들을 저장한다. 픽셀에 부여되는 색상 값이 그대로 저장될 수 있도록, 텍스처 매핑, 쉐이딩 기능 등을 끈다.

단계2: 전경색을 이용해서 점들의 집합을 출력한다. 투명한 픽셀들이 나올 수 있도록, 추가적인 작업을 해야 한다. current raster color를 전경색으로 설정한다. 비트 값이 1인 부분은 대응되는 좌표를 제공하고, 0인 부분은 무시하도록 출력 함수로 주어진 비트맵을 그린다.

단계3: 원래의 기하 파이프라인을 복원한다. 저장해 놓았던 모든 상태 변수들을 복원한다. 좌표 변환, 텍스처 매핑, 쉐이딩 기능 등을 다시 켠다.

단계4: 새로운 위치로 이동하는 추가 작업을 한다. 이제 current raster position을 (*xinc, yinc*) 만큼 이동시킨다.

비트맵의 처리는 주로 비트맵 폰트의 출력을 염두에 두고 설계되었으므로, current raster position을 다음 폰트 출력 위치로 이동시키도록 설정되어 있다. 위에서 설명한 알고리즘들은 우리 구현의 핵심적인 역할을 담당하였다. 그 구현 결과는 다음 절에 제시된다.

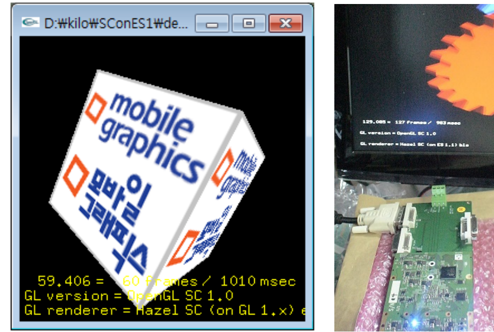
IV. 구현 결과

본 논문에서는 3절에서 소개한 알고리즘들에 기초해서 래스터 파이프라인 기능을 구현했다. 우리의 구현은 이미 설명한 바와 같이, 점들의 집합을 출력하는 방식으로 구현되었다. 따라서, 하드웨어 래스터 파이프라인의 지원이 전혀 필요하지 않고, 기하 파이프라인의 기능만을 필요로 한다.

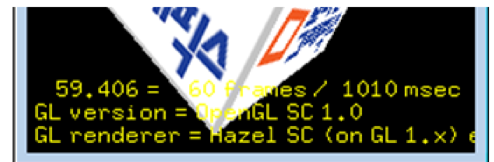
실제 구현은 현재 완전 소프트웨어 방식을 택하였다. 설계 초기부터 기하 파이프라인의 기능만을 사용하도록 했으므로, 하드웨어 래스터 기능이 없는, OpenGL ES 하드웨어들에서도 아무런 문제없이 작동 가능하다. 현재의 구현은 정확히는 OpenGL ES 1.1 하드웨어 상에서 소프트웨어 방식만으로도 작동 가능하다.

[그림 6]은 본 논문에서의 구현으로 출력한 비트맵 글자들이다. 여기에 나온 바와 같이, 이제 OpenGL ES 1.1 응용 프로그램들에서도 비트맵 글자와 픽스맵 방식의 이미지들을 출력할 수 있다.

구현의 정확성을 보이기 위해, 본 논문에서는 크로노스 그룹에서 제공하는 공식 CTS(conformance test suites)[13]의 일부를 사용했다. 본 논문의 구현은 이들 테스트를 모두 통과해서, 표준 문서에서 요구하는 대로, 정확히 구현되었음을 보였다. 수행 속도는 출력하려는 픽스맵, 비트맵의 수나 내용에 따라 차이가 나지만, [그림 6](b)의 예제 프로그램을 Freescale i.MX6 그래픽스 칩에서 테스트한 결과로는 평균적으로 초당 113.47 프레임을 기록하는 등, 현재까지의 실험 결과로는 문제가 될 정도의 속도 저하는 보이지 않았고, 실시간 응용에서 사용하기에 문제가 없었다.



(a) 비트맵 폰트의 출력 (b) 임베디드 보드의 실행 결과



(c) 확대된 화면

그림 6. 구현 결과

V. 결론

현재 사용하는 모바일 그래픽스 플랫폼들은 대부분 하드웨어 래스터 파이프라인을 제거했다. 반면에, 비트맵 폰트의 처리나, 이미지 프로세싱 응용에서는 여전히 래스터 출력 기능을 필요로 한다. 텍스처 매핑을 사용해서 래스터 기능을 에뮬레이트 할 수 있다고 알려져 있지만, 실제로는 알파-블렌딩이나 스텐실 처리와 같은, 추가 기능이 필요하다. 반면에, 어떤 응용 프로그램들은 래스터 처리와 동시에, 이러한 알파-블렌딩이나 스텐실 처리를 함께 요구하므로, 문제가 될 수 있다.

이런 문제점들을 피하기 위해, 본 논문에서는 래스터 연산을 구현하는 새로운 방법으로, 점들의 집합을 출력하는 방식을 제안했다. 이 방식에서는 픽스맵이나 비트맵 이미지마다 점들의 집합을 생성한 후, 전통적인 3차원 기하 파이프라인에서 이들 점들을 처리해서 화면에 표시하게 했다. 즉, 우리의 방식에서는 픽스맵이나 비트

맵 이미지를 색상값을 가지는 3차원 점들의 집합으로 해석해서, 최종적으로는 화면 상에 2차원 이미지를 출력하게 했다.

본 논문에서는 알고리즘의 세부 사항들과 구현 결과를 보였다. 현재까지의 결과로는 OpenGL ES 하드웨어와 같이, 래스터 처리 기능이 전혀 없고, 기하 파이프라인만 제공하는 플랫폼에서도, 소프트웨어 기능의 첨가만으로도 요구되는 래스터 연산들을 모두 제공한다. 이는 래스터 연산들을 활발히 사용하던 기존의 OpenGL 기반 응용 프로그램들이 별다른 수정 없이 OpenGL ES 하드웨어 상에서 사용 가능하다는 것을 의미한다. 개발 비용 측면에서는 새로운 모바일 그래픽스 응용들에서 기존의 데스크탑 기반 소프트웨어들이 별다른 수정 없이 그대로 사용 가능하므로, 소프트웨어 개발 비용을 줄이는 효과를 가져온다.

참 고 문 헌

[1] E. Angle, *Interactive Computer Graphics, A top-down approach with OpenGL*, 6th Ed., 2012.

[2] 최홍준, 김철홍, "범용 응용프로그램 실행 시 하드웨어 구성과 분기 처리 기법에 따른 GPU 성능 분석", 한국콘텐츠학회논문지, 제13권, 제3호, pp.9-21, 2013.

[3] 도주영, 백낙훈, "OpenGL을 이용한 Direct3D 기능의 구현", 한국콘텐츠학회논문지, 제11권, 제11호, pp.19-27, 2011.

[4] K. Pulli, "New APIs for mobile graphics," Proc. SPIE Multimedia on Mobile Devices II, 2006.

[5] D. Blythe, *OpenGL ES Common/Common-Lite Profile Specification*, Khronos Group, 2007.

[6] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification*, Version 2.1, Khronos Group, 2006(12).

[7] A. Munshi and J. Leech, *OpenGL ES Common Profile Specification*, Version 2.0.24, Khronos Group, 2009(4).

[8] B. Lipchak, *OpenGL ES*, Version 3.0.1, 2013(1).

[9] M. Kilgard, *The OpenGL Utility Toolkit (GLUT) Programming Interface API*, Version 3, Silicon Graphics, 1996.

[10] T. Olson, *OpenGL ES Extension No. 7: OES_draw_texture*, Khronos Group, 2004.

[11] N. Baek, "Implementation of the raster pipeline over the 3D geometry pipeline: A point-set approach," Proc. ICCE 2013, pp.15-16, 2013.

[12] N. Mansfield, *The Joy of X: The Architecture of the X Window System*, UIT Cambridge Ltd., 2010.

[13] http://www.khronos.org/opengles/sc/

저 자 소 개

백 낙 훈(Nakhoon Baek) 정회원

- 1992년 2월 : 한국과학기술원 전산학과(공학석사)
- 1997년 2월 : 한국과학기술원 전산학과(공학박사)
- 2004년 3월 ~ 현재 : 경북대학교 컴퓨터학부 교수

<관심분야> : 모바일 그래픽스, 리얼타임 그래픽스