# OpenGL을 통한

# 3차원 그래픽스 프로그래밍: 기초편

서강대학교 공과대학 컴퓨터학과 임 인 성

# 서 문

필자가 1980년대 중반 SGI의 IRIS GL과 HP의 Starbase 라이브러리를 사용하여 3차원 그래픽스 프로그래밍을 처음 시작할 무렵만 해도 대화식의 3차원 렌더링은 고가의 그래픽스 전용 컴퓨터에서나 가능했었다. 사실 지금의 장비들과 비교하면 그 당시의 SGI와 HP 등의 그래픽스 워크스테이션의 성능은 매우 미약했지만, 3차원 공간에서 기하 물체들을 자유자재로 움직일 수 있다는 사실이 상당히 신기하게 느껴졌었다. 그 후 시간이 감에 따라 컴퓨터 하드웨어의 성능이 급격하게 향상이 되었는데, 무엇보다도 3차원 그래픽스 가속기의 가격 대비 성능의 도약은 가히 놀랍다고 할 수 있다. 특히 점차 개인용 컴퓨터에도 고성능 그래픽스 가속기가 장착이 되는 추세여서, 3차원 컴퓨터 그래픽스 기술을 응용한 소프트웨어를 쉽게 사용할 수 있게 되었다.

3차원 게임, 가상 현실, 실시간 인터넷 등의 응용 분야에서 실시간 그래픽스 소프트웨어 기술의 무한한 상업성은 오래 전부터 널리 인식되어 왔으나, 고성능의 그래픽스 하드웨어를 사용해야 한다는 사실 때문에 많은 제약을 받아왔다. 그러나 하드웨어의 성능 향상 및 가격 하락으로 인하여, 이제는 일반 사용자들도 충분히 3차원 그래픽스 소프트웨어를 손쉽게 사용할 수 있는 장비를 갖추게 되었다. 이러한 추세에 맞춰 몇 년 전부터 실시간 그래픽스에 대한 연구 개발 및 그에 기반을 둔 상업용 소프트웨어 제작에 활발한 투자가 이뤄지고 있다.

이 책은 실시간 컴퓨터 그래픽스 중 3차원 렌더링에 관한 책이다. 특히 OpenGL이라는 그래픽스 라이브러리를 통하여 실시간적으로 이미지를

생성하는데 필요한 이론과 프로그래밍 기법을 익히는 것을 목표로 한다. 이 책의 집필 목적은 단순히 '또 하나의 OpenGL의 프로그래밍 가이드'를 쓰기 위한 것이 아니다. 약간은 거창하게 느껴질지는 모르나 다음과 같은 목적을 염두에 두고 집필을 시작했다.

- 실시간 렌더링에 관련된 3차원 그래픽스 이론을 익힌다.
- OpenGL을 통하여 실시간 렌더링 기법을 익힌다.
- OpenGL 시스템의 이해를 통하여 실시간 렌더링 시스템을 이해한다.

위의 세 가지 항목은 서로 밀접하게 관련이 되어 있어 어느 하나도 경시할 수가 없다. OpenGL과 같은 그래픽스 라이브러리의 이해가 없이 단순히 그래픽스 이론만 안다는 것은 컴퓨터 그래픽스와 같이 실용적인 응용 분야에서 별로 의미가 없다. 반면 3차원 컴퓨터 그래픽스 이론에 대한 정확한 이해가 없이 단순히 OpenGL 프로그래밍 기법만 익힌다면 결코 수준 높은 그래픽스 소프트웨어를 제작할 수가 없다. 또한 OpenGL 프로그래밍은 그에 기반이 되는 OpenGL 시스템을 정확하게 이해를 하여야만 최적의 실시간 렌더링 소프트웨어를 제작할 수가 있다. OpenGL 시스템은 현재 3차원 그래픽스 가속기에 의해 구현되는 실시간 렌더링 파이프라인의 가장 대표적인 시스템으로서, 이에 대한 이해는 바로 실시간 3차원 그래픽스 이론의 정확한 이해에 필수 불가결한 사항이 아닐 수가 없다.

이렇듯 서로 유기적으로 관련된 세 가지 목표를 달성하는데 필요한 내용을 한 권의 책에서 자세히 다룬다는 것은 불가능하기 때문에, 각 주제

에 대하여 가장 기본적인 사항들을 살펴보고, 그것들이 어떻게 유기적으로 관련이 되어 있는가를 이해하도록 하였다. 특히 본 기초편에서는 3차원 뷰잉, 조명 계산, 래스터화, 텍스춰 매핑 등 3차원 렌더링 파이프라인에서 가장 기초가 되는 네 가지 주제를 선택하여 구체적으로 설명하려 하였다. 이 책은 기본적인 C/C++ 프로그래밍 정도의 지식이 있는 사람은 누구나 읽을 수 있도록, 반복적인 설명을 통하여 쉽게 이해를 할 수 있도록 노력하였다. 다시 한번 강조를 하지만 단순히 함수의 이름을 외어 OpenGL 프로그래밍를 하는 수준이 아니라, 그에 필요한 이론과 실제를 정확하게 파악하고 최적화된 그래픽스 소프트웨어를 제작할 수 있도록 하는 것이 필자의 희망이다.

이 책을 쓰면서 제한된 시간 안에 한 권의 책을 집필한다는 것이 얼마나 어려운 작업인지를 깨달을 수가 있었다. 특히 컴퓨터 그래픽스 분야의 용어들이 완전하게 한글화가 되지 않은 상태에서 우리말 단어를 사용하는 것이 어려웠다. 가능한 한 정확하게 번역을 하려고 하였으나 일부 용어에 대해서는 논란의 여지가 있으리라 생각된다. 또한 제한된 지면에 담을 주제를 선택하는 것도 쉽지는 않았는데, 일단 현재의 내용으로 마무리를 하려 한다. 미쳐 다루지 못한 내용은 이 책의 속편이라 할 수 있는 응용편에서 설명하겠다. 많은 노력에도 불구하고 아직 내용이 약간 불충분하거나 적지 않은 오류가 있으리라 생각된다. 본 기초편도 계속해서 수정 및 보완을 하려하는바 독자 여러분의 많은 도움을 기대한다. 내용의 오류나 오타에 대한 지적, 그리고 그 외의 비평을 ihm@sogang.ac.kr로 보내 주시면 서강대학교 그래픽스 연구실의 홈페이지(http://grmanet.sogang.ac.kr)에 정리하여 올리도록 하겠다. 또한 이

책에서 사용한 예제 프로그램들과 다른 보조물을 같은 장소에 올려 많은 사람들이 사용할 수 있도록 하겠다.

보잘것없는 이 책의 출판을 허락해주신 그린 출판사의 윤덕우 사장님께 감사를 드린다. 이 책은 한글 LaTeX를 통하여 조판을 하였다. 수식이 많은 그래픽스 분야에서 LaTeX와 같은 도구의 도움이 없이 저술을 한다는 것은 매우 어려운 일이다. 이 자리를 빌어 LaTeX의 한글화에 노력을 기울이신 여러 분들께 감사드린다. 마지막으로 지난 1년간의 집필 기간 중 많은 격려와 꼼꼼한 교정을 통하여 값진 도움을 준 아내 경아와 텍스춰 이미지의 모델로 등장해준 딸 서영에게 고마움을 전한다.

2001년 1월 노고산 자락에서

임 인 성

# 목 차

# 제 1 장

# 들어가는 말

## 제 1 절 실시간 렌더링과 OpenGL

### 1.1 기하 모델링, 애니메이션, 그리고 렌더링

컴퓨터 그래픽스 분야에서 렌더링(rendering)이라는 단어는 넓은 의미로 사용이 되지만, 일반적으로는 가상의 3차원 세상으로부터 마치 카메라로 사진을 찍는 것과 같은 과정을 통하여 2차원 영상을 생성하는 작업으로 의미를 좁혀 사용을 한다. 이러한 작업을 좀 더 정확하게 표현하면 3차원 렌더링(3D rendering)이라 하는데, 이 책은 바로 3차원 렌더링을 위한 컴퓨터 그래픽스 이론과 프로그래밍 기법에 대하여 이해하는 것을 목표로 한다. 특히 3차원 렌더링을 구성하는 여러 분야 중 3차원 게임, 가상 현실, 실시간 인터넷 등과 같은 분야에서처럼 제한된 시간 안에 이미지를 생성해야하는 실시간 렌더링(real-time rendering)에 초점을 맞추려 한다.

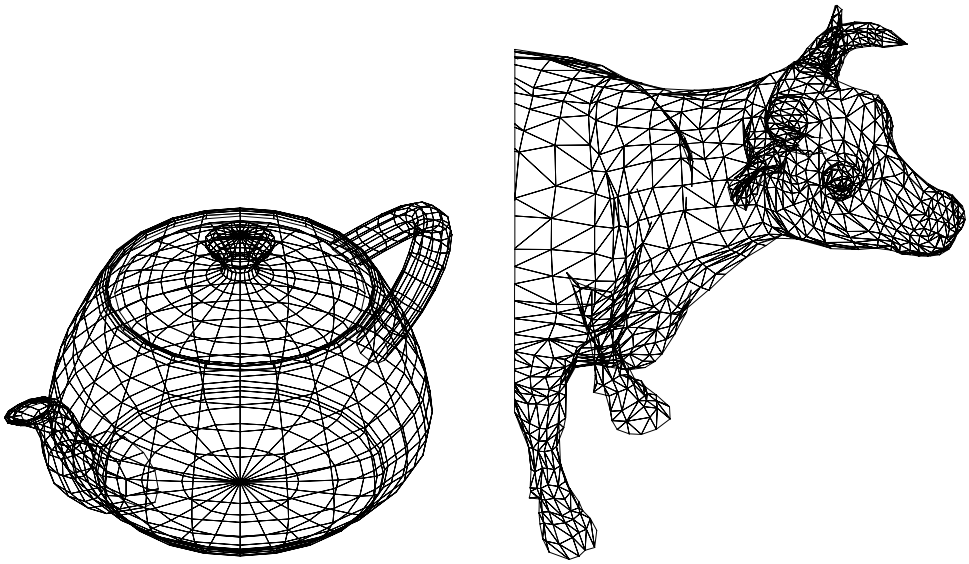
3차원 컴퓨터 그래픽스는 매우 다양한 분야에서 부가 가치가 높은 영상물을 제작하는데 있어 핵심적인 역할을 한다. 그 응용 분야는 일일이 나열할 수 없을 정도로 많은데, 아무래도 가장 먼저 떠오르는 것은 영화나 광고와 같은 영상물의 제작

일 것이다. 과연 3차원 컴퓨터 그래픽스 기법을 사용하여 그러한 영상물을 제작할 때의 이점은 무엇일까? 여러 가지를 생각할 수 있겠지만 무엇보다도 그러한 영상물을 제작하는 데 있어 실사 촬영을 하는 것보다 적은 비용으로 원하는 내용을 자유자재로 구성할 수 있다는 점일 것이다. 예를 들어 공룡이 나오는 영화를 제작하려면, 영화의 가장 중요한 요소인 공룡이 현존하지 않기 때문에 컴퓨터 그래픽스 기법의 도움을 받아야 한다. 또한 비행기가 날아가면서 폭발하는 장면을 촬영해야 할 때, 실제의 비행기를 사용하여 촬영한 내용이 마음에 안 들어 반복하여 촬영을 해야 한다면 이는 과도한 제작비 상승을 초래하게 될 것이다. 이와 같이 불가능하거나 비용이 많이 드는 작업을 3차원 컴퓨터 그래픽스에 기반한 도구를 사용하여 대체를 함으로써 문제를 해결할 수가 있다. 3차원 컴퓨터 그래픽스의 역사라 할 수 있는 최근 20-30년 동안 그래픽스 관련 소프트웨어와 하드웨어 기술이 비약적으로 발전을 해왔는데, 이제는 그 기술 수준이 '머리 속으로 상상을 할 수 있는 것은 무엇이든지' 영상물로 제작을 할 수 있는 경지에 도달했다고 해도 과언은 아닐 것이다.

렌더링과 함께 3차원 컴퓨터 그래픽스 분야의 기초를 이루는 두 가지 주제로 기하 모델링(geometric modeling)과 애니메이션(animation)을 들 수가 있다. 대부분의 그래픽스 작업은 직간접적으로 기하 모델링, 애니메이션, 그리고 렌더링 등의 작업을 거쳐 결과를 생성한다. 영화라는 예를 들어 이 세 가지 작업의 관계를 살펴보자. 영화를 제작하려면, 무엇보다도 촬영의 대상인 장면(scene)을 준비해야 한다. 즉 필요한 건물이나 소품들을 제작하고, 배우들을 분장시켜 적절한 위치에 배치를 해야 한다. 만약 컴퓨터 그래픽스 기술을 사용하여 이미지를 생성하려 한다면, 바로 이러한 작업도 컴퓨터를 통하여 수행하여야 한다. 다시 말해서 가상의 환경(virtual environment), 즉 가상의 세상(virtual world)을 제작해야 하는데, 이것은 장면에 존재하는 모든 대상을 컴퓨터가 처리하기 수월한 형태로 표현을 하고 저장을 하는 것

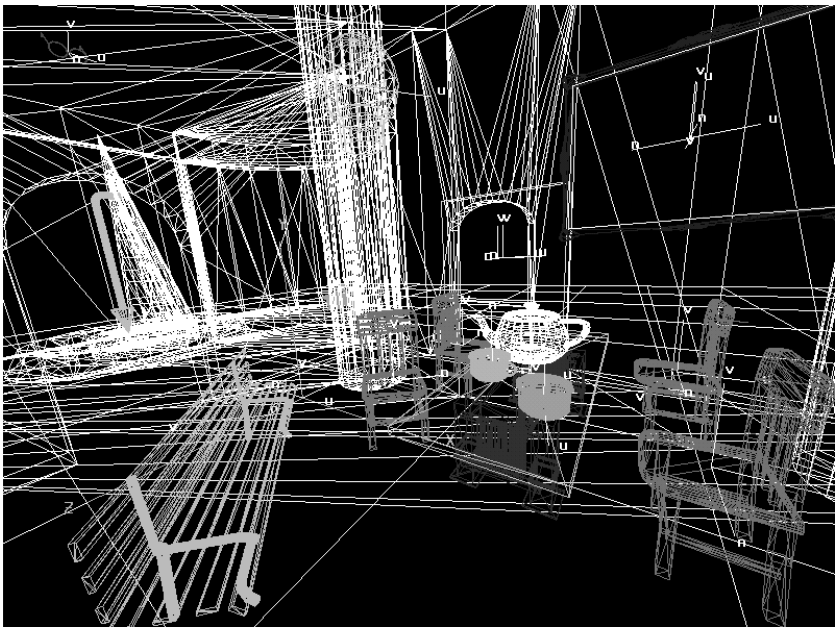
을 의미한다. 이러한 작업을 효과적으로 할 수 있도록 해주는 기술이 바로 기하 모델링인데, 기하 물체를 표현하는 수단은 여러 가지가 존재하나 컴퓨터 그래픽스에서 가장 널리 사용되는 방법은 점, 선분, 다각형 등으로 구성되는 다면체를 사용하여 물체를 표현하는 것이다. 특히 이 책의 주제인 실시간 렌더링 분야에서는 거의 대부분 다면체 모델(polygonal model)을 사용하여 장면을 표현한다. 그림 1.1은 다면체 모델을 사용하여 두 개의 물체를 표현한 모습과 가상의 세상을 구성한 예를 보여주고 있다.

항상 이러한 구분이 옳다고는 할 수 없지만, 어떻게 보면 기하 모델링은 물체의 정적인 기하 정보를 산출하는 과정이라 할 수 있다. 일단 물체를 제작하고 나면, 필요에 따라 동적인 데이터를 산출해야 하는데, 이는 넓게 봐서 애니메이션이라는 작업에 의해 이뤄진다고 할 수 있다. 예를 들어 영화에서 공룡이 뛰어가는 장면을 촬영한다고 생각해보자. 영화와 같은 동영상은 조금씩 변하는 이미지를 1초에 24-30장 정도의 빠른 속도로 보여줌으로써, 물체가 자연스럽게 움직이는 것과 같은 느낌을 가지게 해준다. 따라서 공룡이 뛰어가는 장면을 생성한다는 것은 결국 동영상을 구성하는 각 이미지 프레임(frame)을 제작하는 것이 되는데, 이 경우 연속된 프레임 안에서는 공룡의 위치와 모습이 조금씩 변하게 될 것이다. 문제는 어떠한 방식으로 각 프레임에 대한 공룡의 모습을 설정해야만 마치 살아 있는 공룡이 뛰어가는 듯한 효과를 낼 수 있을 것인가 하는 것이다. 이를 위해서 뛰어가는 모습뿐만 아니라, 공룡의 표정이나 근육의 움직임 등 여러 가지 사항을 자연스럽게 처리를 해주어야 한다. 사실 이는 매우 어려운 작업으로서 애니메이션 분야에서 물리학, 역학, 수학 등의 다양한 이론을 통하여 사실적인 애니메이션 기법을 개발해왔으며, 이와 함께 실시간 렌더링과 같이 사실성은 약간 떨어지나 빠른 시간 안에 움직임을 생성해내는 실시간 애니메이션 기법에 대해서도 연구를 해왔다.



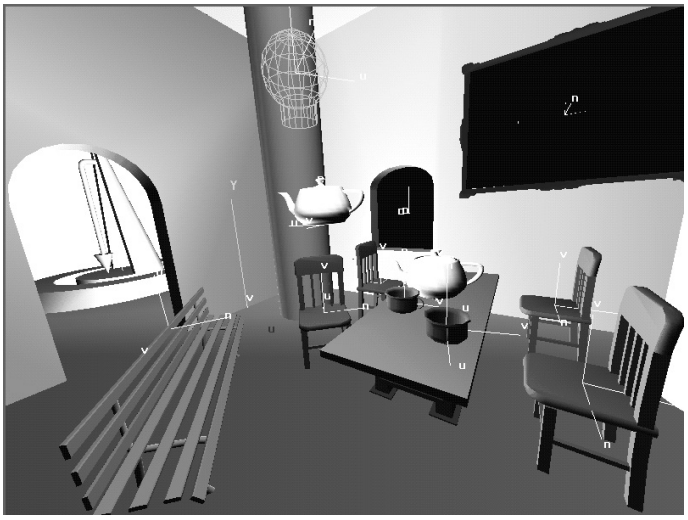
(a) 물주전자에 대한 다면체 모델

(b) 소에 대한 다면체 모델



(c) 다면체 모델로 구축한 가상의 세상

그림 1.1: 다면체 모델을 사용한 기하 모델링



(a) 렌더링 예 1

(b) 렌더링 예 2

그림 1.2: 가상의 세상에 대한 렌더링의 예

일단 다면체 모델과 같은 표현 수단으로 세상을 구성하고 각 프레임에 대한 물체의 움직임을 결정하고 나면, 최종적으로 렌더링 작업을 통하여 이미지를 생성하게 된다. 다시 한번 강조를 하면, 렌더링이란 피사체를 배치하고 적절한 조명을 설치한 후, 카메라를 원하는 위치에 가져가 적절한 구도를 잡아 셔터를 눌러 사진을 촬영하는 과정을 컴퓨터 그래픽스 기법을 사용하여 수행하는 모든 과정을 뜻한다. 효과적인 렌더링을 위하여 다양한 방법이 개발되어 왔는데, 그림 1.2는 그림 1.1(c)의 가상의 세상에 대하여 서로 다른 두 개의 렌더링 기법을 사용하여 이미지를 생성한 예를 보여준다.

지금까지 3차원 컴퓨터 그래픽스를 구성하는 세 가지의 기본 요소에 대하여 개략적으로 살펴보았다. 이 책은 렌더링, 특히 실시간 렌더링에 대하여 이해를 하는 것이 주목적이기 때문에 기하 모델링이나 애니메이션에 대한 주제는 다루지 않도록 하겠다. 여기서는 다면체 모델로 표현된 가상의 3차원 세상으로부터 빠른 시간 안에 원하는 내용의 이미지를 생성하는데 필요한 이론과 실제에 대하여 구체적으로 알아보겠다.

## 1.2 실시간 렌더링

렌더링의 주 목적은 이미지를 생성하는 것이라 했는데, 과연 컴퓨터 그래픽스 렌더링이 추구하는 궁극적인 목표는 무엇일까? 이는 실제로 카메라를 사용하여 촬영하여 얻은 이미지와 구별을 할 수 없을 정도로 정교하고 현실감 있는, 사진과 동일한(photorealistic) 이미지를 생성하는 것이다. 어떻게 보면 렌더링은 컴퓨터 그래픽스의 문제라기보다는 물리학의 문제라 할 수 있다. 카메라를 통한 사진 촬영의 과정을 생각해보면, 태양이나 조명과 같은 광원에서 출발한 빛이 복잡한 경로를 통하여 물체로 들어와서 반사가 된 후, 카메라의 렌즈를 통과하여 필름을 감광시키게 된

다. 따라서 물리학적인 대상인 빛이 광원에서 출발하여 카메라를 향해 날아오는 과정을 설명할 수 있는 물리학적인 모델을 설정하고, 이를 통하여 원하는 이미지 정보를 추출하여야 한다.

그러나 물리학적으로 정확한 이미지를 생성한다는 것은 너무나도 복잡하고 많은 계산 시간을 요하기 때문에, 컴퓨터 그래픽스 분야에서는 원래의 물리학적인 모델에서 실행 가능한 모델을 유추하여 이미지 제작에 사용을 한다. 다시 말해서 주어진 시간 안에 계산을 할 수 있도록, 단순화한 렌더링 모델을 설정하고 이를 효과적인 알고리즘으로 구현을 하여 사용을 한다. 문제는 과연 어느 정도로 단순화를 시킬 것인가 인데, 여기에는 두 가지 측면을 고려해야 한다.

하나는 렌더링 모델을 사용하여 생성한 이미지가 얼마나 사실적인가 하는 것이고, 다른 하나는 이미지를 얼마나 빨리 생성을 할 수 있을 것인가하는 것이다. 사실 이 두 가지 측면은 직접적으로 연관이 되어 있다. 좀 더 사실적인 이미지를 원하면 정확한 물리학적인 모델에 더 가까운 렌더링 모델을 사용해야 하고, 그럴 경우 더 많은 계산 시간을 필요로 하게 된다. 반면 아주 짧은 시간 안에 이미지를 만들어야 한다면, 결국 원래의 모델을 많이 단순화시켜야 하기 때문에 이미지의 사실성 저하는 피할 수가 없다.

영화를 제작한다면 사실성이 높은 이미지를 원하게 될 것이다. 예를 들어 어느 정도의 사실성을 내포한 이미지 한 장을 만드는데, 한 대의 컴퓨터에서 한 시간이 걸린다고 가정하자. 어떻게 보면 정교한 이미지 한 장에 한 시간은 별로 긴 시간이 아닌 것으로 생각할 수 있으나, 만약 1초당 24장의 프레임으로 구성된 75분짜리 영화를 제작을 한다면, 렌더링 계산 시간만 12년이 넘는 시간이 걸린다는 점을 고려하면, 이미지의 사실성과 렌더링 시간의 선택에 있어 매우 신중해야 한다.

컴퓨터 그래픽스 분야에서는 다양한 부류의 렌더링 알고리즘들이 개발이 되어

왔다. 광선 추적법(ray tracing)이나 래디오시티(radiosity) 기법과 같은 방법들은 비교적 적지 않은 계산 시간을 필요로 하는 반면 상당히 정교한 이미지를 생성해준다. 따라서 영화나 광고 등과 같이 사실적인 이미지가 필요한 분야에서 이 방법들이 많이 쓰이고 있다.

반면에 3차원 게임, 가상 현실, 실시간 인터넷 등과 같은 분야에서는 사람과 컴퓨터의 원활한 상호 작용을 위해서 '매우 짧은 시간 안에' 이미지를 생성해주어야 한다. 다시 말해서 '실시간(real-time)적으로' 렌더링 작업을 수행해야 하는 것이다. 여기서 과연 컴퓨터 그래픽스 분야에서의 실시간이란 얼마나 짧은 시간을 의미할까? 직관적으로 말하면 사람이 마우스나 조이 스틱과 같은 입력 장치를 움직일 때, 그와 동시에 화면의 이미지가 의도한 대로 변하는 것과 같이 느끼게 해줄 만큼의 빠른 시간을 의미한다. 보통 컴퓨터 그래픽스 분야에서는 1초에 24장에서 30장 정도의 이미지를 생성할 수 있으면, 다시 말해서 $\frac{1}{30}$초에서 $\frac{1}{24}$초 안의 시간 안에 이미지를 만들 수 있으면, 실시간적으로 이미지를 생성한다고 한다[1].

위에서 예시한 것과 같은 실시간 그래픽스 응용 분야에서는 사람과 컴퓨터간의 상호 작용 속도가 핵심적인 요소이기 때문에, 실시간 렌더링 기술의 적용은 필수적이라 할 수 있다. 한 가지 문제는 3차원 렌더링 알고리즘은 기본적으로 적지 않은 양의 계산을 요구한다는 사실이다. 따라서 과거에는 실시간이라는 목표를 달성하기 위하여 매우 단순화된 렌더링 모델을 사용했기 때문에, 생성한 이미지의 사실성이 그리 높지 않았던 것이 사실이다. 그러나 최근의 컴퓨터의 성능의 급격한 향상과 그래픽스 가속기를 비롯한 컴퓨터 그래픽스 기술의 발전에 힘입어 $\frac{1}{30}$초라는 시간 안에 수행할 수 있는 렌더링 작업의 양이 빠른 속도로 증가하고 있고, 그 결과

[1] 참고로 1초에 5장에서 10장 정도의 이미지를 생성할 때, 대화식(interactive-time)의 속도로 이미지를 생성한다고 한다.

과거와 비교하여 상당히 사실적인 영상을 실시간적으로 생성을 할 수 있게 되었다.

오래 전부터 3차원 컴퓨터 그래픽스 기술의 중요성, 특히 3차원 게임을 비롯한 실시간 렌더링 기술의 응용 분야의 상업성에 대하여 많은 사람들이 인지를 해왔으나, 컴퓨터 하드웨어의 낮은 성능과 높은 가격으로 인하여 대중적인 실시간 그래픽스 응용 소프트웨어를 제작하기가 어려웠다. 하지만 최근의 기술 발전에 힘입어 일반인이 보편적으로 사용하는 개인용 컴퓨터에 비교적 저가의 그래픽스 가속기만 장착해도, 상당히 사실성이 있는 실시간 그래픽스 응용 소프트웨어를 사용할 수 있게 되었다. 그 결과 최근 몇 년 전부터 렌더링을 비롯한 여러 컴퓨터 그래픽스 분야에서 실시간 응용 기술에 대하여 활발한 연구가 진행이 되어 왔고, 이를 토대로 하여 많은 실시간 응용 소프트웨어의 개발이 진행이 되고 있는데, 이러한 추세는 앞으로 더 빠르게 가속화가 되리라 예상된다.

## 1.3 OpenGL 시스템

OpenGL(Open Graphics Library)은 대화식 3차원 그래픽스 렌더링을 위한 응용 프로그래밍 인터페이스(application programming interface, API)로서, 원래 SGI의 그래픽스 워크스테이션에서 사용하기 위하여 1980년대 초반에 개발이 된 IRIS GL을 모태로 한다. OpenGL은 하나의 그래픽스 라이브러리로서 3차원 실시간 렌더링 프로그램의 작성에 필요한 함수들을 제공함으로써, 프로그래머들이 대화식의 그래픽스 응용 소프트웨어를 쉽게 제작할 수 있도록 해준다.

기존에는 SGI의 IRIS GL, HP의 Starbase, 그리고 PHIGS 등 서로 다른 그래픽스 라이브러리들을 사용하여 그래픽스 소프트웨어를 제작하였기 때문에 호환성에 상당한 문제가 있어왔다. 1990년대 초반 SGI는 IRIS GL을 3차원 그래픽스 프로그래밍 인터페이스의 표준으로 만들기 위하여, IRIS GL의 불필요한 기능을 제거하고

필요한 기능을 더욱 확장하여 OpenGL을 개발하였다. 이후 SGI, IBM, MS 등 유수한 컴퓨터 회사들로 구성된 OpenGL 아키텍춰 검토 위원회(OpenGL Architectural Review Board, OpenGL ARB)가 결성이 되어, OpenGL 시스템 전반에 대한 검토와 추세에 맞는 새로운 기능의 추가를 통하여 OpenGL 표준을 지속적으로 발전시켜왔다. 1992년 OpenGL 버전 1.0이 발표된 후 계속적인 발전을 통하여 1996년 버전 1.1이 발표되었고, 그 후 1998년 버전 1.2에 대한 규약이 발표되었다.

OpenGL은 현재 실시간 렌더링에 관련된 그래픽스 응용 소프트웨어의 제작에 있어 가장 대표적인 그래픽스 API로서, 그래픽스 분야의 소프트웨어 및 하드웨어의 기술의 발전과 함께 계속해서 발전을 해나가고 있다. 정확하게 말하면 OpenGL은 3차원 실시간 렌더링을 위한 하나의 추상적인 렌더링 시스템으로서, 효과적인 이미지의 생성을 위한 계산 과정인 렌더링 파이프라인(rendering pipeline)에 대한 아키텍춰를 정의하고, 이를 구동할 수 있는 함수들을 구현하여 프로그래머들이 API 형태로 사용할 수 있도록 해준다. OpenGL을 통하여 이미지를 생성하려면, 우선 이미지의 내용을 계산하는데 필요한 렌더링 인자들을 적절한 상태 값으로 설정해주어야 한다. 실제로 대부분의 OpenGL 함수는 관련 렌더링 인자의 값을 설정하는, 즉 렌더링 인자의 상태를 지정하는데 사용이 된다. 이렇게 필요한 인자들을 값을 적절하게 설정을 한 후, 렌더링 파이프라인으로 그리고자 하는 기하 데이터를 흘려보내면, OpenGL 시스템이 설정된 렌더링 인자들을 사용하여 이미지를 생성하게 된다.

따라서 OpenGL을 사용하여 렌더링을 하려면, 무엇보다도 원하는 이미지를 생성할 수 있도록 OpenGL 렌더링 파이프라인의 인자 값을 적절하게 설정을 해주어야 하기 때문에, 이러한 관점에서 OpenGL 시스템을 상태 기계(state machine)라 부른다. 바로 이 책에서는 가장 대표적인 실시간 렌더링 툴인 OpenGL 시스템의 아키텍춰, 즉 렌더링 파이프라인의 계산 과정을 정확하게 이해하고, OpenGL 함수들을

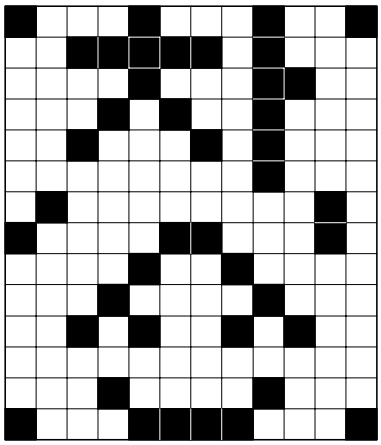


그림 1.3: 래스터 이미지

통하여 렌더링 파이프라인을 정확하게 구동하는 방법을 배움으로써, 3차원 실시간 렌더링에 대한 이해를 높이려 한다.

## 제 2 절 래스터 그래픽스 시스템에 대한 소개

### 2.1 래스터 이미지

컴퓨터 그래픽스 분야의 궁극적인 목표는 원하는 내용의 이미지를 생성하는 것이기 때문에, 이미지의 표현 방식은 그래픽스 작업에 큰 영향을 미친다고 할 수 있다. 최근에 가장 보편적으로 사용되는 방법 중의 하나가 래스터 이미지(raster image)의 형태로 이미지를 표현하는 것이다. 이 방법에서는 직사각형 형태의 이미지 영역을 화소(pixel)라 부르는 조그마한 영역으로 나누어[2], 각 화소를 해당 지역을 대표하는 색깔로 칠하게 된다.

화소는 래스터 이미지를 구성하는 기본 요소로서, 화소는 한 순간에 한 가지의 색깔을 가진다. 그림 1.3은 가로 12개, 세로 14개의 화소로 구성된 래스터 이미지를

[2] pixel이라는 용어는 'picture element'라는 어구를 줄인 것으로서, 이 책에서는 이 용어에 대하여 화소와 픽셀이라는 단어를 혼용하도록 하겠다.

보여주고 있는데, 보통 설명의 편의상 화소는 조그마한 사각형을 표시를 한다. 이 이미지의 경우 전체 직사각형 영역을 작은 개수의 화소로 나누었기 때문에, 화소가 커 보이나 실제로는 이미지를 세밀하게 나누기 때문에 보통 눈으로 화소를 구별하기는 어렵다. 한 이미지가 얼마나 많은 수의 화소로 표현되었는지를 나타내는 것이 해상도(resolution)이다. 이 그림에 도시된 래스터 이미지의 해상도를 $12 \times 14$와 같이 나타내는데, 이는 이미지를 가로와 세로로 각각 몇 개의 화소로 나누는지를 나타낸다. 아마 컴퓨터에 조금만 관심이 있다면, $640 \times 480$, $1024 \times 768$, $1280 \times 1024$와 같은 해상도를 한두 번쯤 들어보았을 것이다.

해상도는 래스터 이미지가 원하는 내용의 영상을 얼마나 정확하게 표현할 수 있는 가를 나타내는 척도로서, 물론 같은 크기의 이미지는 해상도가 높을 수록 원래의 영상 정보를 더 충실하게 나타낼 수 있다. 그림 1.4는 동일한 이미지에 대하여 서로 다른 해상도를 사용하는 세 개의 래스터 이미지를 보여주고 있는데, 해상도가 낮을수록 이미지의 내용이 '각이 져서 보이는' 현상이 심해진다. 해상도를 높이면 낮은 해상도를 가지는 이미지에서 표현할 수 없었던 세밀한 내용을 충실하게 표현할 수 있게 되나, 여기서 명심해야 할 것은 주어진 크기의 이미지에 대하여 해상도를 아무리 높여도 항상 정확하게 나타낼 수 없는 작은 크기의 내용이 존재한다는 사실이다.

사실 우리가 원래 표현하고자 하는 이미지는 래스터 이미지처럼 유한 개의 영역으로 구분이 되는 이산적인 정보가 아니고, 연속적인 형태의 정보이다. 따라서 무한개의 영역으로 이루어진 이미지 정보를 화소라 하는 유한 개의 '그릇'으로 담으려 하기 때문에 대부분의 경우 래스터 이미지는 원래의 이미지 정보를 정확하게 표현할 수 없다. 물론 해상도를 높이면 좀 더 정확해지기는 하겠지만, 이는 곧 이해하겠지만 비용의 증가를 초래하기 때문에 그러한 방법에는 한계가 있다. 이 문제에

(a) 256 × 256 (b) 128 × 128 (c) 64 × 64

그림 1.4: 래스터 이미지의 해상도

대해서는 1.2절에서 다시 한번 설명을 하겠다.

## 2.2 RGB 색깔 모델과 컴퓨터 색깔의 표현

요약을 하면 래스터 이미지는 유한 개의 화소 영역으로 구성되고, 각 화소는 자신의 영역을 나타내는 색깔로 표현이 된다. 그러면 과연 색깔(color)이란 어떻게 표현이 될까? 색깔은 물리학적인 대상인 빛과 밀접한 관계가 있는 것으로서, 가능한 한 빠르게 OpenGL 프로그래밍을 시작할 수 있도록 이에 대한 자세한 설명은 생략을 하려한다. 단 한 가지 반드시 알아야 할 것은 컴퓨터 그래픽스 분야에서 기본적으로 사용을 하는 RGB 색깔 모델(RGB color model)이다. 여기서 R은 빨간색(red), G는 초록색(green), B는 파란색(blue)을 나타내는데, RGB 색깔 모델에서는 바로 이러한 세 가지 원색을 더해서 원하는 색깔을 만들어 낸다. 각 원색의 상대적인 세기에 따라 생성되는 색깔이 서로 다른데, 보통 이 모델에 기반하는 색깔을 $(r, g, b)$ $(0 \leq r, g, b \leq 1)$와 같이 각 원소가 0과 1 사이의 값을 가지는 벡터로 나타낸다. 여기서 각 원소 값은 사용하는 해당 원색의 농도를 나타내는데, 0은 해당 색깔을 전혀 사용하지 않는다는 것을 의미하고, 1은 최대의 농도, 즉 순수한 원색을 더한다는 것

을 뜻한다.

RGB 색깔 모델은 아무런 빛이 없는 상태인 검은색의 상태에서 각 원색을 해당 농도만큼씩의 세기로 더해 원하는 색깔을 혼합해낸다. 따라서 색깔 (0, 0, 0)은 검은색(black, B)을 의미하고, (1, 1, 1)은 삼원색을 모두 더한 색깔이므로 흰색(white, W)이 된다. 또한 (1, 0, 0), (0, 1, 0), (0, 0, 1)은 각각 순수한 빨간색, 초록색, 파란색임을 쉽게 알 수가 있다. 한편 (0, 1, 1), (1, 0, 1), (1, 1, 0)은 흰색에서 각각 빨간색, 초록색, 파란색만 뽑아낸 색깔로 생각할 수가 있는데, 이 색깔들은 각각 시안색(cyan, C), 자홍색(magenta, M), 노란색(yellow, Y)이다. 한편 $r = g = b$가 같을 경우에는 무채색이 되는데, 이 값들이 0에서 1로 변함에 따라 검은색에서 출발하여 어두운 회색(grey)에서 점차 밝은 회색을 거쳐 흰색으로 가는 무채색의 띠가 형성이 된다. 지금까지 살펴본 빨간색, 초록색, 파란색, 시안색, 자홍색, 노란색, 흰색, 검은색, 회색이 가장 기본이 되는 색깔이라고 할 수가 있다.

이외에도 우리가 인식하는 대부분의 색깔이 RGB 색깔 모델을 사용하여 표현을 할 수 있다. 한 예로, (0.196, 0.6, 0.8)과 (0.435, 0.259, 0.259)는 각각 하늘색과 연어색깔을 나타내는데, 사실 대부분의 경우 RGB 색깔 모델로 표현된 임의의 벡터로부터 정확한 색깔을 상상하는 것은 쉽지는 않다. 따라서 직관적으로 색깔을 선택하고 다루어야 하는 상황에서는 이 색깔 모델은 적합하지 않지만, RGB 색깔 모델은 컴퓨터로 처리하기에 매우 효과적이기 때문에 그래픽스 분야에서 가장 기본이 되는 모델로 사용이 된다.

색깔도 다른 정보들과 같이 유한개의 비트를 사용하여 컴퓨터의 색깔로 표현하여 사용을 한다. 가장 간단한 래스터 이미지는 흑백 이미지로서 이 경우에는 화소당 한 개의 비트를 할당하면 된다. 즉 검은색을 0으로, 그리고 흰색을 1로 나타내면, 그림 1.3의 이미지의 첫째 줄은 011101110110과 같은 비트 스트림으로 표현이

된다. 칼라 이미지를 표현하려면 화소 당 더 많은 수의 비트를 할당해야 한다. 예를 들어 하나의 색깔을 나타내기 위하여 24개의 비트를 사용한다고 하면, 보통 R, G, B 각 채널에 8 비트를 할당한다. 이 경우 0과 1 사이의 실수로 표현된 각 채널 값을 $256(=2^8)$ 단계의 컴퓨터 숫자로 변환하여 색깔을 표현하게 된다. 다시 말해서 주어진 채널 값에 대하여 $\frac{0}{255}$, $\frac{1}{255}$, $\frac{2}{255}$, $\cdots$, $\frac{255}{255}$ 중 가장 가까운 값을 선택하여 255배를 한 값으로 표현을 한다. 따라서 24 비트 색깔을 사용하면, 빨간색은 (255, 0, 0), 그리고 하늘색은 (50, 153, 204)와 같이 표현된다. 하나의 색깔을 나타내기 위하여 더 많은 비트를 사용할 경우 더 상세하게 색깔을 표현할 수가 있지만, 그에 따른 메모리 비용이 증가하게 된다.

24 비트를 사용하면 약 1670만$(=2^{24})$ 개의 색깔을 표현할 수 있는데, 대부분의 경우 이 정도면 충분하다고 할 수 있다. 하지만 상황에 따라 더 높은 정밀도가 요구되는 경우, 예를 들어 고급 렌더링 시스템에서 반복되는 색깔에 대한 계산을 통해 이미지를 생성할 때, 채널 당 12 비트를 사용하기도 한다. 어떠한 해상도를 사용하건 래스터 이미지는 결국 0과 1로 구성된 비트들의 집합으로 표현이 되는데, 이러한 래스터 이미지 데이터를 픽스맵(pixmap)이라 부른다. 픽스맵은 또한 비트맵(bitmap)이라 하기도 하는데, 이 용어는 그림 1.3과 같은 흑백 이미지의 픽스맵에 한정하여 쓰이기도 한다.

요약을 해보면, 래스터 이미지를 표현하기 위해서는 두 가지의 해상도를 고려해야 한다. 하나는 이미지를 구성하는 화소의 개수를 결정하는 해상도이고, 다른 하나는 각 화소에 대하여 얼마나 많은 수의 색깔로 구성된 팔레트(palette)에서 원하는 색깔을 선택할 수 있는지를 결정하는 해상도이다. 이 두 가지의 해상도에 따라 하나의 래스터 이미지를 저장하는데 필요한 메모리의 양이 결정이 된다. 예를 들어 $1280 \times 1024$의 해상도를 가지는 래스터 이미지에 대하여 24 비트 색깔을 사용하면

3.75MB($= 1280 \cdot 1024 \cdot \frac{24}{8}\text{B} = 3,932,160\text{B}$)의 메모리가 필요하게 된다.

지금까지 설명한 바와 같이 각 화소에 대한 색깔을 직접 RGB 색깔로 표현하는 모드를 OpenGL에서는 RGB 모드(RGB mode)라 한다[3]. 이 모드를 사용하면 래스터 이미지를 간결하게 나타낼 수가 있는 반면, 두 해상도가 높아질 수록 필요한 메모리의 크기가 빠르게 증가함을 알 수가 있다.

사실 요즈음은 메모리가 그리 비싼 자원은 아니지만, 아직도 해상도가 높은 이미지의 경우 그 크기가 부담이 되기도 한다. 특히 메모리가 부족했던 과거에는 RGB 모드로 컴퓨터 색깔을 표현하는 것이 상당한 부담이 되고는 했다. 따라서 사용하는 메모리의 크기를 줄이기 위하여 색깔 인덱스 모드(color-index mode)라는 방식을 사용하여 색깔을 표현하고는 했다. 이 방법에서는 각 화소에 칠할 $(r, g, b)$ 값을 직접 표현을 하는 것이 아니라, 전체 이미지를 나타내는데 필요한 색깔들을 모두 모아 팔레트를 구성하여 색깔 참조 테이블(color lookup table)이라 하는 메모리 영역에 저장을 해놓고, 각 화소에 대하여 해당하는 색깔이 있는 색깔 참조 테이블의 인덱스를 저장을 한다.

그림 1.5에는 색깔 인덱스 모드를 사용하여 래스터 이미지를 표현하는 예가 주어져 있다. 여기서는 인덱스를 저장하기 위해서 한 화소 당 8 비트를 할당을 하고 있다. 이 경우 화소 당 256($= 2^8$)개의 서로 다른 색깔 인덱스를 저장할 수 있으므로, 색깔 참조 테이블은 256개의 색깔로 구성이 되는데, 색깔의 해상도를 높이기 위하여 색깔 당 24비트를 할당을 하고 있다. 색깔 인덱스 모드의 원리는 간단한데, 그림에서 표시된 것과 같이 화소에 할당된 값이 129($= 10000001$)라면, 이것을 인덱스로 하여 색깔 참조 테이블을 액세스한 값 (50, 153, 204)가 실제로 이 화소에 칠할 색깔

[3] 실제로 OpenGL에서는 뒤에서 설명하겠지만, R, G, B 채널 외에 A라는 채널을 하나 더 사용하기 때문에 RGBA 모드(RGBA mode)라 한다. 일반적으로 채널당 8 비트 이상을 사용할 때 full color mode 또는 true color mode라 부르기도 한다.

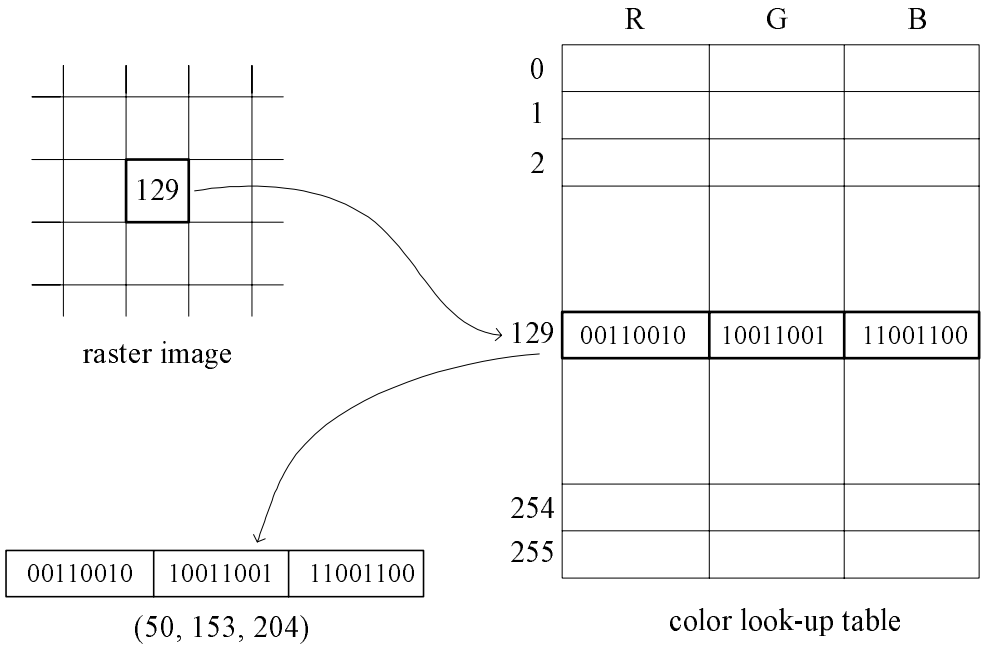



그림 1.5: 색깔 인덱스 모드

값이 된다.

색깔 인덱스 모드는 RGB 모드와 비교할 때 적은 양의 메모리를 사용하여 어느 정도 효과적으로 래스터 이미지의 내용을 표현할 수가 있다. 이 모드의 색깔 해상도는 각 화소 당 할당을 하는 비트 수와, 색깔 참조 테이블의 색깔을 표현하는데 사용되는 비트 수에 의하여 결정이 된다. 전자를 $n$이라 하고, 후자를 $m$이라 할 경우 한 순간에 래스터 이미지에 나타날 수 있는 서로 다른 색깔의 개수는 최대 $2^n$개임을 알 수 있다. 반면에 색깔 참조 테이블의 내용은 마음대로 구성을 할 수가 있는데, 이는 $2^m$개의 색깔로 구성되는 팔레트로부터 래스터 이미지를 칠할 $2^n$개의 색깔을 선택하는 작업이라 할 수 있다. 색깔 인덱스 모드는 아직도 유용하게 쓰이고는 있으나, 컴퓨터 메모리 가격의 하락으로 인하여 점차 RGB 모드에 의하여 밀려나고 있다. OpenGL에서는 RGB 모드와 색깔 인덱스 모드 등 두 가지 모두 지원을 하는데, 이 책에서는 색깔 인덱스 모드는 고려를 하지 않고 RGB 모드를 기본 색깔

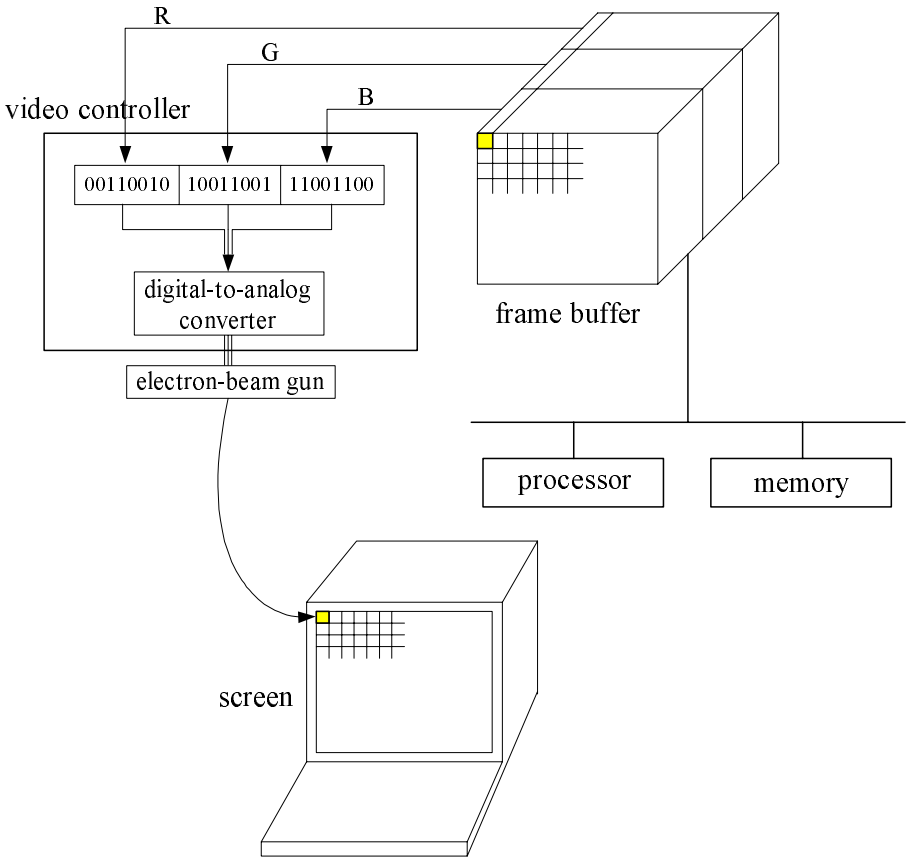



그림 1.6: 개념적인 래스터 그래픽스 시스템

모드로 사용을 하겠다.

## 2.3 래스터 그래픽스 시스템

지금까지 래스터 이미지와 컴퓨터 색깔의 표현 방법에 대하여 간략히 살펴보았다. 요즈음의 대부분의 컴퓨터 그래픽스 관련 시스템들은 래스터 디스플레이(raster display)에 기반한 래스터 그래픽스 시스템(raster graphics system) 방식을 사용한다. OpenGL 프로그래밍도 이러한 시스템에 기반을 하기 때문에, 이 절에서는 그림 1.6에 도시된 것과 같은 래스터 그래픽스 시스템에 대하여 간략히 알아보도록 하겠다.

래스터 그래픽스 시스템은 여러 가지 방향에서 생각을 할 수 있겠지만, 여기서는 3차원 그래픽스 프로그래밍에 가장 자연스러운 관점에서 이해를 해보자. 우리가 그래픽스 프로그래밍을 하는 이유는 머리 속으로 생각을 하고 있는 내용을 프로그래밍 언어와 OpenGL과 같은 그래픽스 라이브러리를 사용하여 이미지의 형태로 생성을 하기 위해서이다. 그 결과 생성되는 이미지를 화면(screen)도시를 하는데, 우리가 사용하는 CRT 모니터는 앞에서 설명한 래스터 이미지와 같이 화면의 영역이 유한 개의 화소로 나누어져 있다. 래스터 그래픽스 시스템에서는 한 순간에 화면에 도시되는 이미지에 대한 데이터가 항상 프레임 버퍼(frame buffer)라고 하는 특수한 시스템 메모리에 저장이 되어야 한다[4]. 그림에서와 같이 화면의 각 화소마다 정해진 수의 비트가 할당이 되는데, 이 때의 비트 수는 화면에 얼마나 많은 수의 색깔을 표시할 수 있는가에 대한 척도로서, 이를 프레임 버퍼의 깊이(frame buffer's depth)라 부른다. 앞에서도 비슷한 이야기를 했지만, 24 비트를 사용하면 약 1670만 개의 색깔, 그리고 16 비트를 할당하면 약 6만 5천 개의 색깔을 도시할 수 있다.

화면의 해상도, 프레임 버퍼의 크기, 표현할 수 있는 색깔의 수의 크기는 서로 연관이 되어 있음을 쉽게 알 수가 있다. 만약 $1280 \times 1024$의 해상도를 가지는 화면에 1670만 개의 색깔, 즉 24 비트 색깔을 표현하려면, 앞에서도 계산한 바와 같이 프레임 버퍼의 크기는 최소한 3.75MB보다 커야 한다. 만약 사용하는 시스템의 프레임 버퍼의 크기가 2MB라면 화면의 해상도를 낮추던지, 아니면 원하는 색깔의 수를 줄여야 한다. 간단한 연습으로 2MB의 프레임 버퍼를 가지는 시스템에서 해상도의 $1024 \times 768$인 화면에 16 비트 색깔을 표현할 수 있는지를 계산해보기 바란다.

재차 강조를 하면, 래스터 그래픽스 시스템에서는 래스터 이미지의 형태의 이미

[4]프레임 버퍼라는 용어 대신에 비디오 램(video ram), 비디오 버퍼(video buffer), 비디오 메모리(video memory), 리프레쉬 버퍼(refresh buffer), 리프레쉬 메모리(refresh memory)등의 용어가 사용 된다. 이 책에서는 컴퓨터 그래픽스 분야에서 가장 자연스럽게 사용되는 프레임 버퍼라는 용어를 사용하겠다.

지를 화면에 도시하기 위해서는 항상 그 내용이 프레임 버퍼에 저장이 되어 있어야 한다. 그러면 과연 메모리에 저장이 되어 있는 이미지가 어떠한 경로를 통하여 화면에 나타날까? 래스터 그래픽스 시스템 안에는 비디오 제어기(video controller)라는 장치가 있는데, 이 장치가 프레임 버퍼로부터 각 화소에 대한 색깔을 읽어 화면의 해당 지역에 대응이 되는 색깔이 나타나도록 해준다. 그림 1.6에서와 같이 비디오 제어기가 한 화소의 색깔이 저장이 되어 있는 프레임 버퍼의 메모리 영역을 액세스하여 빨간색(R), 초록색(G), 파란색(B) 채널로 구성이 된 색깔을 읽어 낸다. 이 색깔은 0과 1로 표현된 디지탈 신호인데, 화면에 해당하는 CRT 모니터는 아날로그 장비이기 때문에 각 채널 값을 아날로그 신호로 변환을 해주어야 한다. 그림에서 보는 바와 같이 디지탈 아날로그 변환기(digital-to-analog converter)가 프레임 버퍼로부터 읽어들인 디지탈 신호인 RGB 색깔의 각 채널에 대하여 아날로그 신호인 전압의 형태로 변환을 해주게 된다.

CRT 모니터의 화면의 각 화소의 영역에는 각각 빨간색, 초록색, 파란색의 빛을 발하는 세 개의 형광체(phospher)가 붙어 있다. CRT 모니터 안에 있는 전자총(electron-beam gun)이 이러한 형광체들을 향하여 해당하는 전압의 세기로 전자빔을 쏴주면, 형광체의 인이 전압의 세기에 비례하는 밝기로 발광을 하게 된다. 따라서 실제로는 화면의 한 화소에 해당하는 영역에서 세 개의 서로 다른 색깔의 빛이 발하고 있으나, 이들이 우리 눈이 구별을 할 수 없을 정도로 가까이 붙어있기 때문에 혼합을 하여 하나의 색깔로 인식을 하게 된다.

형광체인 인은 짧은 시간 동안만 빛을 발하므로 화면의 내용을 유지하기 위해서는 비디오 제어기가 주기적으로 반복해서 이러한 과정을 수행하여야 한다. 다시 말해서 프레임 버퍼의 내용을 반복적으로 스캔(scan)하면서, 프레임 버퍼에 해당하는 이미지를 계속해서 그려주어야 한다. 보통 비디오 제어기는 화면의 가장 윗줄부터

한 줄씩 아래로 내려가면서, 그리고 각 줄을 왼쪽에서 오른쪽으로 화소를 하나씩 훑어가면서 위의 과정을 반복을 한다(그림 1.7). 이 때 각 줄을 스캔 라인(scan line)이라 하고, 이러한 과정을 래스터 스캔(raster scan)이라 한다. 한 번 화면 전체에 대하여 래스터 스캔을 하는 것을 리프레쉬(refresh)한다고 하는데, 보통의 CRT 모니터는 1초에 60에서 72번 정도의 속도로 화면을 리프레쉬 해준다.

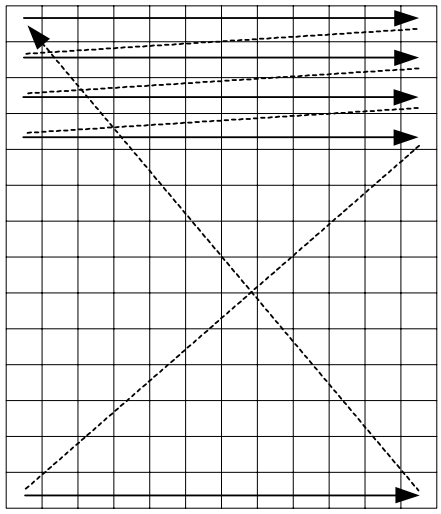


그림 1.7: 래스터 스캔의 순서

전기한 바와 같이 OpenGL과 같은 실시간 렌더링을 위한 그래픽스 라이브러리를 사용하여 프로그래밍을 하는 목적은 원하는 내용의 이미지를 생성하여 화면에 나타나도록 하는 것이다. 이를 좀 더 자세하게 말한다면, 그래픽스 프로그래밍을 통하여 우리가 원하는 내용의 이미지에 대한 RGB 색깔 데이터를 계산하여 프레임 버퍼의 해당하는 영역에 저장을 하는 것이다. 그림 1.6을 다시 보면 우리가 작성한 프로그램이 프로세서(processor)와 메모리(memory)를 사용하여 원하는 이미지를 프레임 버퍼에 저장을 하면, 항상 반복적으로 래스터 스캔을 하고 있는 비디오 제어기에 의해 화면에 도시가 된다. 3차원 그래픽스 렌더링 기법을 통하여 이미지를 생성을 하려면, 일반적으로 계산량이 많기 때문에 CPU와 같은 일반 프로세서뿐만 아니라 보통 그래픽스 가속기라 하는 그래픽스 전용 프로세서를 같이 사용하여 이미지 생성 속도를 높이려 한다. 또한 렌더링 계산 중에는 단순히 결과 이미지뿐만 아니라 다양한 형태의 그래픽스 데이터가 생성이 되고 저장이 되어야 하기 때문에, 일반 메모리뿐만 아니라 그래픽스 전용 메모리도 같이 사용하게 된다. 지금까지 래스터 그래픽스 시스템에 대한 기본적인

개념에 대하여 살펴보았는데, 이 책에서는 OpenGL이라는 소프트웨어 툴을 사용하여 프레임 버퍼에 저장할 이미지를 생성하는 프로그래밍 방법에 대하여 알아보려 한다.

## 2.4 프레임 버퍼

앞 절에서 프레임 버퍼는 화면에 도시할 래스터 이미지를 저장하는 메모리라 하였는데, 실제로는 프레임 버퍼는 그러한 이미지뿐만 아니라 렌더링 계산 중에 생성이 되어 최종 이미지 계산에 사용이 되는 래스터 이미지 형태의 중간 데이터를 저장해 주는 포괄적인 그래픽스 전용 메모리를 뜻한다. 따라서 프레임 버퍼를 다른 관점에서 보면 화면에 도시되는 래스터 이미지의 각 화소에 대한 다양한 정보를 저장하는 특수한 메모리라 할 수 있다.

프레임 버퍼는 서로 다른 사용 목적을 가지는 여러 개의 버퍼로 구성이 되는데, 앞에서 설명한, 화면에 도시할 이미지를 저장하는 버퍼를 색깔 버퍼(color buffer)라 부른다. 보통은 한 개의 색깔 버퍼를 사용하면 되나, 이미지의 내용이 부드럽게 변하는 애니메이션의 내용을 도시하기 위해서는 두 개의 색깔 버퍼를 사용해야 한다. 애니메이션을 위해서는 렌더링 계산을 통하여 조금씩 변하는 이미지를 빠르게 생성을 한 후 화면에 도시하는 작업을 반복해주어야 한다. 따라서 프로세서는 계속해서 화면에 그려줄 이미지의 내용을 생성하여 프레임 버퍼에 넣어주고, 비디오 제어기는 이와는 무관하게 반복적으로 프레임 버퍼의 내용을 앞에서 설명한 순서대로 래스터 스캔을 하며 화면에 그림을 그려주게 된다.

문제는 프레임 버퍼를 공유해야 하는 이 두 작업, 즉 이미지의 생성과 도시간에는 어떠한 공조도 없기 때문에, 종종 화면에 나타나는 이미지의 내용에 문제가 발생한다. 예를 들어 프로세서가 $\frac{1}{20}$초마다 한 장씩 새로운 이미지를 만들어 낸다고

가정하면, 프로세서는 $\frac{1}{20}$초라는 시간 동안에 이미지를 만들면서 조금씩 프레임 버퍼의 내용을 변경을 하게 된다. 반면에 비디오 제어기는 프레임 버퍼의 내용이 조금씩 변하고 있는 동안, $\frac{1}{72}$초에 한 번씩의 속도로 프레임 버퍼의 내용을 스캔을 하게 된다. 따라서 화면에는 완성된 이미지가 하나씩 차례대로 도시되면서 애니메이션이 되는 것이 아니라, 생성하고 있는 내용의 일부, 혹은 경우에 따라서는 전부가 불규칙하게 화면에 도시가 되어, 그 결과 애니메이션이 매우 눈에 거슬리는 현상이 발생하게 된다.

이러한 문제를 극복하는 방법은 두 개의 색깔 버퍼를 사용하는 것이다. 한 순간에 비디오 제어기는 완성된 이미지가 들어 있는 색깔 버퍼로부터 래스터 스캔을 하고, 그 동안 프로세서는 또 다른 색깔 버퍼에 이미지를 생성을 하게 된다. 이 때 전자의 역할을 하는 버퍼를 앞 버퍼(front buffer)라 하고, 후자의 역할을 하는 버퍼를 뒤 버퍼(back buffer)라 한다. 일단 한 이미지의 생성이 끝나면, 앞 버퍼와 뒤 버퍼가 바뀌게 되는데, 그 결과 새롭게 계산한 이미지가 화면에 도시가 되고, 프로세서는 다른 버퍼에 다시 다음 이미지를 만들게 된다. 이렇게 두 개의 버퍼에 대하여 앞 버퍼와 뒤 버퍼의 역할을 바꾸어 가면서 작업을 하는 것을 더블 버퍼링(double buffering)이라 부른다.

더블 버퍼링을 하면 항상 완성된 이미지가 연속적으로 화면에 도시가 되므로, 색깔 버퍼를 한 개 사용할 때보다 훨씬 부드러운 애니메이션을 할 수가 있다. 반면에 필요한 프레임 버퍼의 크기가 두 배로 증가를 하기 때문에 시스템에 부담이 된다. 특히 제한된 크기의 프레임 버퍼에 대하여 색깔 버퍼를 두 개를 사용할 수 없는 경우에는 종종 한 개의 색깔 버퍼를 두 개로 나누어 사용을 하게 된다. 이 경우, 예를 들어 깊이가 24 비트인 색깔 버퍼라면 12 비트 색깔 버퍼를 두 개를 사용하게 되는데, 그 결과 화면에 표현할 수 있는 색깔의 개수가 줄어든다는 문제가 있다.

더블 버퍼링 외에도 입체감이 있는 이미지를 화면에 도시하기 위하여 두 개의 색깔 버퍼를 사용하기도 한다. 사람의 뇌는 약간의 각도 차이가 있는 왼쪽 눈과 오른쪽 눈이 인지한 두 이미지의 차이를 통하여 입체감, 즉 거리감을 느끼게 된다. 이를 응용하여 컴퓨터 그래픽스 시스템에서도 이미지를 생성할 때, 양쪽 눈에서 바라본 것에 해당하는 이미지를 두 개 만들어 화면에 동시에 도시한 후, 특수한 안경을 통하여 각 눈이 자신에 해당하는 이미지만 볼 수 있도록 하여 입체감을 느끼게 해준다. 우리가 보통 말하는 3차원 입체 영화가 바로 이러한 원리에 기반을 두고 있다. 입체감 효과를 위하여 생성한 두 개의 이미지를 저장해주는 색깔 버퍼를 스테레오 버퍼(stereo buffer)라 하는데, 각 버퍼를 구별하여 왼쪽 버퍼(left buffer)와 오른쪽 버퍼(right buffer)라 한다. 만약 더블 버퍼링을 하면서 스테레오 버퍼를 사용을 한다면, 앞-왼쪽, 뒤-왼쪽, 앞-오른쪽, 뒤-오른쪽 등 네 개의 버퍼를 사용해야 한다. 따라서 $1280 \times 1024$ 해상도를 가지는 시스템에서 깊이가 24 비트인 색깔 버퍼를 네 개를 사용한다면 15MB의 프레임 버퍼가 필요하게 된다.

OpenGL 시스템은 RGBA 색깔 모드를 사용하므로, R, G, B 채널 외에도 알파 채널이라 하는 A 채널 값도 저장을 해주어야 한다. 알파 채널은 보통 색깔 버퍼에 같이 저장이 되지만, 경우에 따라 알파 채널에 대한 메모리를 따로 생각해서 알파 버퍼(alpha buffer)라 부르기도 한다.

이러한 색깔 버퍼 외에 이미지 생성 중에 각 화소에 대한 깊이 정보를 저장하는 깊이 버퍼(depth buffer)가 있는데, 이 버퍼는 앞으로 자연스럽게 이해를 하겠지만 Z 버퍼(Z buffer)라 하기도 한다. 깊이 버퍼에는 보통 한 화소 당 16개에서 32개 정도의 비트를 할당하는데, 색깔 버퍼와 함께 가장 기본적인 프레임 버퍼를 이루고 있다.

또한 중요한 프레임 버퍼로 스텐실 버퍼(stencil buffer)와 축적 버퍼(accumulation

buffer)를 들 수가 있다. 스텐실 버퍼는 주로 화면의 특정 지역에만 그림을 그릴 수 있도록 제한을 가하는데 사용이 되는데, 보통 화소 당 1개에서 8개까지의 비트를 할당한다. 반면 축적 버퍼는 그 이름이 암시하듯이 조금씩 다른 이미지를 여러 장 생성하여, 축적을 한 후 평균 이미지를 계산하여 여러 가지 렌더링 효과를 생성하는데 사용이 된다. 보통 축적 버퍼는 그 특성상 색깔 버퍼보다 화소 당 더 많은 수의 비트를 할당한다.

# 제 3 절 윈도우 시스템과 윈도우 프로그래밍

## 3.1 윈도우 시스템과 OpenGL

일반적으로 사용자들이 컴퓨터를 사용할 때 대부분 윈도우 시스템(window system)이라고 하는 하나의 사용자 인터페이스를 통하여 작업을 한다. 마이크로 소프트사의 운영 체제를 사용하는 PC에서의 윈도우스 98, 윈도우스 NT, 윈도우스 2000 등의 윈도우스(Microsoft Windows)와 유닉스나 리눅스 계열의 운영 체제를 사용하는 컴퓨터 상에서의 X 윈도우 시스템(X Window System)이 최근 많이 사용하는 윈도우 시스템의 예라 할 수가 있다. 윈도우 시스템의 일반화에 따라 응용 프로그램을 사용할 때뿐만 아니라 이러한 환경에서 새로운 소프트웨어를 개발할 때에도 사용 윈도우 시스템이 제공하는 기본 도구들을 효과적으로 사용하여야 한다. 이는 어떤 프로그램을 작성을 하려해도 항상 출발점은 윈도우 시스템의 사용과 관련된 프로그래밍, 즉 윈도우 프로그래밍(window programming)이라는 사실을 의미한다. 마찬가지로 이 책의 주제인 OpenGL을 사용하여 그래픽스 프로그래밍을 하기 위해서도 사용 윈도우 시스템에서의 윈도우 프로그래밍이 바탕이 되어야 한다.

OpenGL 프로그래밍 관점에서 볼 때 한 가지 중요한 점은 윈도우 시스템도 하나

의 래스터 그래픽스 시스템으로서의 성질을 가지고 있다는 사실이다. 즉 어떤 윈도우 시스템이건 그에 대한 프레임 버퍼가 정의되어야 하고, 내부적으로 윈도우의 각 화소의 색깔을 어떻게 표현할 것인가가 설정되어야 하며, 선분, 다각형, 폰트 등의 기본 요소들을 효과적으로 그려주는 기능들이 제공되어야 한다. 물론 윈도우 시스템은 사용하는 운영 체제와 긴밀하게 연결이 되어 포괄적인 사용자 인터페이스 기능을 제공하지만, 래스터 그래픽스 시스템으로서의 측면은 윈도우 프로그래밍을 이해하는데 있어 중요한 역할을 한다.

그러면 윈도우 시스템도 하나의 래스터 그래픽스 시스템으로서 그래픽스 작업을 위한 수단을 제공한다면 굳이 OpenGL과 같은 또 다른 그래픽스 시스템을 사용을 해가면서 프로그래밍을 하여야 하는 이유는 무엇일까? 대략적으로 80년대 초 윈도우 시스템이 처음 개발이 될 때만 해도 3차원 그래픽스 작업은 일반 사용자들이 범용 컴퓨터에서 사용하기에는 너무나 많은 계산량을 요구했기 때문에 주로 2차원 그래픽스 기능만 고려하였다. 따라서 직선을 그린다거나, 다각형을 그린다거나 폰트를 그리는 것과 같은 2차원 그래픽스 기능을 효율적으로 구현하는데 초점을 맞추게 되었고, 실제로 대부분의 윈도우 시스템에서 2차원 그래픽스 기능은 아주 효과적으로 구현이 되어왔다.

시간이 흘러감에 따라 컴퓨터 하드웨어의 성능이 급격히 향상이 되었고, 그에 따라 과거에는 불가능하게 보였던 범용 컴퓨터에서의 실시간 3차원 그래픽스 작업의 실현 가능성이 점차로 증대하게 되었다. 하지만 X 윈도우 시스템에서 가장 기본이 되는 Xlib 라이브러리나 마이크로 소프트 윈도우스 시스템의 Win32 라이브러리는 효과적인 2차원 그래픽스를 위한 함수들을 제공하는 반면 3차원 그래픽스를 위한 기능은 제공하지를 못했다. 3차원 그래픽스의 시장성은 방대하다고 할 수가 있기 때문에, 여러 컴퓨터 회사들이 기존의 윈도우 시스템에서 3차원 그래픽스 프로그래

밍을 할 수 있는 도구들을 개발하려 하였다. 특히 Direct3D처럼 특정 윈도우 시스템 환경에서 새로운 라이브러리를 개발하거나, XPHIGS나 OpenGL과 같은 기존의 3차원 그래픽스 시스템을 윈도우 시스템에 연결을 하는 방식으로 윈도우 시스템을 확장하여 왔다.

이 책은 특정 시스템에 영향을 받지 않는 OpenGL 시스템에 관한 책이므로 후자의 방식을 생각해보자. X 윈도우 시스템을 사용하건 마이크로 소프트 윈도우스를 사용하건 3차원 그래픽스 프로그래밍을 하기 위해서는, 1. 우선 기본적으로 사용 윈도우 시스템에서 제공하는 래스터 그래픽스 시스템을 기반으로 하여 윈도우 프로그래밍을 하여야 한다. OpenGL을 사용하는 이유는 3차원 그래픽스 렌더링 작업에 관련된 프로그래밍을 효과적으로 수행하기 위한 것이다. 따라서 이를 위하여 2. 윈도우 프로그래밍 문맥에서 하나의 추상적인 래스터 시스템인 OpenGL 시스템을 윈도우 시스템의 그래픽스 시스템에 연결을 하고, 3. 이후에는 OpenGL에서 제공하는 함수들을 사용하여 3차원 그래픽스 프로그래밍을 하면, 4. 원하는 OpenGL 작업이 실제로 하드웨어를 제어하고 있는 사용 윈도우 시스템이 효율적으로 이해할 수 있는 형태로 전환이 되어 그 결과가 화면에 나타나게 된다.

다시 말해서 전체적인 윈도우 프로그래밍 문맥에 OpenGL 프로그래밍을 끼워 넣는 것으로 볼 수가 있고, 따라서 3차원 그래픽스 작업을 제외한 모든 프로그래밍(예를 들어 마우스의 움직임을 처리하는 부분 등의)은 윈도우 프로그래밍을 통하여 수행하여야 한다. 대부분의 윈도우 시스템들은 OpenGL 시스템을 자신의 윈도우 시스템에 연결을 할 수 있도록 시스템을 확장하였고, 프로그래머가 그러한 작업을 효과적으로 할 수 있도록 라이브러리 함수들을 제공하게 되었다. X 윈도우 시스템의 경우 GLX, 그리고 마이크로 소프트 윈도우스 시스템에서는 WGL이라는 라이브러리를 통하여 기존의 라이브러리가 확장이 되었는데, 그 외에도 애플의 맥킨토시와

IBM의 OS/2 Warp 운영 체제에서 사용하는 윈도우 시스템에서도 OpenGL 확장을 위하여 각각 AGL과 PGL과 같은 확장 라이브러리를 제공한다.

## 3.2 윈도우 프로그래밍의 개념

이 절에서는 윈도우 시스템에서 개발된 윈도우 프로그램들이 실제로 어떻게 작동을 하는지에 대하여 알아보자. 윈도우 프로그래밍이라는 주제는 이 책의 범위를 벗어나므로, 여기서는 그 개념을 단순화시켜 전형적인 상황에 대해서만 간략하게 살펴보도록 하겠다. 자세한 내용은 윈도우 프로그래밍에 관련된 서적을 참조하기 바란다.

처음에 응용 프로그램을 수행을 시키면, 보통 그 프로그램의 사용에 필요한 윈도우들이 화면에 뜨고, 그 안에는 여러 기능을 위한 버튼이나 대화 창 등의 대화 도구들이 그려지게 된다. 이후 응용 프로그램은 가만히 기다리고 있다가 사용자가 마우스를 움직이거나, 마우스 버튼을 누르거나, 또는 키보드를 누르는 등의 행위를 통하여 원하는 작업을 요청하면, 응용 프로그램은 그에 대한 서비스를 제공하게 된다. 예들 들어 워드 프로세서 프로그램에서 현재 마우스가 있는 곳에 'a'라는 글자를 삽입하는 과정을 생각해보자. 사용자가 키보드의 'a'라는 키를 눌러 이곳에 'a'라는 문자를 삽입해 달라는 요청을 하면, 응용 프로그램에서는 이러한 요청을 받아들여 화면에 'a'라는 글자를 그려주고 나머지 그 뒤의 문자들은 한 칸씩 뒤로 밀어준다. 이렇게 윈도우 시스템 상에서의 응용 프로그램은 '사용자의 서비스 요청'-'응용 프로그램의 서비스 제공'이라는 과정이 계속적으로 반복이 되면서 수행이 된다고 볼 수가 있다.

그런데 여기서 중요한 것은 과연 윈도우 시스템 내부에서 그러한 과정이 어떻게 수행이 되는가를 이해를 해야 한다는 점이다. 우리가 작성하는 응용 프로그램은 단

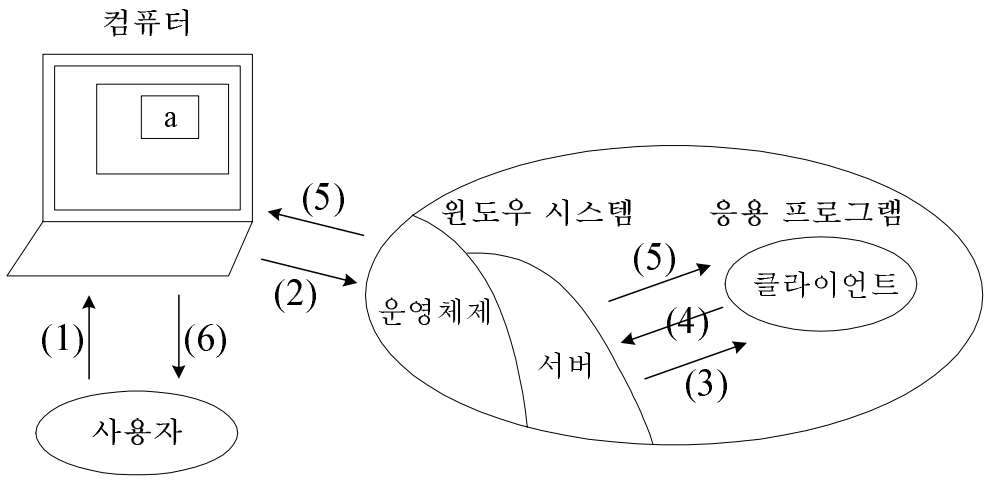



그림 1.8: 윈도우 응용 프로그램의 수행 개념

순히 하나의 클라이언트 프로그램이다. 시스템의 관점에서 보면 응용 프로그램은 지속적으로 서비스를 요청하는 일종의 손님으로서, 사용하고자 하는 시스템을 직접적으로 제어를 할 수 있는 권한이 없다. 실제로 사용 컴퓨터 시스템의 모든 하드웨어와 입출력 장치를 '장악'하고 제어하는 것은 운영 체제와 윈도우 시스템(또는 윈도우 시스템 서버)이다. 윈도우 시스템은 운영 체제의 일부로 볼 수도 있고, 또한 운영 체제와 긴밀하게 연결된 하나의 시스템 프로그램이라 할 수도 있다.

어떠한 개념으로 윈도우 시스템을 이해를 하건, 여기서는 설명을 단순하게 하기 위해서 윈도우 시스템의 서버가 컴퓨터의 모니터, 마우스, 키보드 등을 비롯한 하드웨어를 제어하고 있다고 가정하자. 이러한 상황에서는 화면에 그림을 그리는 작업을 하려해도 클라이언트인 응용 프로그램이 직접 그리는 것이 아니라, 윈도우 서버에게 원하는 작업을 요청하여 서버가 그림을 그려주는 방식을 취한다. 또한 사용자가 응용 프로그램의 어떤 기능을 사용하기 위해 마우스를 움직인다고 할 때, 그러한 행위가 직접 응용 프로그램에게 보고가 되는 것이 아니라, 입출력 장치를 제어하고 있는 서버가 그러한 사실을 인지한 후 해당 응용 프로그램에게 알려주는 방식을 취한다.

그림 1.8은 윈도우 응용 프로그램의 단순화된 수행 개념도를 보여주고 있다. 즉 사용자가 응용 프로그램 사용을 위하여 어떤 행위를 취했을 때(예를 들어 키보드를 누르거나, 마우스를 움직이거나, 또는 마우스 버튼을 누르는 등의), 그러한 요청이 어떠한 과정을 거쳐 수행이 되는가를 설명하고 있다. 사용자로부터 어떤 입력이 들어왔거나 응용 프로그램이 무엇인가를 처리해줘야 할 필요성이 발생하였을 때((1)), 시스템을 제어하고 있는 운영 체제와 서버가 그러한 사실을 인식을 한 후((2)), 해당 응용 프로그램에게 그러한 사실을 통지하게 된다((3)). 이는 서버가 응용 프로그램에서 응용 프로그램이 사용하고 있는 윈도우에서 현재 어떠한 일이 일어났으니 그에 적합한 처리를 하라고 알려주는 것이라 할 수 있다. 이렇게 그것이 무엇이건 응용 프로그램이 어떤 처리를 해주어야 할 필요성이 발생하였을 때, 이벤트(event)가 발생되었다고 한다. 여기서는 설명을 단순하게 하기 위해 이에 대한 자세한 설명은 생략하겠지만, 한 가지 언급을 해야 할 것은 이벤트가 발생하는 상황은 단순히 사용자의 입력 장치의 사용보다 훨씬 다양하다는 점이다. 예를 들어 어떤 응용 프로그램을 사용하다가 이메일을 보기 위해 이메일 처리 프로그램의 윈도우를 원래의 윈도우 위에 띄워 작업을 한 후 그 윈도우를 제거하면, 이 때 원래의 윈도우의 내용이 지워져 다시 그려야 하기 때문에 윈도우 서버가 해당 응용 프로그램에게 윈도우의 어느 지역을 다시 그려주어야 하는 이벤트가 발생하였다고 알려준다.

서버는 이벤트 메시지(event message)를 통하여 응용 프로그램의 윈도우에 대하여 발생한 사건을 알려준다. 그러면 응용 프로그램은 이벤트 메시지를 받아 그것을 해석하여 자신의 윈도우에 어떠한 일이 일어났는지를 인지하여, 그에 맞는 처리를 해주게 된다. 예를 들어 사용자가 왼쪽 마우스를 눌렀다고 가정해보자. 만약 그러한 행위가 이 응용 프로그램에서는 화면의 색깔을 바꾸는 의미를 가지고 있다면, 응

용 프로그램이 그에 맞는 처리를 해주어야 한다. 중요한 것은 응용 프로그램이 직접 화면을(정확하게 말하면 윈도우 시스템에서 사용하고 있는 래스터 그래픽스 시스템의 색깔 버퍼를) 원하는 색깔로 직접 바꿀 수가 없다는 점이다. 앞에서 언급한 바와 같이 하드웨어는 윈도우 서버가 제어를 하고 있기 때문에, 클라이언트인 응용 프로그램은 자신이 직접하는 것이 아니라 서버에게 원하는 서비스를 요청하게 된다((4)). 서버가 응용 프로그램으로부터 그러한 요청을 받아들여 화면의 색깔을 바꾸어 주고((5)), 경우에 따라 요청 서비스에 대한 결과를 응용 프로그램에게 알려주면((5)), 사용자는 자신의 행위에 대한 결과를 컴퓨터를 통하여 인지하는 것이다((6)).

지금까지 윈도우 응용 프로그램이 어떻게 수행이 되는지에 대하여 간략히 알아보았는데, 응용 프로그램은 바로 이러한 틀에서 클라이언트 프로그램으로서의 역할을 수행할 수 있도록 작성이 되어야 한다. 대략적으로 응용 프로그램을 작성하기 위해서는 무엇보다도 크게 두 가지 부분을 코딩을 해야 한다. 앞에서 설명한 바와 같이 응용 프로그램의 가장 중요한 요소는 바로 윈도우 서버로부터 이벤트 메시지를 받았을 때, 그 이벤트의 종류에 따라 적절히 처리를 해주는 부분이라고 할 수 있다. 또한 사용자가 응용 프로그램을 수행시키면 대부분 한 개 이상의 윈도우가 화면에 뜨기 때문에, 처음에 화면에 뜰 윈도우의 성질을 결정하고 화면에 띄워주는 부분도 작성이 되어야 한다[5]. 다시 말해서 윈도우 프로그램을 작성할 때는 1. 윈도우의 초기화 및 2. 이벤트 메시지의 처리 두 가지가 가장 기본이 되는 요소라 할 수 있는데, 실제로 대부분의 노력이 두 번째 요소를 프로그래밍하는데 쓰인다. 전기한 바와 같이 여기서는 편의상 윈도우 프로그래밍의 개념을 상당히 단순화를 시켰지만, 이 두 가지 요소가 가장 기본이 됨은 분명하다.

[5] 윈도우를 화면에 띄우는 것을 윈도우를 매핑(mapping)한다고 한다.

## 3.3 간단한 윈도우스 프로그램의 예

이 절에서는 마이크로 소프트의 윈도우스 환경에서 제공하는 Win32 함수들을 사용하여 간단한 윈도우 프로그램을 작성해보자(예제 프로그램 1.A참조). 아래의 프로그램 예 1.1의 예제 프로그램을 수행시켜보면, 그림 1.9와 같이 가로, 세로 500 화소의 해상도를 가지는 윈도우가 화면에 나타난다. 이 윈도우의 배경은 푸른 계통의 색깔(RGB 색깔 = (66, 66, 111))로 칠해져 있고, 왼쪽 가운데에 흰 색의 선분이 그려져 있다. 또한 윈도우 안으로 마우스를 움직여 'q' 키를 누르면 프로그램의 수행을 종료하도록 되어 있다.

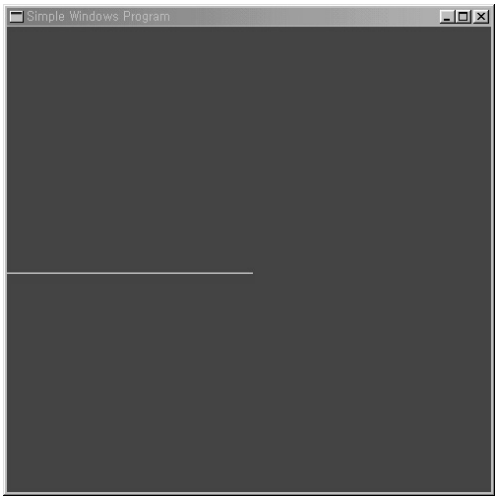



그림 1.9: 프로그램 예 1.1의 수행 결과

이 프로그램은 두 개의 함수 `WinMain(*)`과 `WndProc(*)`으로 구성이 되어 있다. 전자는 이 윈도우 프로그램의 메인 프로그램에 해당하는데, 이 함수의 대부분은 앞에서 설명한 윈도우 프로그램의 첫번 째 요소인 윈도우의 초기화를 위한 코드임을 알 수가 있다. 우선 Line (a)에서 Line (d)까지의 코드는 생성을 하려하는 윈도우의 일반적인 성질을 설정하고 있다. 즉 윈도우 클래스(window class) 타입(`WNDCLASSEX`)으로 선언된 스트럭춰 변수 `wcex`에 희망하는 윈도우의 성질을 기술하고 있다. 예를 들어 Line (c)와 그 다음 문장은 이 윈도우 클래스에 속하는 윈도우에서 사용할 아이콘과 커서 모양을 지정하고 있다. 바로 그 밑의 문장에서 변수 `wcex.hbrBackground`에는 이 윈도우에서 사용할 배경 색깔 (66, 66, 111)이 지정이 되고 있다. Line (b)에서는 또 하나 중요한 윈도우의 속성인 윈도우 프로시져(window procedure)가 변수

wcex.lpfnWndProc에 설정이 되고 있다. 이 예에서는 두 번째 함수인 WndProc(*)이 윈도우 프로시져로 지정이 되는데, 윈도우 프로시져는 윈도우 프로그래밍의 두 번째 요소인 이벤트 메시지 처리를 위한 함수이다. 따라서 이렇게 속성을 지정할 경우 이 윈도우 클래스에 기반을 둔 윈도우에 대하여 이벤트가 발생이 되면 그것을 처리하기 위하여 바로 이 함수가 수행이 된다. 이렇게 윈도우 클래스에 대한 속성을 설정한 후 Line (e)에서 등록을 하게 된다.

다음 Line (f)에서 CreateWindow(*) 함수를 사용하여 실제로 사용할 윈도우를 내부적으로 생성을 한다. 이 때 등록된 윈도우 클래스의 속성을 가지는 윈도우가 생성이 되는데, 이 함수의 인자들을 통하여 좀 구체적인 속성을 지정할 수가 있다. 예를 들어 인자들 중 두 개의 숫자 500은 생성하려는 윈도우의 가로, 세로 크기를 나타낸다. 이렇게 윈도우 시스템의 내부에서 윈도우를 생성을 한 후, 다음 문장 ShowWindow(hwnd, nCmdShow);을 통하여 실제로 윈도우를 화면에 매핑을 해준다. 그 다음 아래에서 다시 설명하겠지만 UpdateWindow(*) 함수를 호출하여 화면의 초기 내용이 그려지도록 한다.

다음 Line (g)에서 마치 무한 루프처럼 보이는 while 루프로 들어간다. 앞에서 응용 프로그램은 서버로부터 계속적으로 들어오는 이벤트 메시지를 처리해주는 과정을 반복한다고 했는데, 이 '메시지 루프'(message loop)가 그에 해당한다. 프로그램을 수행하는 과정에서 이벤트는 동시 다발적으로 발생할 수 있으므로, 윈도우 서버는 이벤트가 발생할 때마다 해당 이벤트 메시지를 메시지 큐(message queue)에 넣어준다. 응용 프로그램은 메시지 루프안으로 들어가 기다리고 있다가 서버로부터 메시지가 들어오면 GetMessage(*) 함수를 통하여 메시지 큐에서 메시지를 꺼내 온다. 이 때 MSG 타입의 스트럭춰 변수 msg에는 현재 이벤트 메시지에 관한 여러 정보(이벤트 종류, 이벤트가 발생한 윈도우, 발생 이벤트에 고유한 정보, 이벤트 발생 시간,

이벤트 발생 시의 마우스의 위치 등)가 저장이 된다. 다음 TranslateMessage(*) 함수를 통해 키보드 관련 처리를 한 후 DispatchMessage(&msg); 문장을 수행시키면, 윈도우가 발생한 윈도우의 윈도우 프로시져로 등록된 함수가 호출이 된다. 해당 윈도우 프로시져에서 현재 이벤트의 처리가 끝나면 DispatchMessage(*) 함수의 수행이 종료가 되고, 다시 다음 이벤트를 처리하기 위하여 GetMessage(*) 함수를 반복적으로 수행하게 된다.

**프로그램 예 1.1** 간단한 윈도우스 프로그래밍 예.

```
#include "stdafx.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
   HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
 HWND hwnd; MSG msg; WNDCLASSEX wcex;
 static char szAppName[] = "SimpleWinProg";

 wcex.cbSize = sizeof(WNDCLASSEX); // Line (a)
 wcex.style = CS_HREDRAW | CS_VREDRAW;
 wcex.lpfnWndProc = (WNDPROC) WndProc; // Line (b)
 wcex.cbClsExtra = 0; wcex.cbWndExtra = 0;
 wcex.hInstance = hInstance;
 wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION); // Line (c)
 wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
 wcex.hbrBackground =
            (HBRUSH) CreateSolidBrush(RGB(66, 66, 111));
 wcex.lpszMenuName = NULL;
 wcex.lpszClassName = szAppName;
 wcex.hIconSm =
            LoadIcon(NULL, IDI_APPLICATION); // Line (d)

 RegisterClassEx(&wcex); // Line (e)
```

```
  hwnd = CreateWindow(szAppName, "Simple Windows Program",
        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
        500, 500, NULL, NULL, hInstance, NULL); // Line (f)

  ShowWindow(hwnd, nCmdShow);
  UpdateWindow(hwnd);

  while (GetMessage(&msg, NULL, 0, 0)) { // Line (g)
   TranslateMessage(&msg);
   DispatchMessage(&msg);
  }
  return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
         WPARAM wParam, LPARAM lParam) {
  HDC hdc;
  PAINTSTRUCT ps;

  switch (message) { // Line (h)
   case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps); // Line (i)
    SelectObject(hdc, GetStockObject(WHITE_PEN));
    MoveToEx(hdc, 0, 250, NULL);
    LineTo(hdc, 250, 250);
    EndPaint(hwnd, &ps); // Line (j)
    break;
   case WM_CHAR:
    if (wParam == 'q') {
     PostQuitMessage(0);
    }
    break;
   case WM_DESTROY:
    PostQuitMessage(0);
    break;
   default:
    return DefWindowProc(hwnd, message, wParam, lParam);
  }
```

```
    return 0;
}
```

이제 윈도우 프로시져인 WndProc(*) 함수에서 이벤트 메시지를 어떠한 방식으로 처리하는지 알아보자. 윈도우 서버는 이 함수의 네 개의 인자들을 적절한 값으로 지정을 한 후 호출을 한다. 첫 번째 인자 hwnd은 어떤 윈도우에서 현재 처리하려는 이벤트가 발생하였지를 알려주고, 두 번째 인자 message는 이벤트의 종류에 대한 고유 값을 가진다. 마지막 두 인자 wParam과 lParam에는 해당 이벤트에 관련된 정보들이 담겨온다. 이 함수에서는 Line (h)에서 어떤 종류의 메시지인지를 파악한 후 적절한 처리를 하고 있다. 이 예제 프로그램에서는 세 가지의 부류의 이벤트 WM_PAINT, WM_CHAR, WM_DESTROY를 처리하고 있다. WM_PAINT는 어떻게 보면 윈도우 프로그래밍에 있어 가장 중요한 메시지 타입 중의 하나라고 할 수 있다. 특히 이 메시지를 처리하는 부분에서 우리의 관심사인 3차원 렌더링에 관련된 코드가 수행이 된다. WM_PAINT는 조금이라도 윈도우의 내용을 다시 그려야 할 필요가 발생하였을 때 서버가 보내주는 메시지이다. 예를 들어 다른 윈도우에 일부가 가렸다가 그 윈도우가 사라졌거나 윈도우의 크기가 변경이 되었을 때, 아이콘 상태에서 다시 정상 상태로 돌아 왔을 때, 또한 렌더링 프로그램에서 화면의 내용을 바꾸고 싶을 때 등 다양한 경우에 윈도우의 내용을 다시 그려주어야 한다. WM_PAINT는 이러한 일이 발생하였을 때마다 들어오는 메시지이기 때문에, 이를 처리해주는 부분에서 윈도우의 내용을 올바르게 그려주는 코드가 오게 된다.

한 가지 궁금한 것은 윈도우가 처음 화면에 뜰 때, 어떠한 경로를 통하여 그림이 그려질까? 윈도우가 처음 화면에 뜨면 아직 그림이 그려진 상태가 아니므로 WM_PAINT 메시지를 통하여 그림을 그려주어야 한다. 앞에서 WinMain(*) 함수에서

ShowWindow(*) 함수를 통하여 화면에 윈도우를 띄운 후, UpdateWindow(*) 함수를 수행을 시킨다고 하였다. 이 함수는 목적은 윈도우 서버가 메시지 큐에 WM_PAINT 메시지를 집어 넣게 해주는데 있다. UpdateWindow(*) 함수를 수행을 시키면 이 메시지가 큐에 들어갔다가 메시지 루프가 시작이 되면 처리가 되고, 따라서 윈도우가 화면에 뜬 후 올바른 내용으로 그려지게 되는 것이다.

마이크로 소프트의 윈도우스 환경에서는 전체 윈도우 시스템 중 그래픽스에 관련된 부분을 GDI(Graphics Device Interface)라 하는데, 윈도우에 무엇인가를 그리기 위해서는 윈도우스의 GDI에서 제공하는 함수들을 사용한다. Line (i)와 Line (j) 사이의 문장들에서 사용된 함수들이 GDI 함수들인데, BeginPaint(*) 함수가 수행될 때 위에서 설정된 배경 색깔로 윈도우 전체를 다시 칠한 후, 윈도우에 점 (0, 250)에서 점 (250, 250)까지 흰 색을 사용하여 선분을 그려준다.

WM_PAINT 메시지 다음에 처리하는 메시지는 WM_CHAR인데, 이는 사용자가 키보드를 눌렀을 때 발생하는 이벤트 메시지이다. 이 메시지의 경우 사용자가 누른 키의 ASCII 코드가 변수 wParam에 담겨오는데, 여기서는 이 값이 'q'이면 응용 프로그램의 수행을 종료하는 과정에 들어간다. 이 때 PostQuitMessage(*) 함수를 수행시키면 서버가 WM_QUIT 메시지를 메시지 큐에 넣어준다. Line (g)의 메시지 루프를 다시 보면, 이 루프는 마치 무한히 돌 것처럼 보이나, 메시지 큐에서 GetMessage(*) 함수가 메시지를 꺼내올 때, 만약 메시지 타입이 WM_QUIT이면 이 함수는 0 값을 리턴하고 따라서 메시지 루프에서 벗어나 프로그램이 종료하게 된다. 어떻게 보면 PostQuitMessage(*) 함수를 호출하는 대신 exit(-1);과 같은 문장을 수행시켜 프로그램을 종료할 수 있으나 여기서 사용하는 방식이 훨씬 더 조직적인 방법이라 할 수 있다. 마지막 WM_DESTROY는 사용자가 Close나 Alt-F4 등의 버튼을 눌러 윈도우를 제거하려할 때 발생하는 메시지로서 마찬가지로 PostQuitMessage(*) 함수를

사용하여 처리를 하고 있다.

전체 윈도우 프로그래밍 중 그래픽스와 관련된 부분은 윈도우에서 제공하는 GDI 함수를 사용한다고 하였다. 이에 대한 자세한 설명은 생략을 하려하는데, 한 가지 이해를 하여야 하는 개념이 바로 디바이스 문맥(device context)이다. 디바이스 문맥은 GDI 내부에서 사용되는 그래픽스와 관련한 모든 정보를 저장하는 자료 구조라 생각하면 된다. 무엇보다도 디바이스 문맥에는 현재 사용하는 그래픽스 관련 디바이스에 대한 구체적인 정보가 저장이 된다. 또한 GDI 함수가 화면에 그림을 그리는데 필요한 여러 그래픽스 속성들에 대하여 현재 설정되어 있는 값들이 저장이 된다. 일반적으로 화면에 그림을 그릴 때, 그에 관련하여 여러 속성들을 지정하여야 한다. 예를 들어 이 예제 프로그램에서 GDI 함수인 `LineTo(*)` 함수를 사용하여 선분을 그릴 때, 선분을 어떤 색깔로 그릴 지, 얼마나 두껍게 그릴 지, 그리고 어떤 패턴을 사용하여 그릴 지 등을 결정하여야 한다. 일반적으로 이러한 속성들은 선분을 그리는 함수를 호출할 때 모두 지정하는 것이 아니라, 미리 필요한 속성들을 관련 함수들을 사용하여 지정을 하고, 실제로 선분을 그릴 때는 좌표 값만 지정하면 디바이스 문맥에 저장되어 있는 속성 값들을 사용하여 선분을 그릴 수 있다. Line (i) 다음의 `SelectObject(hdc, GetStockObject(WHITE_PEN));` 문장은 선분을 그리는데 사용할 색깔을 지정하기 위한 것인데, 그 결과 디바이스 문맥의 해당 값이 바뀌고, `LineTo(hdc, 250, 250);` 문장에서 선분을 그릴 때 그러한 속성 값들을 사용하게 된다. 재차 강조하면 디바이스 문맥에는 그래픽스 작업을 위한 모든 정보가 저장이 되는데, 그러한 정보는 또 하나의 그래픽스 렌더링 툴인 OpenGL 시스템에서 그래픽스 작업을 하려할 때 필요하게 된다.

## 3.4 윈도우스 프로그램과 OpenGL의 연결 예

앞에서의 윈도우 프로그래밍 예에서는 마이크로 소프트 윈도우스 시스템의 일부인 GDI에서 제공하는 함수들을 사용하여 그래픽스 관련 부분을 프로그래밍하였다. 이제 여기서 그래픽스 관련 부분을 OpenGL 함수를 사용하여 코딩하여 보자(예제 프로그램 1.B참조). 앞에서 잠깐 언급한 바와 같이 이를 위해서 우선 GDI에 의해 정의되는 윈도우 시스템의 그래픽스 시스템과 OpenGL이 정의하는 그래픽스 시스템을 연결을 하여야 한다. 마이크로 소프트의 윈도우스 시스템에서는 이러한 작업을 위해 Win32 라이브러리의 기능을 확장하였고, 또한 Wiggle 함수라 부르는 확장 함수들을 OpenGL 라이브러리에 추가를 하였다.

전체적으로 보면 이전의 예제 프로그램과 비교해 볼 때, 두 래스터 그래픽스 시스템을 연결하는 코드가 추가되어야 하고, 또한 GDI 함수들을 사용하여 작성한 그래픽스 관련 코드를 OpenGL 함수들을 사용하여 바꿔 주어야 한다. 프로그램 예 1.2에는 OpenGL을 사용한 예제 프로그램이 주어져 있는데 메인 함수인 `WinMain(*)`은 거의 변함이 없다. 윈도우 프로시져인 `WndProc(*)` 함수를 보면 새로운 메시지 `WM_CREATE`가 처리되고 있음을 볼 수가 있다. 이 메시지는 윈도우가 내부적으로 처음 생성이 될 때, 즉 CreateWindow(*) 함수가 수행이 될 때 발생이 된다. 따라서 이 메시지는 윈도우를 생성을 할 때 필요한 여러 가지 초기화 작업을 하기 위한 코드를 수행하는 데 사용된다. 이 예제에서는 `WM_CREATE` 메시지를 처리하는 과정에서(Line (d)에서 Line (f)까지) OpenGL 시스템을 윈도우에 연결을 시키고 있다.

두 그래픽스 시스템을 연결하는 데 있어 가장 중요한 사항 중의 하나는 OpenGL을 사용하기 위하여 GDI에서 제공하는 화소 형식(pixel format)을 적절히 지정을 해야 한다는 점이다. 화소 형식이란 쉽게 말해서 OpenGL에서 사용할 프레임 버퍼

에 관한 정보라 할 수 있다. 즉 색깔 버퍼를 위해서 한 화소 당 몇 비트를 사용을 할 것인가, 더블 버퍼를 사용을 할 것인가, 깊이 버퍼를 사용할 것인가, 그렇다면 한 화소 당 몇 비트를 사용할 것인가 등을 결정하여야 하는데, 이러한 사항들이 화소 형식이라는 개념을 통하여 설정이 된다. 윈도우 시스템에서 제공할 수 있는 프레임 버퍼의 성능은 실제로 사용하는 하드웨어의 성능에 따라 제한을 받는다. 따라서 1. OpenGL에서는 희망하는 프레임 버퍼의 성능을 화소 형식을 통하여 기술하면, 2. 실제로 하드웨어에 대한 정보를 가지고 있는 GDI에서 하드웨어가 지원을 하는 범위 내에서 OpenGL에서 요구한 성능을 최대한 만족 시켜주는 화소 형식을 결정하여, 3. 윈도우 시스템의 그래픽스에 관련된 모든 정보를 저장하는 디바이스 문맥에 저장을 한 후 그러한 정보를 사용하게 된다.

**프로그램 예 1.2** 윈도우스 환경에서의 OpenGL 연결 예.

```
#include "stdafx.h"
#include <gl/gl.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
void SetDCPixelFormat(HDC hdc);

int APIENTRY WinMain(HINSTANCE hInstance,
   HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
      ⋮
}

void SetDCPixelFormat(HDC hdc) {
 int nPixelFormat;

 static PIXELFORMATDESCRIPTOR pfd = { // Line (a)
    sizeof(PIXELFORMATDESCRIPTOR), 1,
    PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL, PFD_TYPE_RGBA,
```

```
    24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    PFD_MAIN_PLANE, 0, 0, 0, 0 };

 nPixelFormat = ChoosePixelFormat(hdc, &pfd); // Line (b)
 SetPixelFormat(hdc, nPixelFormat, &pfd); // Line (c)
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
         WPARAM wParam, LPARAM lParam) {
 HDC hdc;
 HGLRC hrc;

 switch (message) {
  case WM_CREATE:
   hdc = GetDC(hwnd); // Line (d)
   SetDCPixelFormat(hdc);
   hrc = wglCreateContext(hdc); // Line (e)
   wglMakeCurrent(hdc, hrc); // Line (f)
   break;
  case WM_PAINT:
   glClearColor(0.259, 0.259, 0.435, 1.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glBegin(GL_LINES);
    glVertex2f(-1.0, 0.0); glVertex2f(0.0, 0.0);
   glEnd();
   glFlush();
   ValidateRect(hwnd, NULL);
   break;
  case WM_CHAR:
   if (wParam == 'q') {
    wglMakeCurrent(hdc, NULL);
    wglDeleteContext(hrc);
    PostQuitMessage(0);
   }
   break;
  case WM_DESTROY:
   wglMakeCurrent(hdc, NULL);
   wglDeleteContext(hrc);
```

```
        PostQuitMessage(0);
        break;
      default:
        return DefWindowProc(hwnd, message, wParam, lParam);
      }
    return 0;
  }
```

예제 프로그램의 Line (d)에서는 우선 `GetDC(*)` 함수를 통하여 현재 사용하고 있는 디바이스 문맥에 대한 핸들, 즉 포인터를 변수 `hdc`에 저장한다. 디바이스 문맥은 한 순간에 여러 개가 존재할 수 있으므로, 항상 그래픽스에 관련된 함수를 호출할 때는 어떤 디바이스 문맥을 사용하는지를 분명히 하여야 한다. 다음 `SetDCPixelFormat(hdc);` 문장을 수행시켜 사용할 화소 형식을 결정한다. `SetDCPixelFormat(*)` 함수를 보면 우선 Line (a)에서 `PIXELFORMATDESCRIPTOR` 타입의 스트럭춰 변수 `pfd`에 OpenGL 시스템쪽에서 사용하고 싶은 화소 형식의 내용을 저장하고 있다. 이 예에서는 단순히 한 화소 당 24비트를 사용하는 RGBA 모델의 색깔 버퍼를 하나만 요구하고 있음을 쉽게 추측할 수가 있다. 만약 세 번째 필드가 `PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER`이고, 19번째 필드가 24라면 이는 더블 버퍼링을 하기 위하여 24비트의 RGBA 색깔 버퍼를 사용하고, 한 화소 당 24비트를 사용하는 깊이 버퍼를 사용하고 싶다는 것을 의미하게 될 텐데, 이에 대한 설명은 지면상 생략하도록 하겠다.

다음 Line (b)에서 `ChoosePixelFormat(hdc, &pfd);` 문장을 수행시키면 `pfd`에 담긴 정보가 GDI로 넘어가는데, GDI 쪽에서는 `hdc`가 가리키는 디바이스 문맥에 저장되어 있는 하드웨어의 성능에 따라 적절한 화소 형식을 결정하여 그에 대한 고유 번호를 넘겨준다. 만약 결정된 화소 형식에 만족을 한다면 다음 Line (c)의

SetPixelFormat(*) 함수를 통하여 이 화소 형식을 사용하겠다고 GDI에게 알려주고, 그 결과 디바이스 문맥의 OpenGL 관련 부분에 그 화소 형식에 대한 정보가 저장이 된다.

이렇게 SetDCPixelFormat(*) 함수의 수행이 끝나면 다시 WM_CREATE 메시지를 처리해주는 부분으로 돌아와 OpenGL 시스템과 연결을 하게 된다. 어떻게 보면 이제 윈도우 시스템과 OpenGL 시스템에서 사용할 그래픽스 시스템의 타입에 대한 합의를 본 상황이라 할 수 있다. 그런 다음 Line (e)에서 Wiggle 함수 wglCreateContext(*)를 사용하여 윈도우 시스템에서 OpenGL에 관련된 모든 정보를 저장할 렌더링 문맥(rendering context)을 생성하는데, OpenGL에서의 렌더링 문맥은 GDI의 디바이스 문맥에 해당한다고 생각하면 된다. 물론 이 함수의 수행을 통하여 렌더링 문맥을 생성할 때, 현재 사용하는 디바이스 문맥에 저장된 화소 형식에 관련된 정보를 사용한다. 디바이스 문맥의 경우처럼 여러 개의 렌더링 문맥을 번갈아 사용할 수 있으므로, 렌더링 문맥을 생성한 후 다음 문장에서 wglMakeCurrent(*) 함수를 사용하여 이 문맥을 현재 사용할 렌더링 문맥으로 지정을 한다. 이렇게 함으로써 OpenGL 시스템을 사용할 준비를 마치게 된다.

WM_PAINT 메시지를 처리하는 부분을 보면 여기서도 프로그램 예 1.1에서와 같이 화면에 그림을 그려주는 코드가 나오고 있다. 앞에 경우와 다른 점은 여기서는 이름이 gl로 시작하는 OpenGL 함수를 사용하여 화면의 배경을 (0.259, 0.259, 0.435) = ($\frac{66}{255}$, $\frac{66}{255}$, $\frac{111}{255}$)로 칠한 후, 흰 색 (1.0, 1.0, 1.0)의 선분을 점 (-1.0, 0.0)에서 점 (0.0, 0.0)까지 그리고 있다. 따라서 화면에는 앞의 예제 프로그램과 같은 그림이 그려진다. 마지막으로 WM_DESTROY 메시지를 처리할 때 wglDeleteContext(*) 함수를 통하여 이 프로그램에서 생성하여 사용하던 렌더링 문맥을 제거하는데, 이는 프로그램의 종료 시 불필요한 데이터를 시스템에 돌려주는 좋은 습관이라 하겠다.

# 제 4 절 GLUT 프로그래밍에 대한 소개

## 4.1 GLUT와 OpenGL

3절에서 살펴본 내용을 요약하면, 한 윈도우 시스템을 기반으로 하여 OpenGL을 통한 렌더링 프로그램을 작성하려면, 우선 1. 윈도우 시스템에서 제공하는 윈도우 함수들을 사용하여 바탕이 되는 윈도우 프로그램을 작성하고, 2. 윈도우 함수들이 제공하지 못하는 3차원 그래픽스 프로그래밍의 기능을 사용하기 위하여 OpenGL 시스템을 윈도우 시스템 확장 함수들을 사용하여 연결을 한 후, 3. 원하는 3차원 렌더링 작업을 OpenGL 함수들을 사용하여 수행하면 된다. 물론 이 경우 3차원 렌더링에 관련된 부분을 제외하면 기본적으로 윈도우 프로그램을 작성하는 것이라고 할 수가 있다.

윈도우 프로그래밍에 대한 경험이 많지 않은 프로그래머에게 있어 이는 OpenGL을 배워야 한다는 것과 함께 또 하나의 짐이 아닐 수가 없다. 이 책은 OpenGL 시스템을 이해하고 그것을 바탕으로 효과적인 실시간 3차원 렌더링 프로그램을 작성하는 법을 배우는 것이 주 목적이기 때문에, 가능한 한 윈도우 프로그래밍의 부담을 최소화하기 위하여 GLUT(OpenGL Utility Toolkit)라는 툴킷을 사용하고자 한다. GLUT는 마크 킬가드가 만든 하나의 프로그래밍 툴킷으로서, 윈도우 프로그래밍과 관련하여 많은 노력을 기울여 작성해야 하는 코드를 몇 개의 GLUT 함수를 사용하여 대치할 수 있게 해준다. 물론 GLUT를 사용하면 윈도우 시스템에서 제공하는 윈도우 함수들을 직접 사용할 때처럼 윈도우 시스템의 모든 인터페이스 기능을 사용할 수는 없지만, 윈도우 프로그래밍에 경험이 많지 않은 프로그래머의 입장에서는 무엇보다도 OpenGL 프로그래밍에 전념을 할 수가 있다는 장점이 있다. 또한 GLUT는 현재 여러 윈도우 시스템에 포팅이 되어 있어, 사용 윈도우 시스템을 바꾸

더라고 프로그램을 변경할 필요가 없다. 이 절에서는 독자들이 쉽게 OpenGL 프로그래밍을 시작하는데 필요한 최소한의 GLUT 기능에 대하여 알아보겠다. 특히 앞에서 설명한 바와 같이 윈도우 프로그래밍에 있어 중요한 두 가지 요소라 할 수 있는 윈도우의 초기화 및 이벤트 메시지 처리에 관련된 GLUT 함수들을 살펴보겠다. 자세한 내용은 관련 서적[6]이나 인터넷 상에서 쉽게 구할 수 있는 GLUT 문서를 참조하기 바란다.

## 4.2 윈도우에 대한 초기화

앞에서도 설명한 바와 같이 윈도우 프로그래밍을 하기 위하여 가장 먼저 해야 할 일은 화면에 띄우려고 하는 윈도우들에 대한 성질을 설정하고, 내부적으로 생성을 한 후, 화면에 매핑을 하는 것이라 할 수 있다. OpenGL관점에서 볼 때 윈도우를 생성하는 부분에서 사용하고자 하는 프레임 버퍼의 성능(capabilities)이 결정이 되기 때문에, 렌더링 목적에 적합한 초기화 작업을 해주어야 한다.

우선 void glutInit(int argc, char **argv); 함수는 GLUT 라이브러리를 초기화해주는 역할을 한다. 이 함수는 다른 어떤 GLUT 함수보다 먼저 부르는 것이 좋은데, GLUT를 사용하여 윈도우 프로그래밍을 하는데 필요한 초기화를 해주는 함수라 생각하면 된다. 이 함수의 인자로 커맨드 라인 옵션(command line option)에서 설정한 해당 윈도우 시스템 관련 명령을 받아들일 수가 있는데, 그에 대한 설명은 생략하기로 한다.

다음 glutInitDisplayMode(unsigned int mode);는 함수 이름이 나타내듯이 화면에 띄우고자하는 윈도우에 대한 프레임 버퍼의 성능, 즉 디스플레이 모드(display mode)를 결정한다. 앞에서도 설명한 바와 같이 윈도우 시스템과 OpenGL 시스템은

[6] M. Kilgard. *OpenGL Programming for the X Window System*, Addison-Wesley, 1996.

| 버퍼 종류 | GLUT 상수 | 용 도 |
|---|---|---|
| 색깔 버퍼 | GLUT_RGBA | RGBA 모드 선택(디폴트) |
| | GLUT_RGB | GLUT_RGBA와 같음 |
| | GLUT_INDEX | 색깔 인덱스 모드 선택 |
| | GLUT_ALPHA | 알파 채널에 대한 메모리 요청 |
| | GLUT_SINGLE | 싱글 버퍼 모드 선택(디폴트) |
| | GLUT_DOUBLE | 더블 버퍼 모드 선택 |
| | GLUT_STEREO | 스테레오 버퍼 요청 |
| | GLUT_LUMINANCE | 휘도 색깔 모델 지원 버퍼 요청 |
| | GLUT_MULTISAMPLING | 다중 샘플링 지원 버퍼 요청 |
| 기타 버퍼 | GLUT_ACCUM | 축적 버퍼 요청 |
| | GLUT_DEPTH | 깊이 버퍼 요청 |
| | GLUT_STENCIL | 스텐실 버퍼 요청 |

표 1.1: 디스플레이 모드에 관련된 GLUT 상수

프레임 버퍼에 관련된 부분을 공유하고 있기 때문에, 윈도우를 생성할 때 OpenGL 쪽에서 필요한 요구 사항을 기술해야 한다. 즉 프레임 버퍼에 속하는 여러 버퍼 중 색깔 버퍼는 한 화소 당 몇 픽셀을 사용하는지, RGBA 모드를 사용하는지 아니면 색깔 인덱스 모드를 사용하는지를 설정하여야 한다. 또한 그러한 색깔 버퍼를 한 개를 사용하는지 아니면 두 개를 사용하는지도 결정하여야 한다. 또한 깊이 버퍼링이라는 연산을 위해 특별히 깊이 버퍼를 사용하는지, 또한 스텐실 버퍼나 축적 버퍼를 사용하는지도 결정해야 한다.

어떻게 보면 서로 다른 두 개의 그래픽스 시스템이 서로 협정을 맺는 것으로 볼 수가 있는데, 버퍼들을 사용하고자 하는 OpenGL 시스템 쪽에서 필요한 버퍼의 성능을 요구하면, 윈도우 시스템 쪽에서 실제로 사용하는 하드웨어의 용량이 허용하는 한도 내에서 그러한 요구 사항을 최대로 만족을 시켜주는 버퍼를 할당해준다. 표 1.1에는 GLUT에서 제공하는 디스플레이 모드에 관련된 상수에 대하여 요약이 되어 있는데, 원하는 프레임 버퍼의 성능을 비트 매스크인 해당 상수에 대한 비트별 OR 연산을 통하여 기술을 한 후 이 함수의 인자로 사용하면 된다.

만약 glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);와 같이 함수를 호출한다면, 이는 색깔 버퍼에 대해서 RGBA 색깔 모델의 더블 버퍼를 사용하고, 또한 깊이 버퍼를 사용하고자 한다는 사실을 요청하는 것이 된다. 디스플레이 모드에 대한 디폴트는 GLUT_RGBA | GLUT_SINGLE 로서 RGBA 모델을 사용하는 싱글 버퍼 형태의 색깔 버퍼만 사용한다.

다음 윈도우에 대한 성질을 설정하는 함수로 glutInitWindowSize(int width, int height);와 glutInitWindowPosition(int x, int y);가 있다. 이 두 함수는 윈도우가 처음 생성될 때의 크기와 위치를 결정하는데 인자들의 의미는 분명하다. 이렇게 윈도우의 디스플레이 모드, 크기, 위치 등의 성질을 설정하고 난 후, glutCreateWindow(char *name); 함수를 통하여 윈도우를 내부적으로 생성해준다. 이 함수를 호출한다고 해서 실제로 화면에 윈도우가 나타나지는 않고, 아래에서 설명할 glutMainLoop() 함수를 수행하여야만 화면에 윈도우가 매핑이 된다. 이 함수의 인자(name)로 들어오는 스트링은 윈도우의 타이틀 바에 나타난다. 함수의 수행 결과 윈도우 시스템에서 정수 값을 리턴해 주는데, 이 값은 현재 생성한 윈도우에 고유한 숫자로서 추후 윈도우를 여러 개 사용할 경우 그 윈도우를 지칭하는데 사용된다.

프로그램 예 1.3에는 프로그램 예 1.1과 프로그램 예 1.2에 주어진 예제 프로그램을 GLUT 함수들을 사용하여 다시 작성한 예가 주어져 있다(예제 프로그램 1.C참조). main(*) 함수의 처음 네 개의 문장이 지금까지 설명한 윈도우의 초기화에 관련된 부분인데, glutInitDisplayMode(GLUT_RGBA); 문장은 52쪽의 프로그램 예 1.2의 윈도우 프로시져에서 WM_CREATE 메시지를 처리하는 코드에 해당한다는 사실을 명심하기 바란다.

**프로그램 예 1.3** GLUT를 사용한 OpenGL 프로그래밍 예 1.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <GL/glut.h>


void display(void) {
 glClearColor(0.259, 0.259, 0.435, 1.0); // Cornflower Blue
 glClear(GL_COLOR_BUFFER_BIT);
 glColor3f(1.0, 1.0, 1.0);
 glBegin(GL_LINES);
  glVertex2f(-1.0, 0.0); glVertex2f(0.0, 0.0);
 glEnd();
 glFlush();
}


void keyboard(unsigned char key, int x, int y) {
 switch (key) {
  case 'q':
   exit(0);
 }
}


void RegisterCallback(void) {
 glutDisplayFunc(display);
 glutKeyboardFunc(keyboard);
}


void main(int argc, char **argv) {
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_RGBA);
 glutInitWindowSize(500, 500);
 glutCreateWindow("My First OpenGL Code");
 RegisterCallback();
 glutMainLoop();
}
```

## 4.3 이벤트 메시지의 처리

자, 이제 GLUT를 사용하여 어떻게 이벤트 메시지를 처리하는 지에 대하여 알아보자. 전기한 바와 같이 이 예제 프로그램을 수행시키면 가로, 세로 500 화소인 윈도우가 화면에 뜨고, 사용자가 마우스를 그 윈도우 안으로 옮겨 'q' 키를 누르면 종료하게 된다. 이 프로그램과 관련하여 우선 사용자가 'q' 키를 눌렀을 때의 입력 이벤트를 처리해주어야 한다. 이 시점에서 이벤트 메세지의 처리 과정을 다시 생각해 보자. 사용자가 키보드를 눌렀을 때 그러한 사실이 윈도우 서버에게 보고가 되고, 서버는 그러한 사실을 이벤트 메시지를 통하여 우리가 작성하는 응용 프로그램에게 알려 준다. 메시지를 받은 응용 프로그램, 즉 클라이언트 프로그램에서는 서버로부터 받은 메세지가 어떤 메세지인지를 파악한 후 그에 대하여 적절한 처리를 해주게 된다. 그런데 이 예제 프로그램을 보면 `keyborad(*)` 함수가 이러한 키보드 이벤트를 처리해주는 코드인 것처럼 보이기는 하나, 앞에서의 윈도우 프로그램 예제에서 보았던 것과 같이 키보드 이벤트가 들어 왔을 때 이 함수를 호출하는 부분이 없는 것처럼 보인다.

GLUT 라이브러리는 이벤트 메시지의 처리에 대한 코딩을 단순화 시켜주기 위하여 컬백 함수(callback function)라는 도구를 사용한다. 컬백 함수는 윈도우 프로그램 수행 시 발생하는 여러 종류의 윈도우 및 입력 이벤트 중 관심이 있는 이벤트 각각에 대하여 어떠한 처리를 하고자 한다는 것을 기술해 주는 함수이다. 그러면 어떤 이벤트가 발생하였을 때 과연 그에 대한 컬백 함수가 어떠한 과정을 거쳐 수행이 될까? GLUT에서는 프로그래머가 관심이 있는 이벤트 각각에 대하여 1. 컬백 함수를 정의하고, 2. 등록(registration)을 하면, 서버로부터 해당 이벤트 메시지가 들어 왔을 때, 1. GLUT가 어떤 이벤트 메시지인지를 알아낸 후, 2. 그 이벤트에 대하여 컬백 함수가 등록이 되어 있으면 컬백 함수의 인자에 적절한 값을 설정하여

그 함수를 불러주는 방식을 취한다.

이 예제 프로그램에서 `keyboard(*)` 함수가 바로 키보드 이벤트를 처리해주는 컬백 함수의 역할을 하는데, `RegisterCallback()` 함수 안에서 `glutKeyboardFunc(*)` 함수를 통하여 이 함수가 키보드 컬백 함수(keyboard callback function)로 등록이 되고 있다. void glutKeyboardFunc(void (*func)(unsigned int key, int x, int y)); 함수는 인자로 들어오는 함수 func가 키보드 이벤트에 대한 처리 함수, 즉 키보드 컬백 함수라고 등록을 하는데 사용이 된다. 이 때 이 컬백 함수는 함수 정의에서 기술이 된 것과 같이 세 개의 인자를 받아들이는 함수로 정의되어야 한다. 키보드 컬백 함수의 첫번 째 인자 key는 사용자가 누른 키의 ASCII 코드 값을 가지고, 나머지 두 인자 (x, y)는 키를 눌렀을 때의 해당 윈도우 안에서의 마우스의 위치 값을 가진다.

한편 `RegisterCallback()` 함수를 보면 키보드 컬백 함수와 함께 또 다른 컬백 함수 `display()`가 `glutDisplayFunc(display);` 문장을 통하여 등록이 되고 있음을 볼 수 있다. void glutDisplayFunc(void (*func)(void)); 함수는 디스플레이 컬백 함수(display callback function)를 등록함을 목적으로 한다. 이 컬백 함수는 윈도우스의 WM_PAINT 메시지와 같이 윈도우의 내용을 일부라도 다시 그려야 할 필요가 생겼을 때 수행이 되는 함수이다. 따라서 디스플레이 컬백 함수의 내용이 OpenGL 프로그래밍과 직접적으로 관련이 있는데, `display()` 함수에서는 프로그램 예 1.2에서와 같이 OpenGL 함수를 사용하여 화면을 원하는 색깔로 칠한 후 선분을 그리고 있다.

지금까지 컬백 함수의 정의 및 등록을 통한 이벤트 메시지의 처리에 대하여 살펴보았는데, GLUT를 사용할 경우 이벤트 처리에 대한 코딩을 간단하게 할 수가 있음을 알 수가 있다. 이렇게 `main(*)` 함수에서 윈도우를 생성한 후, `RegisterCallback()` 함수를 불러 필요한 컬백 함수를 등록을 하면, 다음

`glutMainLoop();` 문장을 통하여 메시지 루프에 들어가게 된다. 즉 GLUT에서는 void glutMainLoop(void);를 통하여 이벤트 처리를 위한 무한 루프로 들어가, 서버로부터 계속해서 들어오는 이벤트 메시지 중 등록된 컬백 함수에 대한 이벤트만 처리해준다. 이 함수는 한 번만 수행을 시켜야 하고, 일단 수행을 시키면 다시 리턴을 하지 않기 때문에 프로그램을 종료하기 위해서는 프로그래머가 명시적으로 `exit(0);`과 같은 문장을 사용하여 무한 루프에서 벗어나와야 한다.

GLUT에서의 다른 컬백 함수의 사용에 대하여 알아보기 위하여 조금 더 복잡한 예제 프로그램을 살펴보자. 아래의 프로그램 예 1.4에 주어진 프로그램은 앞의 예보다 복잡한 기능을 가진다. 우선 `main(*)` 함수의 전체적인 구조는 앞의 프로그램과 같다. 아직 우리가 OpenGL 함수에 대해서 배우지 않았지만, 디스플레이 컬백 함수인 `display()` 함수의 코드를 보면 전역 변수 r, g, b를 사용하여 화면의 배경을 색깔 (r, g, b)로 칠한 후 선분을 한 개 그리고 있음을 알 수가 있다. 이 전역 변수들은 키보드 컬백 함수인 `keyboard(*)`에서 어떤 키를 눌렀는가에 따라 빨간색(‘`r`’), 초록색(‘`g`’), 파란색(‘`b`’) 중의 색깔로 적절히 선택이 된다. 다음 세 가지 경우 모두에 대해 `glutPostRedisplay();` 문장이 수행이 되고 있다. 과연 이 문장의 목적은 무엇일까? 이 프로그램을 수행을 시켜보면 알겠지만 색깔에 해당하는 키를 누를 때마다 윈도우의 배경 색깔이 즉시 바뀐다. 여기서 화면의 색깔은 디스플레이 컬백 함수인 `display()` 함수를 수행해야만 바뀌는데, 앞에서 말했듯이 이 함수는 윈도우의 내용을 일부라도 다시 그려야 할 상황이 발생이 되었을 때 호출이 된다.

그러면 서버는 사용자가 키보드를 눌렀을 때 과연 이러한 상황을 윈도우의 내용을 업데이트 해주어야 하는 상황으로 인식을 할까? 답은 그렇지가 않다. 윈도우 시스템의 입장에서 보면 이는 단순히 사용자가 키보드를 통하여 원하는 정보를 입력하는 이벤트가 발생한 것이기 때문에, 꼭 윈도우의 내용을 다시 그려야할

필요가 생긴 것이라고는 할 수 없다. 그러한 필요는 응용 프로그램을 작성하는 프로그래머 입장에서 생긴 것이기 때문에, 서버는 그러한 사실을 알 도리가 없고, 따라서 응용 프로그램, 즉 클라이언트가 서버에게 윈도우의 내용을 업데이트 해야 할 필요가 생겼다고 알려주어야 한다. 즉 프로그래머가 명시적으로 서버에게 윈도우에 그림을 다시 그려야하는 상황이 발생하였으니, 필요한 이벤트 메시지를 발생하여 디스플레이 컬백 함수가 호출이 되도록 요청을 하여야 한다. GLUT에서는 void glutPostRedisplay(void); 함수를 사용하여 그러한 요청을 할 수 있다. 따라서 `keyboard(*)` 함수 내에서 색깔을 적절히 바꾼 후 이 함수를 불러 화면의 내용을 다시 그려주는 디스플레이 함수가 수행이 되도록 하고 있다. 물론 이 함수를 부르는 대신 컬백 함수인 `display()` 함수를 직접 부를 수도 있을 것이다. 그러나 이는 윈도우 관리나 입력 처리 등 모든 윈도우 시스템 관련 상항을 관리하고 있는 윈도우 서버를 통하여 적절한 컬백 함수를 부르는 것이 윈도우 시스템의 작동 원리에 더 충실하다고 할 수가 있다.

**프로그램 예 1.4** GLUT를 사용한 OpenGL 프로그래밍 예 2.

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>

short rightbuttonpressed = 0;
double r = 1.0, g = 0.0, b = 0.0;

void display(void) {
  glClearColor(r, g, b, 1.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 1.0, 1.0);
  glBegin(GL_LINES);
    glVertex2f(-1.0, 0.0); glVertex2f(0.0, 0.0);
  glEnd();
```

```
 glFlush();
}

void keyboard(unsigned char key, int x, int y) {
 switch (key) {
  case 'r':
   r = 1.0; g = b = 0.0;
   glutPostRedisplay();
   break;
  case 'g':
   g = 1.0; r = b = 0.0;
   glutPostRedisplay();
   break;
  case 'b':
   b = 1.0; r = g = 0.0;
   glutPostRedisplay();
   break;
  case 'q':
   exit(0);
 }
}
void mousepress(int button, int state, int x, int y) {
 if ((button == GLUT_LEFT_BUTTON) && (state == GLUT_DOWN))
  printf("*** The left mouse button was pressed at (%d,
%d).\n", x, y);
 else if ((button == GLUT_RIGHT_BUTTON) && (state ==
GLUT_DOWN))
  rightbuttonpressed = 1;
 else if ((button == GLUT_RIGHT_BUTTON) && (state ==
GLUT_UP))
  rightbuttonpressed = 0;
}

void mousemove(int x, int y) {
 if (rightbuttonpressed)
  printf("$$$ The right mouse button is now at (%d,
%d).\n", x, y);
}
```

```
void reshape(int width, int height) {
  printf("### The new window size is %dx%d.\n",
          width, height);
}

void RegisterCallback(void) {
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  glutMouseFunc(mousepress);
  glutMotionFunc(mousemove);
  glutReshapeFunc(reshape);
}

void main(int argc, char **argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA);
  glutInitWindowSize(500, 500);
  glutCreateWindow("My Second OpenGL Code");
  RegisterCallback();
  glutMainLoop();
}
```

한편 `RegisterCallback()` 함수에서는 `glutMouseFunc(mousepress);` 문장을 통하여 마우스 컬백 함수(mouse callback function)라는 또 다른 종류의 컬백 함수를 등록하고 있다. 함수 void glutMouseFunc(void (*func)(int button, int state, int x, int y));는 사용자가 마우스의 버튼을 눌렀거나, 누른 상태에서 다시 원래의 상태로 띠었을 때 발생하는 마우스 이벤트를 처리해주는 컬백 함수 func를 등록하는데 사용이 된다. 이 함수에는 네 개의 인자가 있는데, 우선 button은 비트 매스크인 상수 GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON 중 한 값을 가지는데 그 의미는 분명하다. 다음 state는 GLUT_UP 또는 GLUT_DOWM 중 하나의 값을 가지는데, 그 이름이 나타내듯이 각각 사용자가 마우스 버튼에서 손을

누른 상태에서 손을 띠었는지, 아니면 새롭게 마우스 버튼을 눌렀는지를 나타낸다. 마지막 두 개의 인자 (x, y)는 앞에서와 같이 그러한 이벤트가 일어났을 때의 마우스의 위치를 의미한다.

다음에 등록하고 있는 함수는 움직임 컬백 함수(motion callback function)이다. 함수 void glutMotionFunc(void (*func)(int x, int y));는 사용자가 마우스의 버튼을 하나 이상 누른 상태에서 움직였을 때, 그러한 움직임 이벤트를 처리해주는 컬백 함수 func를 등록 해준다. 이 때 움직임 컬백 함수의 두 개의 인자는 움직임이 발생하였을 때의 마우스의 위치 (x, y)를 나타낸다.

이 예제 프로그램에서 마우스 컬백 함수인 `mousepress(*)`와 움직임 컬백 함수인 `mousemove(*)`를 보면, 마우스를 윈도우 안으로 옮긴 후 왼쪽 마우스 버튼을 누를 때마다 그 때의 마우스 위치의 좌표가 출력이 되고, 오른쪽 마우스를 누른 상태에서 움직이면 마우스가 움직이는 경로의 좌표가 출력이 되도록 프로그래밍이 되어있음을 이해할 수 있을 것이다.

마지막으로 `glutReshapeFunc(reshape);` 문장에서 등록이 되는 컬백 함수는 리쉐이프 컬백 함수(reshape callback function)으로서, void glutReshapeFunc(void (*func)(int width, int height)); 함수에 의해 등록이 된다. 이 컬백 함수 func는 윈도우의 크기가 바뀌었을 때 발생이 되는 리쉐이프 이벤트 메시지를 처리하는데 사용이 된다. 또한 윈도우가 처음 생성이 된 후 그에 대하여 첫 번째 디스플레이 컬백 함수가 호출되기 직전에 한 번 호출이 된다. 이 함수의 인자 width와 height는 윈도우의 새로운 크기 값을 가지는데, 이 예제 프로그램에서는 윈도우의 크기가 변할 때마다 새로운 크기가 화면에 출력이 된다.

지금까지 1. 윈도우 성질의 설정 및 생성, 2. 컬백 함수의 등록, 3. 무한 메시지 루프로의 입장으로 구성된 간단한 GLUT 프로그램에 대하여 살펴보았다. 이러한 구

조는 비단 GLUT 프로그래밍뿐만 아니라 일반적인 윈도우 프로그래밍의 기본 틀을 이루고 있는데, GLUT에서는 지금까지 설명한 함수들을 포함하여 다양한 윈도우 프로그래밍 함수들을 제공한다. 이 책에서는 GLUT에서 대한 설명을 여기서 마치고, 새로운 GLUT 함수가 나올 때마다 설명을 하도록 하겠다.

# 제 2 장

# OpenGL의 뷰잉 모델

## 제 1 절 3차원 뷰잉과 OpenGL 기하 파이프라인

비단 OpenGL뿐만 아니라 Direct3D나 RenderMan과 같은 대부분의 3차원 그래픽스 렌더링을 위한 API를 통하여 프로그램을 작성하기 위하여 가장 중요한 것은 바로 사용 API의 렌더링 과정을 정의하는 렌더링 파이프라인 중 기하 변환에 관한 부분을 정확하게 이해하는 것이다. 사실 한 번 정확하게 이해를 하면 어려울 것이 없는 부분이나, 3차원 그래픽스 프로그래밍을 처음 접하는 사람한테 있어 가장 혼란스러울 수 있는 점을 꼽으라면 기하 물체를 다루는 좌표 변환이 아닌가 한다.

렌더링은 앞에서도 기술한 바와 같이 카메라로 사진을 촬영하는 과정에 해당한다고 할 수가 있다. 렌더링 파이프라인은 여러 단계로 이루어져 있는데, 이 중 하나인 기하 파이프라인(geometry pipeline)의 목적은 가상의 세상에 카메라의 위치와 방향을 고정하고 구도를 잡았을 때, 다면체 모델로 표현된 개개의 물체가 화면의 어디에 떨어질 것인가를 결정하는 것이다. 따라서 이 과정에서는 3차원 공간인 가상의 세상에서 정의된 물체의 각 꼭지점이, 2차원 공간인 화면의 어느 지점에 나

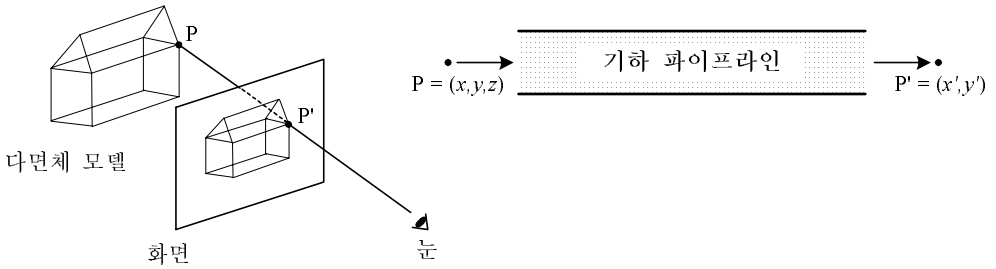



그림 2.1: 3차원 뷰잉과 기하 파이프라인

타날 지를 결정해주는, 3차원 공간에서 2차원 공간으로의 기하학적인 좌표 변환에 대한 계산이 수행이 된다. 이러한 계산 과정을 3차원 뷰잉(3D viewing)이라 하는데, 이는 다면체 모델을 구성하고 있는 각 꼭지점의 화면에서의 위치를 결정하는 과정을 뜻한다(그림 2.1). 3차원 뷰잉은 기하 공간에 존재하는 물체 들에 대한 기하 계산으로서, 뒤에서 다룰 조명 계산, 래스터화 계산, 텍스춰 매핑 계산, 은면 제거 계산 등의 다른 계산 과정과 함께 렌더링 파이프라인의 중요한 부분을 이루게 된다.

본 2장에서는 OpenGL에서 사용되는 기하 파이프라인에서의 계산 과정을 정확하게 이해하고, 응용 예를 통하여 프로그래밍 기법을 익히도록 하겠다. 이 장을 읽으면서 OpenGL 프로그래밍의 중추라 할 수 있는 107쪽의 그림 2.15의 기하 파이프라인 구조를 항상 염두에 두기 바란다. 이 장의 구성은 다음과 같다. 2절에서는 3차원 컴퓨터 그래픽스 분야에서 필요로 하는 기본적인 기하학적인 개념에 대하여 알아보겠다. 다음 3절에서 OpenGL 시스템의 기하 파이프라인에 대하여 상세히 알아본 후, 마지막 4절에서 OpenGL 프로그래밍을 통하여 기하 파이프라인에 대한 이해를 높이도록 하겠다.

# 제 2 절 기하 프리미티브, 좌표계, 그리고 기하 변환

## 2.1 기하 프리미티브

3차원 그래픽스 프로그래밍을 통하여 렌더링을 하기 위해서는 무엇보다도 렌더링의 대상이 되는 가상의 세상을 구축하여야 한다고 하였다. 즉 기하 모델링이라는 과정을 통하여 세상에 존재하는 가상의 물체들, 예를 들어 집, 자동차, 사람 등에 대한 기하 모델을 만들어 그래픽스 프로그래밍에 적합한 형태로 표현을 하여야 한다. 컴퓨터 그래픽스에서 기하 물체를 표현하는데 있어 가장 기본이 되는 요소를 기하 프리미티브(geometric primitive)라 한다. 한 물체를 표현하기 위해서는 여러 부류의 프리미티브를 생각할 수 있으나, 실시간 렌더링을 기본 목표로 하는 OpenGL 프로그래밍에 있어서는 일반적으로 점, 선분, 다각형, 다면체 등의 선형 프리미티브(linear primitive)를 사용한다.

다면체(polyhedron)는 잘 알다시피 다각형(polygon)으로 구성되어 있고, 다각형은 선분(line segment)으로 구성되어 있으며[1], 선분은 다시 꼭지점(vertex)으로 구성되어 있다. 이들이 3차원 물체를 (근사적으로) 표현하는데 있어 가장 기본이 되는 기하 프리미티브로서, 이 네 가지 요소로 기하 물체를 표현하는 것을 다면체 모델로 모델링을 한다고 한다. 이들의 공통적인 특징은 수학적으로 1차식으로 표현될 수 있는 선형적인 요소라는 사실이다. 기하 물체를 수학적으로 표현하는 데 있어 2차 이상의 식을 사용할 경우 더 정확하게 물체를 표현할 수 있는 반면 그러한 모델을 처리하는 시간이 증가하게 된다. 따라서 표현 능력의 한계에도 불구하고 특히 실시간 컴퓨터 그래픽스에 있어 선형적이라는 도구와 계산 방식은 그 효율성을 인하여 매우 중요하게 쓰이고 있다.

[1] 또는 변(edge)이라고도 함.

다면체 모델을 사용할 경우 3차원 물체는 결국 다각형들의 집합으로 표현되므로 한 물체를 그린다는 것은 그러한 다각형을 하나씩 순서대로 그리는 것으로 생각할 수 있다. 다각형이건 선분이건 결국 꼭지점으로 구성이 되어 있으므로 기하 물체를 그리는데 있어 가장 기본이 되는 연산은 바로 꼭지점의 좌표를 나열하는 것이다. 물론 그래픽스 프로그래밍에 있어 좌표뿐만 아니라 꼭지점에 연관된 여러 속성들로 같이 기술하게 되는데, 여기서는 일단 좌표의 기술에 대해서만 고려를 하자. OpenGL에서도 여타 그래픽스 API에서와 같이 꼭지점의 좌표를 기술할 수 있는 함수를 제공하는데 바로 void glVertex3f(GLfloat x, GLfloat y, GLfloat z); 함수가 그러한 역할을 담당한다. 이 함수는 좌표가 $(x, y, z)$인 꼭지점을 나열하는데 사용을 한다.

**잠깐 2.1** 실제로 OpenGL에서는 꼭지점의 좌표를 기술하기 위하여 다양한 형태의 함수를 제공하는데, 그러한 함수들을 void glVertex{234}{sifd}[v](TYPE coords);와 같이 간결하게 나타낼 수 있다. 여기서 우선 2, 3, 4 중 하나를 선택을 할 수 있는데 이는 함수 인자로 좌표 값이 몇 개가 들어오는지를 나타낸다. 곧 설명을 하겠지만 OpenGL에서 3차원 공간의 좌표는 $(x, y, z, w)$와 같이 표현을 한다. 2차원 공간에서의 좌표를 기술하려면 2를 선택을 하면 되는데, 이 때 함수의 인자로 2차원 좌표 $x$와 $y$를 기술하면 된다. 만약 3을 선택하면 위에서와 같이 3차원 공간에서의 좌표를 기술하게 된다. 마지막으로 4를 선택하면 $w$ 좌표를 포함한 네 개의 좌표가 이 함수의 인자로 들어오게 된다.

또한 s, i, f, d 중 하나를 선택해야 하는데, 이것은 함수의 인자로 들어오는 좌표 값의 타입을 결정한다. 이 네 가지는 각각 16비트 정수 타입인 GLshort, 32비트 정수 타입인 GLint, 32비트 부동 소수점 타입인 GLfloat, 그리고 64비트 부동 소수점 타입인 GLdouble을 나타낸다. 마지막으로 v는 포인터를 통하여 좌표를 기술할 수

있도록 해주는데, 좌표 값들을 배열에 연달아 저장을 해 놓고 첫 번째 좌표 값의 주소를 함수의 인자로 사용하면 된다. 다음은 이러한 타입의 함수를 사용하는 예로서 그 의미는 명확하다.

```
GLdouble vertex0[4] = {1.0, -2.5, 0.0, 1.0};
     ⋮
glVertex4dv(vertex0);
```

이와 같이 OpenGL에서는 같은 목적을 가지는 함수에 대하여 다양한 타입에 대하여 선택을 할 수 있도록 해준다. 이 책은 OpenGL을 통하여 3차원 그래픽스 프로그래밍에 대한 이해를 돕는 방향으로 초점이 맞추어 있으므로, 매번 그러한 함수들에 대하여 자세히 설명하기 보다는 앞에서와 같이 하나의 경우에 대하여 설명을 하거나 glVertex*()와 같이 기술하도록 하겠다.

## 2.2 OpenGL의 10가지 기하 프리미티브

기하 물체를 구성하는 프리미티브들을 기술하는 방식에는 여러 가지가 있을 수 있다. 어떠한 방법을 사용하건 항상 glVertex*(*) 함수를 통해서 기술을 해야 하는데, 이 함수는 항상 void glBegin(GLenum mode); 함수와 void glEnd(void); 함수 사이에서만 호출되어야 한다. 다시 말해서 OpenGL에서는 기하 물체에 대한 꼭지점 정보가 항상 이 두 함수 사이에서 기술되어야 한다. glBegin(*) 함수의 인자로 나열형 타입(GLenum)의 상수를 사용하는데, 이 상수는 꼭지점을 연속하여 나열할 때 그것들이 어떤 종류의 기하 프리미티브를 형성하는지를 결정한다.

OpenGL에서는 10가지의 형태의 방식으로 기하 프리미티브를 기술하는 방법을 제공한다. GL_POINTS는 꼭지점을 그리기 위하여, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP 등은 선분을 그리는데 사용한다. 한편 다각형은 GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, GL_PO-

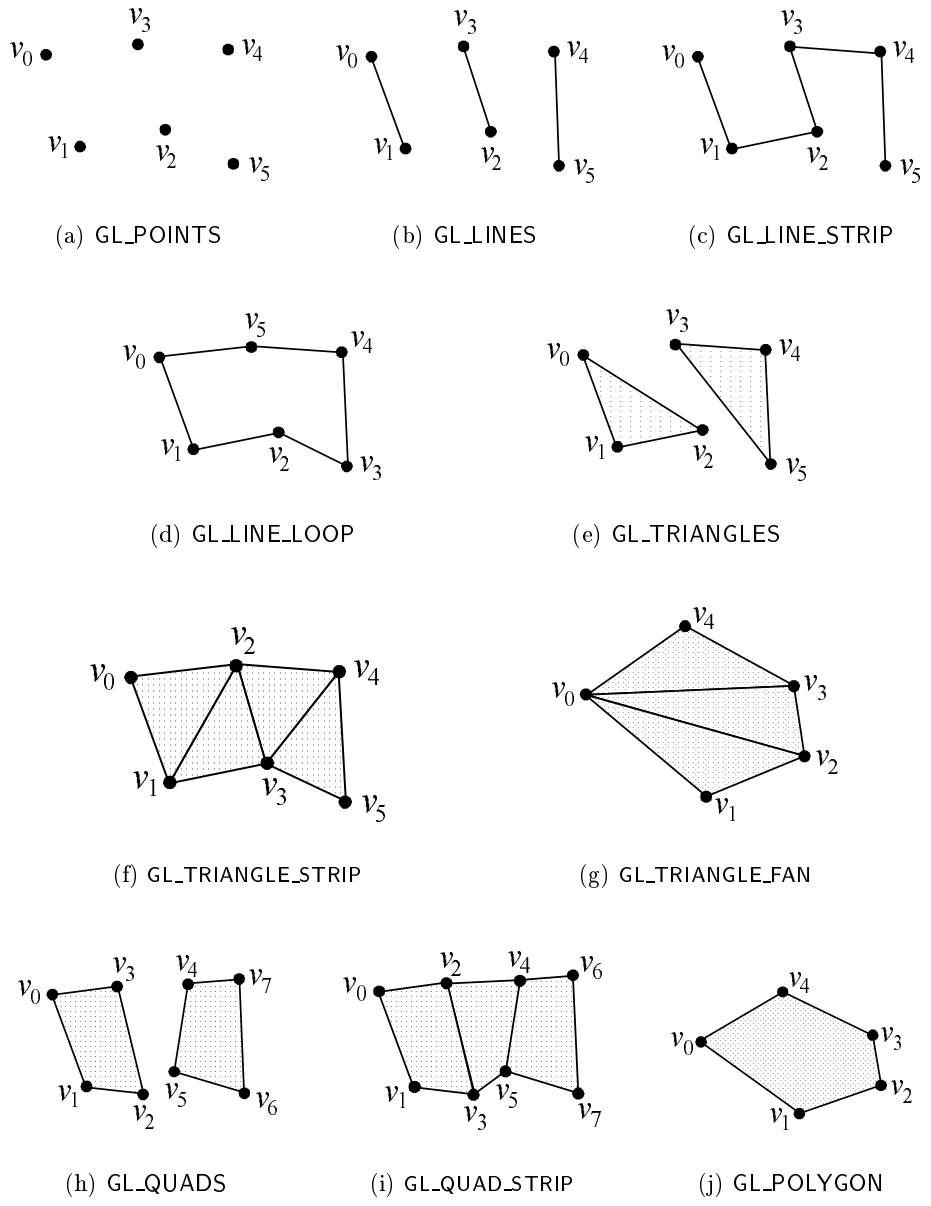
LYGON 등을 통하여 그릴 수가 있다.

그림 2.2에는 OpenGL에서 제공하는 10가지 형태의 기술 방법이 요약되어 있는데, glBegin(*) 함수와 glEnd() 함수 사이에 glVertex*(*) 함수를 사용하여 나열한 꼭지점을 다음과 같이 $v_0, v_1, v_2, \cdots$라 할 때, glBegin(*) 함수의 인자에 따라 꼭지점들이 어떻게 연결이 되는지를 알 수 있다.

```
glBegin(GL_LINES);
  glVertex3f(3.0, 2.5, -1.5); // v0
  glVertex3f(3.0, 2.1, -1.2); // v1
  glVertex3f(2.9, 1.5, -1.0); // v2
  glVertex3f(2.5, 1.5, -1.5); // v3
      ⋮
glEnd();
```

몇 가지 예를 들어보면, GL_LINES의 경우 나열된 꼭지점들을 두 개씩 짝지어 선분을 구성한다. GL_POLYGON의 경우에는 나열된 꼭지점들을 순서대로 연결한 후 마지막 꼭지점과 첫 번째 꼭지점을 연결하여 다각형을 구성한다. GL_POLYGON은 GL_LINE_LOOP와 유사하나 후자는 단순히 꼭지점들을 선분으로 연결하는 반면, 전자의 경우에는 다각형, 즉 그 내부까지 채워지는 전혀 다른 형태의 기하 프리미티브를 만들어 준다. 또한 이름에 STRIP이 들어가는 종류들은 프리미티브들을 효율적으로 기술하기 위한 것들인데, 그 의미들 또한 분명하다. 120쪽의 프로그램 예제에서의 `draw_teapot()` 함수에서 물체를 그리는 예를 참고하기 바란다.

**잠깐 2.2** 아무래도 가장 중요한 기하 프리미티브는 다각형이라 할 수 있는데 이에 대하여 약간 고려해야할 사항이 있다. 다각형은 그 성질에 따라 여러 가지 부류로 분류를 할 수 있다. OpenGL에서는 다각형이라는 기하 프리미티브의 형태에 제한을 가하고 있는데, 그러한 요구 사항들을 만족하는 다각형에 대해서만 정확한 렌더링을 보장한다. 그러면 OpenGL에서는 다각형에 대하여 어떠한 제한을 가할까? 첫




(a) GL_POINTS (b) GL_LINES (c) GL_LINE_STRIP

(d) GL_LINE_LOOP (e) GL_TRIANGLES

(f) GL_TRIANGLE_STRIP (g) GL_TRIANGLE_FAN

(h) GL_QUADS (i) GL_QUAD_STRIP (j) GL_POLYGON

그림 2.2: OpenGL 지원 기하 프리미티브

째로, 한 다각형을 구성하는 선분들이 서로 교차를 하면 안 된다. 둘째로, 볼록 다각형(convex polygon)을 사용하여야 한다. 수학적으로 말하면 다각형의 내부의 임의의 두 점을 선택하여 선분으로 연결하였을 때 그 선분상의 모든 점이 그 다각형에 포함되면, 그러한 다각형을 볼록 다각형이라 한다. 따라서 이 두 조건을 만족하는 다각형은 단순하게 생긴 다각형임을 알 수가 있다. 그 외에 한 가지 더 제한을 가하는데, 3차원 공간에서 정의된 다각형의 꼭지점이 모두 한 평면 위에 존재해야 한다는 사실이다. 그 이유는 만약 그렇지 않는 다각형을 2차원 평면에 투영을 하였을 때, 위의 두 조건을 만족시키지 못하는, 즉 선분들이 평면 상에서 서로 교차하거나 볼록이 아닌 다각형을 생성할 수 있기 때문이다.

그러면 위의 세 가지 조건을 항상 만족시키는 다각형의 부류는 무엇일까? 바로 삼각형으로서 이러한 이유로 다면체 모델을 만들 때 삼각형이 가장 자주 사용이 되고는 한다. 다른 그래픽스 관련 API에서는 항상 이러한 세 가지 조건을 요구하지는 않는다. 예를 들어 X 윈도스나 마이크로 소프트의 윈도우스 환경에서 사용되는 2차원 그래픽스 함수들은 위의 조건을 만족시키지 못하는 다각형들도 정확하게 그려주도록 노력을 하는데, 이를 위하여 다각형 모양에 대한 여러 경우를 모두 처리하여야 하기 때문에 렌더링 비용이 증가한다고 할 수 있다. 반면 OpenGL에서는 3차원 기법을 사용하여 렌더링을 할 때 계산 비용을 줄이기 위하여, 여러 가지의 특수한 경우를 고려할 필요가 없는 단순한 형태의 다각형으로 가정하고 렌더링을 하게 된다.

## 2.3 3차원 좌표계

3차원 공간에 존재하는 점의 좌표를 정확하게 기술하기 위해서는 그것의 기준이 되는 좌표계(coordinate system)가 필요하다. OpenGL은 3차원 그래픽스를 위한 렌더링 시스템이기 때문에 기본적으로 모든 좌표는 $(x, y, z)$와 같이 3차원 공간의 점으로 기술이 된다. 좀 더 정확하게 말하면 OpenGL에서는 내부적으로 3차원 공간에서의 점을 $(x, y, z, w)$와 같이 네 개의 좌표 값을 사용하여 표현하는데, 이에 관해서는 아래의 동차 좌표계에 대한 절에서 설명을 하도록 하겠다. OpenGL에서도 2차원 그래픽스를 위한 좌표 $(x, y)$를 사용할 수 있는데, 사실 이는 $z$ 좌표를 0으로 놓은, 즉 $(x, y, 0)$인 형태의 3차원 점으로 저장되어 계산이 수행된다. 따라서 glVertex2f(0.5, 1.0);과 같이 2차원 좌표를 설정을 하면 내부적으로 glVertex3f(0.5, 1.0, 0.0);과 같은 함수를 호출한 것과 같은 효과가 있게 된다.

점에 대한 좌표는 열 벡터로 표현을 하는 것이 여러 가지 경우에 더 자연스럽기 때문에, 이 책에서는 기본적으로 점을$(x\ y\ z)^t$나 $(x\ y\ z\ w)^t$와 같이 열벡터로 나타내겠다. 하지만 문맥에 따라서 문제가 없으면 $(x, y, z)$ 또는 $(x, y, z, w)$와 같이 기술하도록 하겠다.

### 2.3.1 오른손 좌표계와 왼손 좌표계

일반적으로 2차원 좌표계에서는 $x$축은 오른쪽으로, $y$축은 위쪽으로 값이 증가하도록 설정이 된다. X 윈도우스나 마이크로 소프트의 윈도우스 환경에서는 $y$축은 아래쪽으로 값이 증가 하나 OpenGL에서는 위로 갈수록 값이 증가한다. 3차원의 좌표계의 경우 $x$축과 $y$축 외에 좌표축이 하나 더 있는데 바로 이 $z$축의 방향을 정확하게 규정하지 않으면 문제가 생길 수 있다.

그림 2.3에 있는 두 개의 좌표계를 보면 비슷하게 보이나 사실은 서로 다른 좌표계를 나타내고 있다. 왼쪽의 좌표계는 $x$축과 $y$축을 자연스럽게 종이에 그렸을 때 $z$축이 종이로부터 바깥쪽으로 나올수록 값이 증가하도록 설정이 되어 있다. 반면에 오른쪽에 있는 좌표계는 $z$축이 반대 방향, 즉 종이의 안쪽으로 들어갈 수록 값이 증가하도록 설정이 되어 있다. 전자의 경우를 오른손 좌표계(right-handed coordinate system)라 하고, 후자의 경우를 왼손 좌표계(left-handed coordinate system)라 부른다. 이렇게 이름을 붙인 이유는 해당 좌표계에서 이름에 해당하는 손의 엄지 손가락을 제외한 네 손가락을 $x$축에서 $y$축으로 감을 때 자연스럽게 엄지 손가락이 향하는 방향이 양의 $z$ 방향이 되기 때문이다.

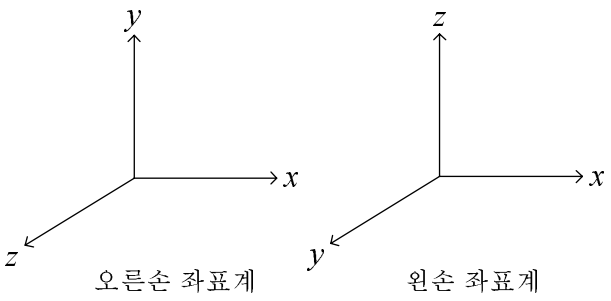



그림 2.3: 3차원 좌표계

실제로 두 좌표계의 유일한 차이는 $z$축의 방향이 반대라는 점이다. 따라서 한 좌표계에서 표현된 좌표를 다른 좌표계로 바꾸려면 $z$값의 부호만 바꾸어 주면 된다. 이 두 좌표계는 모두 널리 쓰이고 있는데, OpenGL에서는 오른손 좌표계를 기본으로 하고 있다. $x$축과 $y$축이 각각 컴퓨터 화면의 오른쪽과 위쪽 방향을 나타낸다고 할 때, 그리고 화면에서 안쪽으로 멀리 떨어질수록 깊이 값에 해당하는 $z$값이 증가한다고 하면 이런 경우에는 왼손 좌표계가 더 자연스럽다고 할 수 있다. 따라서 전에는 기하 파이프라인에서 끝 쪽으로 갈수록 왼손 좌표계가 자연스럽게 사용되었다. 전기한 바와 같이 OpenGL에서는 기본적으로 오른손 좌표계를 사용하나, SGI사의 GL에서 진화한 OpenGL에는 왼손 좌표계의 잔재가 남아있는데 이에 대해서는 뒤에서 간략히 설명하도록 하겠다.

## 2.4 동차 좌표계

OpenGL 시스템에서는 한 색깔을 $(r, g, b)$가 아니라 $(r, g, b, a)$로 표현하듯이 3차원 공간의 좌표는 $(x, y, z)$가 아니라 $(x, y, z, w)$로 표현을 한다. 3차원 좌표를 이렇게 나타내는 것을 동차 좌표계(homogeneous coordinate system)를 사용하여 표현하였다고 한다. 이 절에서는 3차원 그래픽스 프로그래밍의 기본 개념을 이해하는데 있어 최소한으로 필요한 동차 좌표계에 관한 사실에 대하여 살펴보도록 하자. 좀 더 쉽게 동차 좌표계를 이해하기 위하여 2차원 좌표계를 예를 들어 생각하자. 우리가 오래 전부터 배워온 유클리드 기하학(Euclid geometry)에서는 2차원 평면에서의 한 점을 $(x, y)$로 표현을 한다. 이러한 점들로 이루어진 공간을 2차원 유클리드 공간(Euclidean space)라 하는데 이는 2차원 기하학의 이론을 전개하는 기본 공간으로 사용되어 왔다. 유클리드 공간과 밀접한 관계가 있는 공간으로 아핀 공간(affine space)을 생각할 수 있는데, 후자는 점과 벡터로 이루어진 공간으로, 그리고 전자는 아핀 공간에 기하학 이론을 전개하는데 반드시 필요한 거리의 개념이 들어간 공간이라고 생각하면 된다.

이 두 공간은 기하학에서 기본이 되는 공간으로서 종종 불완전한 공간으로 인식이 되어왔다. 예를 들어 2차원 평면에서 서로 다른 두 직선은 한 점에서 만난다는 사실은 유클리드 기하학에서 기본이 되는 사실이나 여기에는 예외가 존재한다. 또한 아핀 공간의 평면 상의 점과 벡터는 모두 $(x, y)$ 형태로 표현이 되나, 그 의미가 다르게 해석이 되는 것도 부자연스러운 점이라고 할 수가 있다. 좀 더 완전한 공간에서의 기하학에 대한 시도로서 수학자들은 투영 기하학(projective geometry)에 대하여 연구를 하여 왔다. 여기서는 유클리드 공간보다 더 완전한 공간인 투영 공간(projective space)을 사용하는데, 이 공간에서는 2차원 공간의 점을 나타내기 위

하여 $(x, y)$ 형태가 아니라 $(X, Y, Z)$ 형태의 동차 좌표계를 사용한다[2]. 두 개의 자유도를 가지는 2차원 공간의 점을 세 개의 값으로 나타낼 경우 여기에는 중복성이 존재하게 된다. 2차원 투영 공간에서의 두 점 $(X, Y, W)$와 $(X', Y', W')$은 다음과 같은 조건을 만족하는 경우에, 그리고 오직 그럴 경우에만 같은 점을 나타낸다.

$$(X, Y, W) \equiv \alpha(X', Y', W') = (\alpha X', \alpha Y', \alpha W'), \alpha \neq 0$$

즉 동차 좌표계를 사용하여 표현한 어떤 좌표에 0이 아닌 상수를 곱해 다른 좌표를 만들 수 있으면 두 좌표는 투영 공간에서 같은 점을 나타내게 된다. 예를 들어 $(2, -3, 1)$, $(4, -6, 2)$, $(-1, 1.5, -0.5)$는 모두 같은 점을 나타낸다. 이러한 투영 공간에서의 동차 좌표 $(X, Y, W)$와 유클리드 공간 또는 아핀 공간에서의 좌표 $(x, y)$와는 어떠한 관계가 있을까? $W$가 0이 아닌 경우 $(X, Y, W)$와 $(\frac{X}{W}, \frac{Y}{W}, 1)$은 같은 점을 나타내는데, 바로 이 때 앞의 두 개의 좌표 값이 $(X, Y, W)$에 대응되는 유클리드 공간에서의 점의 좌표가 된다. 즉 $W$가 0이 아닐 때 $(X, Y, W) \equiv (x, y) = (\frac{X}{W}, \frac{Y}{W})$이고, 이러한 사실은 유클리드 공간 또는 아핀 공간의 모든 점이 투영 공간에 포함이 된다는 사실을 암시한다. 그러면 과연 투영 공간은 유클리드 공간에 존재하지 않는 점을 포함하고 있을까? 그리고 있다면 어떠한 점들일까?

이러한 궁금증을 해결하기 위하여 $W$가 0인 경우를 다음과 같은 예를 들어 이해해보자. 두 개의 서로 다른 직선 $L_1 : 2x - 4y + 4 = 0$과 $L_2 : 2x - 4y + 8 = 0$이 주어졌을 때 유클리드 공간에서는 이 두 직선은 서로 만나지 않으므로 두 개의 서로 다른 직선이 한 점에서 만난다는 사실의 예외의 경우에 해당한다(그림 2.4). 과연 투영 공간에서도 그러한지를 알기 위해 두 공간의 좌표 값의 관계인 $x = \frac{X}{W}$, $y = \frac{Y}{W}$를 $L_1$과 $L_2$에 대입하여 정리하면 두 직선은 투영 공간에서 $L_1 : 2X - 4Y + 4W = 0$과

[2] 이 때 $X$, $Y$, $W$ 모두 동시에 0이 되는 경우는 제외한다.

$L_2 : 2X - 4Y + 8W = 0$과 같이 나타낼 수 있다. 이 둘의 교점을 계산하기 위하여 $L_2$에서 $L_1$을 빼면 $4W = 0$, 즉 $W = 0$이 되고, 따라서 투영 공간에서의 교점의 좌표는 $(X, \frac{X}{2}, 0)$ (단 $X \neq 0$)이 된다. 즉 유클리드 공간에서는 안 만나는, 정확하게 이야기하면 만나지 않는다고 배워온 평행한 두 직선이 투영 공간에선 다른 직선들과 마찬가지로 한 점에서 만나게 되는 것이다. $(x, y)$ 형태로 좌표를 나타내는 유클리드 공간에서 $x = 0$이거나 $y = 0$인 점이 다른 점들과 다르게 취급되지 않듯이, 투영 공간에서도 $W = 0$인 점들이 다른 점들과 다르게 취급이 되지 않기 때문에 위의 두 직선도 투영 공간의 한 점에서 만난다고 생각을 하는 것이다.

따라서 투영 공간은 유클리드 공간에 존재하지 않는 점들이 더 있는 상대적으로 보다 완전한 공간이라 할 수 있다. 그러면 유클리드 공간에 더 익숙한 우리들에게 있어 투영 공간에서의 교점 $(X, \frac{X}{2}, 0)$을 어떻게 이해를 할 것인가? 동차 좌표계의 정의에 따르면 이 점은 $(1, 0.5, 0)$ 또는 $(-1, -0.5, 0)$과 같은 점을 나타내는데, 이는 유클리드 공간에서 두 직선이 뻗은 양 방향을 가리키는 두 벡터 $(1, 0.5)$와 $(-1, -0.5)$와 밀접한 관계가 있어 보인다. 불완전한 유클리드 공간의 관점에서는 평행한 두 직선이 이 벡터들이 가리키는 방향으로 아주 멀리 있는, 즉 무한대에 있는, 점에서 만난다고 추측을 할 수 있다. 이러한 유추는 투영 공간에는 존재하나 유클리드 공간에는 존재하지 않는 $W$가 0인 점 $(X, Y, 0)$을 유클리드 공간에서 어떻게 상상을 할 지에 대한 단서를 제공한다. 즉 이 점은 유클리드 공간의 벡터 $(X, Y)$가 가리키는 방향으로 아주 멀리 있는 점을 나타낸다고 생각하면 되는데, 실제로 이러한 점을 무한대 점(point at infinity)라 부른

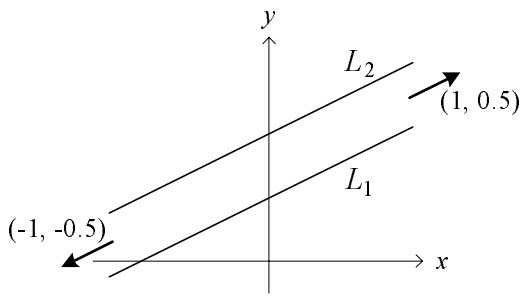



그림 2.4: 평행한 두 직선

다.

한 가지 재미있는 것은 투영 공간에서 $W$가 0이 아닌 점들과 0인 점들은 유클리드 공간에서 각각 점과 벡터에 자연스럽게 대응이 된다는 점이다. 따라서 투영 공간은 유클리드 공간 또는 아핀 공간에 존재하는 점들에다가 방향을 나타내는 모든 벡터들을 더한 공간이라고 볼 수 있다. 유클리드 공간에서 좌표 $(x, y)$는 점의 위치를 나타낼 수도 있고 방향을 나타낼 수도 있다. 앞에서도 언급한 바와 같이 동차 좌표 $(X, Y, W)$의 경우에는 $W$가 0이건 아니건 투영 공간에서는 별 특별한 차이가 없기 때문에 이 좌표는 점이라는 하나의 개념을 나타내고, 따라서 투영 공간에서는 유클리드 공간에서와 같은 점과 벡터의 구별이 없어지게 된다.

3차원의 경우에는 좌표 값이 하나 더 늘어난다는 점 외에는 위에서 기술한 사실들이 자연스럽게 확장이 된다. 즉 투영 공간에서의 점은 $(X, Y, Z, W)$와 같이 표현이 되고 $W$가 0이 아닐 경우 $(\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W})$가 이에 대응되는 유클리드 공간에서의 점의 좌표가 된다. 동차 좌표계를 사용하면 다음 절에서 설명할 아핀 변환을 행렬과 벡터의 형태로 간결하게 표현을 할 수 있다. 또한 비아핀 변환인 원근 변환을 사용할 때에는 동차 좌표계의 사용이 필수적인데 이를 통하여 그러한 변환의 기하학적인 의미를 쉽게 이해를 할 수가 있다. 또한 예를 들어 2차원 유클리드 공간에서의 평면 곡선을 NURBS(nonuniform rational B-spline)처럼 $C(t) = (\frac{X(t)}{W(t)}, \frac{Y(t)}{W(t)})$와 같이 나타낼 경우에도 이를 투영 공간에서의 곡선 $\hat{C}(t) = (X(t), Y(t), W(t))$으로 생각을 하면 그에 대한 기하학적인 성질을 더 잘 이해를 할 수 있다.

앞에서 언급한 바와 같이 우리는 오래 전부터 유클리드 공간에 익숙하여 왔고, 또한 컴퓨터 그래픽스 전반에 걸쳐 유클리드 공간이 사용되고 있다. 특히 가상의 세상을 위한 좌표계에서 물체를 표현할 때나 화면에 물체를 나타낼 때 모두 유클리드 공간을 사용한다. 즉 3차원 그래픽스 렌더링 파이프라인의 입력 부분과 출력 부분

모두에서 유클리드 공간이 사용되고 있는 반면 그 중간 부분에 투영 공간이 잠시 나타나게 된다. 당분간 3차원 공간의 점은 $(x, y, z)$나 $(x, y, z, 1)$와 같이 표현을 하도록 하겠다.

## 2.5 기하 변환

OpenGL의 렌더링 파이프라인에서 기하 파이프라인의 주된 목적은 가상의 세상에서 정의된 기하 물체들을 컴퓨터 화면의 어디에 그릴 것인가를 결정하는 것이라 하였다. 이를 위해서 각 물체의 다면체 모델에 대한 기하 변환(geometric transformation)을 필요로 하게 된다. 우리가 여기서 사용할 기하 변환은 물체의 평면성 또는 선형성을 유지하여 주는 성질을 가진다. 즉 예를 들어 한 직선을 변환한다고 했을 때 그 결과도 직선이 되는, 그러한 성질을 가지는 기하 변환만 고려할 것이다. 따라서 다면체 모델은 점, 선분, 다각형들로 이루어져 있는데, 결국 점에 대한 변환만 하면 다른 기하 프리미티브들도 간단하게 변환을 할 수 있다. 본 절에서는 3절에서 OpenGL의 기하 파이프라인에서 사용되는 기하 변환을 이해하는데 필수적인 기하 변환에 대한 사실들에 대하여 살펴보도록 하겠다.

### 2.5.1 아핀 변환

3차원 공간의 한 점 $(x, y, z)$을 $(x', y', z')$로 기하 변환을 한다고 할 때, 이는 수학적으로 $(x', y', z') = f(x, y, z)$와 같이 좌표간의 사상(mapping), 즉 함수 $f : R^3 \longrightarrow R^3$로 표현할 수 있다. 여기서 $f$는 함수의 성질만 만족한다면 어떤 것이든 상관이 없다. 즉 $f$는 수식으로 간결하게 기술될 수도 있고, 또는 복잡한 절차에 의해(procedurally) 정의될 수도 있을 것이다. 수식에 의하여 간결하게 표현이 되는 경우에도 사실 상당히 방대한 부류의 좌표 변환이 가능하지만, 그래픽스 분야에서는 보통 아핀 변

환(affine transformation)이라고 하는 선형적인 기하 변환이 중요하게 사용이 되고 있다.

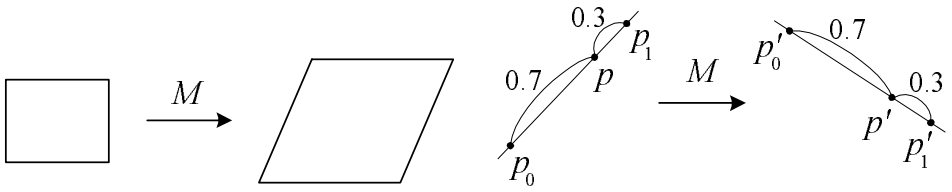
주어진 점 $p = (x\ y\ z)^t$를 아래와 같은 방식의 계산을 통하여 점 $p' = (x'\ y'\ z')^t$의 좌표를 구할 때 이러한 기하 변환을 3차원 아핀 변환이라 한다:

$$p' = Ap + v, \quad A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \in R^{3\times 3}, \quad v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \in R^3$$

기하 변환을 위하여 점의 좌표를 3차원 공간의 열 벡터로 표현을 할 때, 3차원 아핀 변환은 임의의 3행 3열 행렬 $A$와 점 벡터를 곱한 후 다시 임의의 3차원 공간 벡터 $v$를 더하여 새로운 좌표를 계산하는 형태의 기하 변환인데 간단한 계산을 통하여 다음과 같이 간결하게 나타낼 수가 있다.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & v_1 \\ a_{21} & a_{22} & a_{23} & v_2 \\ a_{31} & a_{32} & a_{33} & v_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

여기서 자연스럽게 유클리드 공간의 점 $p$와 $p'$에 대한 동차 좌표가 나타나는데, 이를 통하여 아핀 변환을 단순히 4행 4열 행렬과 4개의 원소를 가지는 열 벡터의 곱으로 간결하게 표현할 수 있다. 따라서 앞으로 3차원 공간의 점 $p$를 나타낼 때 $(x\ y\ z)^t$ 또는 $(x\ y\ z\ 1)^t$를 의미하게 될 것인데, 어떤 것일지는 문맥에 따라 적절하게 판단하기 바란다. 위의 4행 4열 행렬을 $M$이라 하고 동차 좌표를 사용한다고 했을 때 아핀 변환은 $p' = Mp$와 같이 표현된다. 여기서 주의할 점은 $M$이 임의의 4행




(a) 평행성의 유지 (b) 비율의 보존

그림 2.5: 아핀 변환의 성질

4열 행렬이 아니라 네 번째 행이 항상 (0 0 0 1)이라는 제약을 가진다는 사실이다. 즉 3차원 아핀 변환에 해당하는 행렬은 16개가 아니라 12개의 자유도를 가지는데, 이 12개의 변수가 어떤 값을 가지는가에 따라 여러 형태의 유용한 기하 변환을 표현하게 된다.

**잠깐 2.3** 아핀 변환의 가장 큰 특징중의 하나는 기하 물체의 선형성을 유지시켜 준다는 사실이다. 즉 변환 전에 직선적 또는 평면적이었던 것이 변환 후에도 동일한 성질을 계속해서 만족하게 된다. 또 다른 특징으로는 평행성을 유지시켜 준다는 사실이다. 그림 2.5(a)에서처럼 물체의 구성 요소의 길이나 각도가 변해도 변환 전에 평행하였던 것은 변환 후에도 평행하다는 점이다. 또한 아핀 변환은 점들간의 비율을 보존시켜 준다. 이 것의 의미는 그림 2.5(b)를 보면 쉽게 알 수 있는데 이러한 사실들은 어렵지 않게 증명할 수가 있다. 예를 들어 그림 2.5(b)에서의 $p$는 $p = 0.3p_0 + 0.7p_1$과 같이 표현 가능하다. 이 점에 대하여 아핀 변환을 수행하면 $p' = Ap + v = A(0.3p_0 + 0.7p_1) + v = 0.3Ap_0 + 0.7Ap_1 + v = 0.3(Ap_0 + v) + 0.7(Ap_1 + v) = 0.3p'_0 + 0.7p'_1$과 같이 되어 변환 후에도 점들간의 비율이 유지가 됨을 알 수 있다. 아핀 변환은 그 계산이 단순할 뿐만 아니라 평면적인 물체를 다루는데 적합한 성질들이 있어 다면체 모델로 표현된 기하 물체 변환에 유용하게 쓰이고 있다.

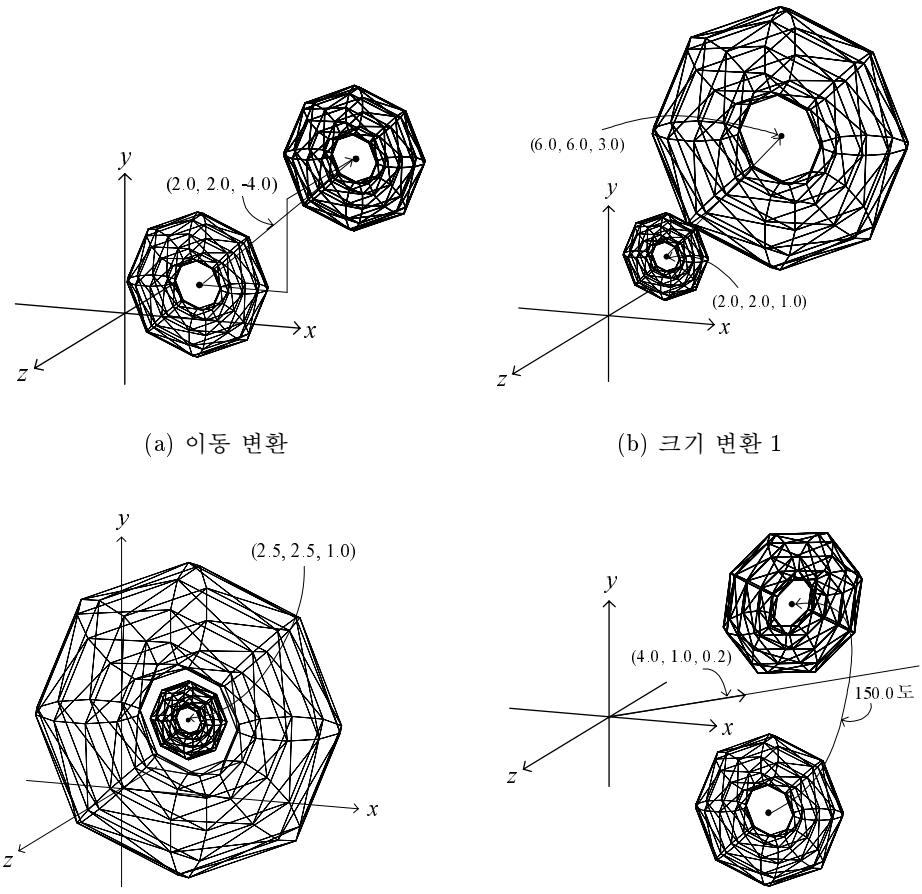
지금까지는 아핀 변환의 전반적인 성질에 대하여 간략히 살펴보았는데 지금부터는 이러한 변환이 컴퓨터 그래픽스 분야에서 어떻게 쓰이는지를 구체적으로 알아보자. 아핀 변환의 대표적인 예로 이동 변환, 크기 변환, 회전 변환, 쉬어링, 반사등을 들 수가 있는데, 여기서는 앞의 세 가지 변환에 대하여 생각해보겠다.

**이동 변환**

이동 변환(translation)은 그 이름이 의미하듯이 3차원 공간에서 기하 물체를 이동 시켜 그 위치를 바꾸어 주는 변환에 해당한다. 그림 2.6(a)는 도우넛 형태의 물체를 $x$축 방향으로 2만큼, $y$축 방향으로 2만큼, 그리고 $z$축 방향으로 $-4$만큼 이동시키는 예를 보여주고 있다. 이처럼 어떤 물체를 이동시키기 위해서는 그 물체를 구성하는 각 꼭지점의 좌표에 해당하는 값을 더해주면 된다. 아핀 변환 행렬 $M$이 그림 2.7(a)의 $T(t_x, t_y, t_z)$의 형태로 표현이 될 때, 이는 주어진 점을 $x$, $y$, $z$축 각 방향으로 $t_x$, $t_y$, $t_z$만큼 이동을 시키는 변환에 해당한다. 이러한 사실은 실제로 행렬 $T(t_x, t_y, t_z)$에 점의 좌표$(x, y, z)$를 곱해보면 변환된 점$(x', y', z')$의 좌표 값이 $x' = x + t_x$, $y' = y + t_y$, $z' = z + t_z$가 되어 점이 이동이 됨을 알 수가 있다.

**크기 변환**

$M$이 그림 2.7(b)에서와 같이 $S(s_x, s_y, s_z)$의 형태를 취할 때, 이는 주어진 기하 물체의 크기를 $x$, $y$, $z$축 방향으로 각각 $s_x$, $s_y$, $s_z$배만큼 변화시켜 주는 크기 변환(scaling)에 해당한다. $s_x$, $s_y$, $s_z$ 값이 1보다 클 때는 해당 방향으로 물체가 커지게 되고, 1보다 작을 때는 물체의 크기가 작아지게 된다. $s_x = s_y = s_z$일 때는 물체의 모양이 그대로 유지되면서 크기만 변하고, 그렇게 못할 때는 물체가 찌그러지




(a) 이동 변환

(b) 크기 변환 1

(c) 크기 변환 2

(d) 회전 변환

그림 2.6: 기본 변환의 예

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(a) 이동변환: $T(t_x, t_y, t_z)$

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(b) 크기변환: $S(s_x, s_y, s_z)$

$$\begin{bmatrix} \bar{n}_x^2(1-c)+c & \bar{n}_x\bar{n}_y(1-c)-\bar{n}_z s & \bar{n}_z\bar{n}_x(1-c)+\bar{n}_y s & 0 \\ \bar{n}_x\bar{n}_y(1-c)+\bar{n}_z s & \bar{n}_y^2(1-c)+c & \bar{n}_y\bar{n}_z(1-c)-\bar{n}_x s & 0 \\ \bar{n}_z\bar{n}_x(1-c)-\bar{n}_y s & \bar{n}_y\bar{n}_z(1-c)+\bar{n}_x s & \bar{n}_z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$c = cos(\alpha),\ s = sin(\alpha),\ (\bar{n}_x, \bar{n}_y, \bar{n}_z) = \frac{(n_x, n_y, n_z)}{|(n_x, n_y, n_z)|}$$

(c) 회전변환: $R(\alpha, n_x, n_y, n_z)$

그림 2.7: 이동, 크기, 회전 변환 행렬

게 된다. 전자와 같은 크기 변환을 등방성 크기 변환(isotropic scaling)이라 하고, 후자의 경우를 비등방성 크기 변환(anisotropic scaling)이라 한다.

한 가지 주의해야 할 것은 크기 변환은 주어진 물체의 크기가 원점을 중심으로 하여 사방으로 변하게 된다는 점이다. 그림 2.6(b)에는 도우넛 형태의 물체를 3배 확대하는 상황을 보여주고 있는데, 물체의 크기가 변하면서 물체의 중심이 원점에서 멀어지고 있는 것을 알 수가 있다. 그림 2.6(c)에는 물체의 중심을 유지하면서 크기만 변하는 상황을 보여주고 있는데, 이는 $S(s_x, s_y, s_z)$ 행렬에 의한 기본 크기 변환과는 전혀 다른 변환으로서 이러한 변환에 대해서는 뒤에 설명을 하겠다.

### 회전 변환

세 가지의 기본 변환의 마지막 예로서 회전 변환(rotation)을 생각해보자. 이 변환은 앞의 두 변환에 비해서 다루기가 그리 수월하지가 않다. 따라서 오래 전부터 복잡한 회전 변환을 쉽게 다룰 수 있는 기법들이 연구가 되어 왔다. 3차원 공간에서 한 벡터 $n = (n_x\ n_y\ n_z)^t$가 주어졌을 때 원점을 지나고 이 벡터의 방향으로 뻗어 나

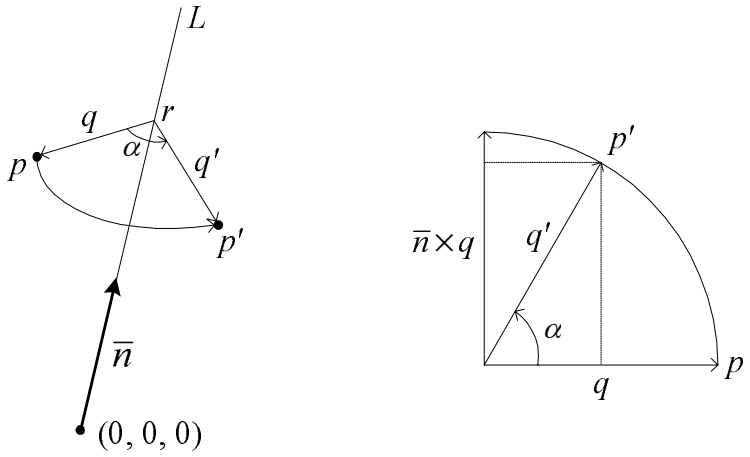



그림 2.8: 회전 변환의 유도

가는 직선 $L$을 생각할 수 있다(그림 2.8의 왼쪽). 주어진 점을 이 직선 $L$ 둘레로 $\alpha$도 만큼 회전시키려 할 경우 어느 방향으로 돌려야 하는지를 정확하게 지정을 하여야 한다. $L$은 $n$에 의해 방향이 결정되는데 OpenGL에서는 기본적으로 오른손 좌표계를 사용하므로 오른손 엄지 손가락을 이 벡터의 방향으로 향하게 했을 때 자연스럽게 휘감는 방향이 양의 회전 방향으로 정의된다. 이 때 $L$을 위에서 보면 반시계 방향으로 회전함을 알 수 있다. 이러한 회전 변환에 대한 변환 행렬 $R(\alpha, n_x, n_y, n_z)$이 그림 2.7(c)에 주어져 있다. 여기서 $c$와 $s$는 각각 각도 $\alpha$에 대한 사인과 코사인 값이고, $\bar{n}_x$, $\bar{n}_y$, $\bar{n}_z$는 $n$을 정규화를 해준, 즉 길이를 1로 만들어 준 벡터의 좌표 값에 해당한다. 회전 변환의 특수한 예로 $x$, $y$, $z$ 축 각 방향으로의 회전을 생각할 수 있는데, 그 때의 변환 행렬 $R_x(\alpha)$, $R_y(\alpha)$, $R_z(\alpha)$에 대하여 생각해보기 바란다. 그림 2.6(d)는 회전 변환의 예를 보여주고 있다.

### 2.5.2 회전 변환 행렬의 유도

다음으로 넘어가기 전에 여기서 회전 변환 행렬이 어떻게 얻어진 것인지를 유도하여 보자. 그림 2.8의 왼쪽에 원점을 지나고 단위 벡터 $\bar{n} = (\bar{n}_x\ \bar{n}_y\ \bar{n}_z)^t$의 방향으로

뻗은 직선 $L$ 둘레로 한 점 $p = (x\ y\ z)^t$를 각도 $\alpha$ 만큼 회전하여 점 $p' = (x'\ y'\ z')^t$을 얻는 과정이 도시되어 있다[3]. 여기서 이 점은 직선 $L$에 수직인 평면 위에서 회전을 하는데, 이 평면과 $L$이 만나는 점을 $r$이라 하고, $r$에서 $p$와 $p'$을 향한 벡터를 각각 $q$와 $q'$이라 하면, 변환된 점 $p'$의 좌표는 다음과 같이 나타낼 수가 있다.

$$p' = r + q' \tag{2.1}$$

점 $r$을 지나고 $L$에 수직인 평면을 위에서 바라본 모습이 그림 2.8의 오른쪽에 주어져 있는데, 여기서 $q$에 수직인 벡터는 $\bar{n} \times q$의 값을 가진다. 따라서 $q'$은 다음과 같이 정리를 할 수가 있다.

$$q' \quad = \quad q\cos\alpha + (\bar{n} \times q)\sin\alpha \tag{2.2}$$

이 때 벡터의 내적 $\cdot$ 와 외적 $\times$의 기본적인 성질을 사용하면, $r = (\bar{n}\cdot p)\bar{n}$, $q = p - r = p - (\bar{n}\cdot p)\bar{n}$, 그리고 $\bar{n} \times q = \bar{n} \times (p - (\bar{n}\cdot p)\bar{n}) = \bar{n} \times p$가 됨을 알 수가 있다. 이러한 사실을 식 (2.1)과 (2.2)에 대입하면, 다음과 같은 결과를 얻게 된다.

$$\begin{aligned} p' &= r + q' \\ &= (\bar{n}\cdot p)\bar{n} + (p - (\bar{n}\cdot p)\bar{n})\cos\alpha + (\bar{n} \times p)\sin\alpha \end{aligned}$$

이렇게 $p'$과 $p$의 관계를 유도하였는데, 지금 우리가 원하는 것은 두 점간의 함수 관계를 $p' = Rp$와 같이 행렬과 벡터의 곱으로 표현해주는 3행 3열 행렬 $R$의 내용

[3]회전 변환은 4행 4열의 아핀 변환 행렬의 왼쪽, 위에 위치한 3행 3열의 부행렬에만 영향을 받는다. 따라서 이 절에서는 편의상 그러한 행렬 $R$에 대하여 세 개의 원소를 가지는 열 벡터 $p$를 곱해 $p'$을 구하는 형태의 변환을 고려하겠다.

을 구하는 것이다. 위 식의 $(\bar{n} \cdot p)\bar{n}$ 부분을 잠시 생각해보면, 이 값은 $(\bar{n} \times \bar{n}^t) \cdot p$와 같이 3행 3열 행렬 $\bar{n} \times \bar{n}^t$와 점 $p$의 곱으로 표현할 수 있음을 알 수가 있다. 다음 $(p - (\bar{n} \cdot p)\bar{n}) \cos\alpha$는 $\{\cos\alpha(I - \bar{n} \times \bar{n}^t)\} \cdot p$와 같이(여기서 $I$는 3행 3열의 단위 행렬), 그리고 $S$를 다음과 같은 3행 3열의 행렬로 정의할 때,

$$S = \begin{bmatrix} 0 & -\bar{n}_z & \bar{n}_y \\ \bar{n}_z & 0 & -\bar{n}_x \\ -\bar{n}_y & \bar{n}_x & 0 \end{bmatrix}$$

$(\bar{n} \times p) \sin\alpha$는 $\{\sin\alpha\, S\} \cdot p$와 같이 됨을 알 수가 있다. 따라서 $R = \bar{n} \times \bar{n}^t + \cos\alpha(I - \bar{n} \times \bar{n}^t) + \sin\alpha\, S$라 놓으면, 이는 그림 2.7(c)의 $R(\alpha, n_x, n_y, n_z)$의 회전 변환에 해당하는 3행 3열 부행렬과 일치하는데, 바로 이 행렬이 우리가 구하고자 하는 회전 변환에 대한 행렬이 된다. □

### 2.5.3 행렬과 기하 변환

지금까지 아핀 변환이라는 기하학적인 개념과 행렬과 벡터의 곱셈이라는 행렬 계산과의 관계에 대하여 기본적인 사실들을 살펴보았다. 어떤 기하 변환이 주어졌을 때 거기에는 항상 특정 행렬이 대응이 된다. 따라서 기하 변환에 관한 성질들을 행렬을 통하여 쉽게 유추할 수 있다. 즉 행렬 계산의 여러 성질들은 기하 변환의 여러 성질을 이해하는데 기초가 된다고 할 수 있다.

예를 들어 어떤 변환을 한 후 그것을 되돌리고 싶을 때, 즉 역변환(inverse transformation)을 하고자 하면, 이러한 변환은 원래의 변환에 대한 행렬 $M$의 역행렬 $M^{-1}$을 사용하여 수행할 수 있다. 앞에서 설명한 세 가지의 기본 변환의 의미를 생

각해보면, 역변환에 대한 행렬은 다음과 같이 됨을 어렵지 않게 알 수 있다.

$$\begin{aligned} T^{-1}(t_x, t_y, t_z) &= T(-t_x, -t_y, -t_z) \\ S^{-1}(s_x, s_y, s_z) &= S(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}) \\ R^{-1}(\alpha, n_x, n_y, n_z) &= R(-\alpha, n_x, n_y, n_z) \end{aligned}$$

만약 어떤 변환에 대한 행렬 $M$의 계수(rank)가 4보다 작다면, 예를 들어 $M = S(1.0, 2.0, 0.0)$일 때, 그러한 행렬의 역행렬이 존재하지 않으므로 그 변환은 원래의 상태로 되돌릴 수가 없게 된다. 다음 두 절에서는 이러한 행렬과 기하 변환과의 관계에 대하여 더 알아보도록 하겠다.

### 2.5.4 강체 변환과 직교 행렬

특별하게 취급되는 부류의 기하 변환으로 강체 변환(rigid body transformation)을 들 수가 있다. 강체 변환이란 기하 물체를 변환할 때 변환 후에도 물체의 구성 요소의 길이나 각도가 보존이 되는 변환, 즉 물체의 형태가 그대로 보존이 되는 변환을 뜻한다. 예를 들어 비행기가 날아가는 애니메이션을 제작할 때 당연히 비행기의 모양이 반복되는 변환 후에도 보존이 되어야 하기 때문에 여기서 사용되는 변환은 강체 변환이어야 한다. 이러한 강체 변환에 관한 성질도 행렬의 성질을 통하여 쉽게 설명할 수가 있는데, 이를 위하여 선형대수에 관련된 몇 가지 기본적인 개념을 이해하여야 한다.

$u = (u_1\ u_2\ \cdots\ u_n)^t$과 $v = (v_1\ v_2\ \cdots\ v_n)^t$을 $n$차원 공간의 두 벡터라 하자. 벡터의 성질에 대한 이론을 전개하기 위하여 가장 기본이 되는 개념 중의 하나가 바로

내적(inner product)이다. 두 벡터의 내적은 다음과 같이 정의가 된다.

$$u \cdot v \equiv u_1 v_1 + u_2 v_2 + \cdots + u_n v_n = |u||v| \cos\theta$$

여기서 $\theta$는 두 벡터간의 각도를 나타내는데, 내적은 무엇보다도 벡터의 길이를 정의하는데 사용된다. 즉 주어진 벡터에 대하여 자기 자신과의 내적을 취하면 $u \cdot u = |u|^2$과 같이 되므로, 벡터의 길이, 특히 유클리드 노옴(Euclidean norm)을 정의하는 기초가 된다. 또한 내적을 사용하면 두 벡터가 얼마나 유사한 방향을 가리키고 있는지를 판단할 수 있다. 즉 $\frac{u \cdot v}{|u||v|}$ 값을 계산하면 이는 $\cos\theta$에 해당하므로, 이 값이 1이면 두 벡터가 같은 방향, 그리고 -1이면 정 반대 방향을 가리킴을 알 수 있다. 또한 이 값이 0이라면 두 벡터는 수직으로 만나게 된다.

$m$개의 $n$차원 공간 벡터 $v_1, v_2, \cdots, v_m$이 주어졌을 때 서로 다른 벡터들간의 내적이 모두 0이면($v_i \cdot v_j = 0 \; \forall \; i \neq j$), 이 벡터들이 직교 (orthogonal)한다고 한다. 또한 직교하는 벡터들의 길이가 모두 1일 때($v_i \cdot v_j = 0 \; \forall \; i \neq j$, $v_i \cdot v_i = 1 \; \forall \; i, j$), 이러한 벡터들을 정규 직교(orthonormal)한다고 한다. 이러한 정의의 기하학적인 의미를 이해하기 위하여 3차원 공간의 세 개의 벡터를 생각해보자. 이 벡터들이 직교한다는 것은 3차원 공간에서 서로 수직인 벡터임을 의미하며, 정규 직교한다고 하는 것은 서로 수직이면서 각 벡터들의 길이가 1인 단위 벡터임을 뜻한다. 따라서 $n$차원 공간에서 동시에 직교할 수 있는 벡터의 개수는 최대 $n$임을 알 수 있는데 이것이 바로 공간의 차원임을 의미한다.

주어진 $n$행 $m$열 행렬 $M$이 $M^t M = I$일 때, 즉 자신과 자신의 전치 행렬을 곱한 결과가 단위 행렬이 되면 이러한 행렬을 직교 행렬(orthogonal matrix)이라고 한다. 어떤 행렬이 직교 행렬이 되기 위해서는 그것을 구성하는 $m$개의 열 벡터들이 정규

직교하여야 한다[4]. 어떻게 보면 이러한 행렬을 정규 직교 행렬이라 부르는 것이 더 자연스럽다고 할 수 있지만 관례상 직교 행렬이라 부른다.

직교 행렬의 정의를 보면 $n = m$인 직교 행렬 $M$의 역행렬은 자신의 전치 행렬임을 알 수 있다($M^{-1} = M^t$). 일반적으로 임의의 행렬의 역행렬을 계산하는 것은 매우 비용이 많이 드는 어려운 계산이기 때문에 직교 행렬의 이러한 성질은 매우 바람직하다고 할 수가 있다. 또 직교 행렬의 좋은 성질 중의 하나는 이 행렬을 통하여 선형 변환을 할 때 노옴, 즉 크기가 보존이 된다는 사실인데, 이에 대한 수학적인 고찰은 이 책의 범위를 벗어나므로 생략하기로 하고, 여기서는 직교 행렬과 물체의 크기와 각도를 보존하여주는 강체 변환과 밀접한 관계가 있다는 사실을 명심하기 바란다.

이제 아핀 변환의 정의 $p' = Ap + v$를 다시 생각해보면, 여기서 $v$ 벡터는 단순히 물체의 위치를 바꾸어주는 역할을 하기 때문에 물체의 형태를 변화시키지는 못한다. 즉 변환 후에 물체의 모양이 바뀌었다면 이는 행렬 $A$에 기인할 것이다. 역으로 말하면 이 행렬이 직교 행렬이라면 이러한 변환은 물체의 형태를 보존하게 될 것이다. 따라서 임의의 3차원 공간에서의 아핀 변환에 해당하는 4행 4열 행렬 $M$이 주어졌을 때 $M$의 왼쪽 위의 3행 3열에 해당하는 부행렬이 정규 직교 행렬일 경우 이 변환은 바로 강체 변환이 된다. 앞에서 살펴본 세 가지 기본 아핀 변환 중 이동 변환과 회전 변환 행렬이 이러한 성질을 만족함을 알 수 있다. 이는 직관적으로 생각을 해도 이 두 변환이 물체의 형태를 그대로 유지시키는 성질이 있음을 쉽게 알 수가 있다. 반면에 크기 변환 행렬은 $s_x = s_y = s_z = 1$인 경우를 제외하고는 직교 행렬이 아닌데, 당연히 이러한 경우에는 물체의 크기가 변함을 알 수 있다. 임의의 강체 변환은 항상 이동 변환과 회전 변환을 사용하여 표현할 수 있다. 즉 강체 변환에

[4] 이런 것을 정규 직교계(orthonormal system)를 이룬다고 한다.

대한 행렬 $M$은 $M = R(\alpha, n_x, n_y, n_z) \cdot T(t_x, t_y, t_z)$와 같이 표현 가능한데, 이러한 사실은 123쪽의 3.6절에서 뷰잉 변환에 대하여 설명할 때 다시 고려하도록 하겠다.

### 2.5.5 기하 변환의 합성

앞에서 언급한 이동 변환, 크기 변환, 회전 변환 등은 그 자체로 매우 중요한 역할을 하지만 일반적으로 많이 사용되는 아핀 변환은 이들 보다는 더 복잡한 형태를 가지게 된다. 일반적으로 어떤 기하 변환이 필요할 때 이미 알고 있는 기본 변환을 여러 번 반복하여 새로운 변환을 만들게 되는데, 이러한 과정을 기하 변환의 합성(composition of transformation)이라 한다.

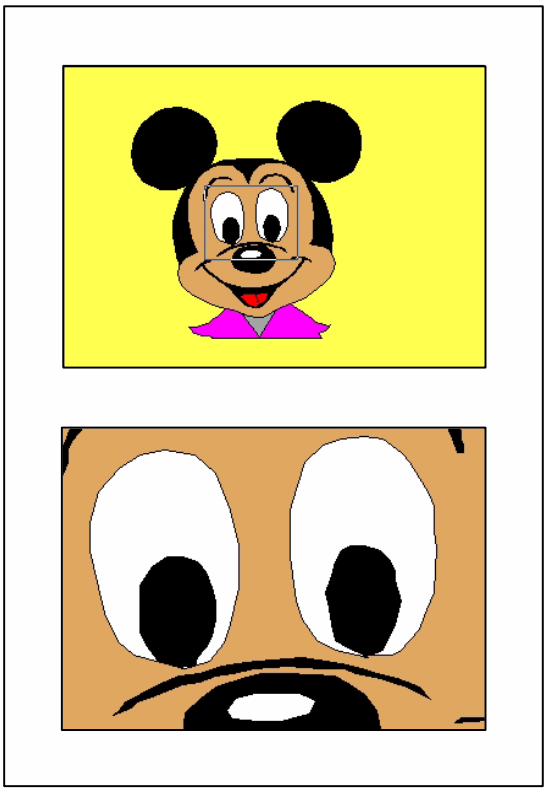


그림 2.9: 영역 확대 프로그램

간단한 예로 그림 2.9를 보자. 전체 윈도우에는 크기가 같은 두 개의 부윈도우가 위-아래로 위치하고 있다. 위쪽의 윈도우에는 미키 마우스의 얼굴이 그려져 있는데 이 윈도우에서 마우스를 사용하여 임의의 영역을 선택하면 그 안의 내용이 아래쪽 윈도우에 확대되어 보여지게 된다. 이 그림은 실제로 다각형과 선분을 사용하여 그리고 있기 때문에, 이러한 프로그램을 개발하기 위해서는 사용자가 원하는 영역을 마우스를 통하여 선택하였을 때 그 안의 기하 프리미티브들을 아래 쪽 윈도우로 옮겨주는 기하 변환을 계산하여야 한다. 전체 윈도우의 왼쪽 아래 모서리가 원점이고 왼쪽 방향이 $x$축, 위쪽 방향이 $y$축이 되도록 좌표계를 설정할 때, 이는 결국 선택 영역의 주어진 점을 아래 윈도우로 어떻게 기하 변환을 할 것인가 하는 문제가 된다. 즉 선택 영역의 내용을 아래 쪽 윈도우로

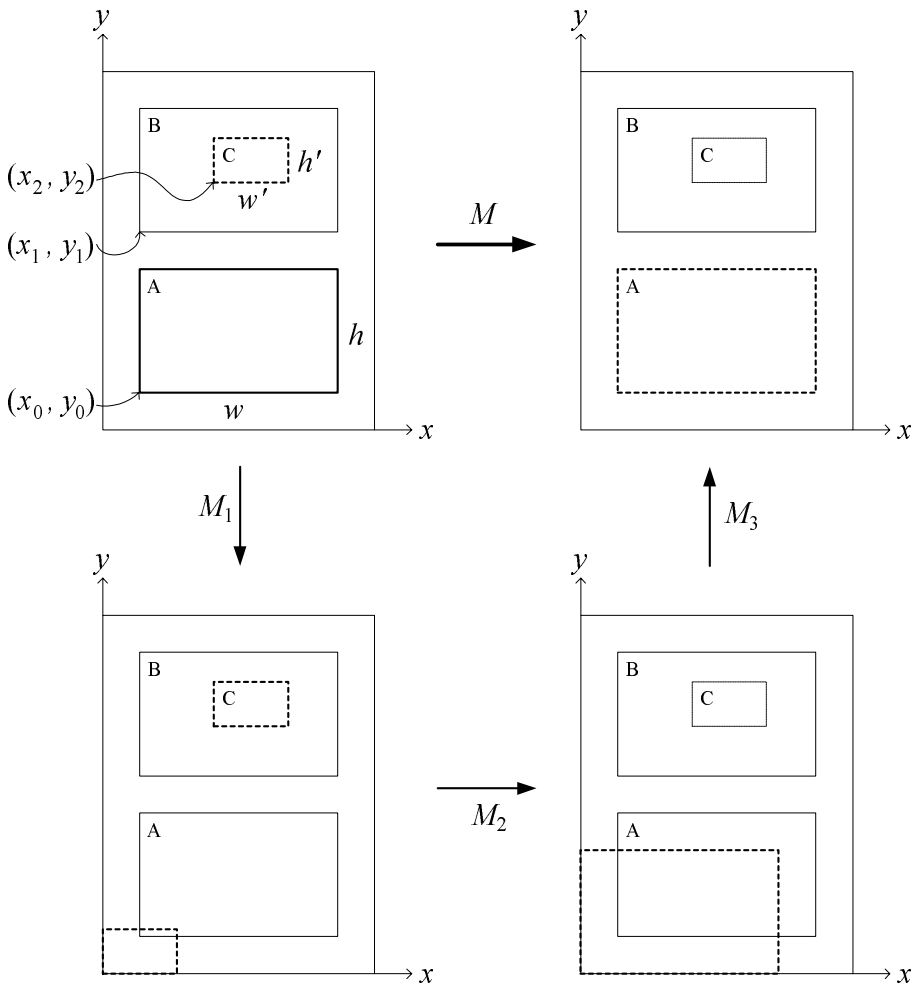



그림 2.10: 변환 행렬의 계산

그대로 일치시켜주는 변환이 필요한데 이는 앞에서 언급한 기본 변환 하나로는 불가능하다.

그림 2.10을 보면 이러한 변환을 어떻게 구할 수 있을지에 대한 단서를 제공한다. 이 예는 사실 2차원 변환이나 이는 OpenGL에서 그러하듯이 모든 점의 $z$ 좌표 값이 0인 3차원 공간의 변환으로 생각하면 된다. 우선 두 개의 부윈도우 $A$, $B$와 선택 영역 $C$의 왼쪽 아래 모서리의 좌표가 각각 $(x_0, y_0)$, $(x_1, y_1)$, $(x_2, y_2)$이고, 부윈도우의 크기가 가로 $w$ 화소, 세로 $h$ 화소, 그리고 선택 영역의 크기가 가로 $w'$ 화소, 세로 $h'$ 화소라고 하자. 문제는 어떻게 하면 우리가 이미 알고 있는 기본 변환을 반복해서 사용하

여 원하는 변환을 합성하는가 하는 것이다. 그림을 보면 우선 선택 영역의 왼쪽 아래 모서리가 원점이 되도록 이동 변환을 하고 있다($M_1 = T(-x_2, -y_2, 0)$). 다음 선택 영역의 크기가 부윈도우의 크기와 일치하도록 크기 변환을 한 후($M_2 = S(\frac{w}{w'}, \frac{h}{h'}, 1)$), 확대된 영역의 왼쪽 아래 모서리가 아래 쪽 부윈도우의 모서리와 일치하도록 이동 변환을 하고 있다($M_3 = T(x_0, y_0, 0)$). 따라서 이 프로그램에 필요한 변환 행렬 $M$은 이 세 개의 기본 변환을 합성하여 다음과 같이 만들 수가 있게 된다.

$$M = M_3 \cdot M_2 \cdot M_1 = T(x_0, y_0, 0) \cdot S(\frac{w}{w'}, \frac{h}{h'}, 1) \cdot T(-x_2, -y_2, 0)$$

여기서 명심하여야 할 것은 먼저 수행하는 변환이 항상 오른쪽에 온다는 사실인데 이는 쉽게 이해를 할 수가 있을 것이다. 요약하면 한 변환은 다른 변환을 여러 번 반복하여 합성할 수 있는데, 이는 행렬 계산의 관점에서 보면 사용하는 변환의 행렬의 순서대로(뒤에 사용하는 변환의 행렬이 왼쪽에 오도록) 곱해 변환 행렬을 계산할 수 있게 된다.

앞에서 잠깐 언급한 87쪽의 그림 2.6(c)의 변환을 다시 생각해보자. 여기서는 중심이 $p = (2.5, 2.5, 1)$에 있는 도우넛 형태의 물체(작은 것)의 크기를 중심을 유지하면서 5배 확대하고 있다. 변환 $S(5, 5, 5)$를 사용하면 이는 원점을 중심으로 하는 확대이므로 중심이 변하게 된다. 반면 이 경우는 점 $p$를 기준으로 한 확대이므로 다른 형태의 변환을 사용하여야 한다. 이 변환은 '이동 변환-크기 변환-이동 변환'의 형태로 합성할 수 있는데 간단한 연습 문제로 구해 보기 바란다.

### 2.5.6 투영 변환

아핀 변환과 함께 반드시 이해를 해야 하는 부류의 변환으로 투영 변환을 들 수가

있다. 투영 변환(projection transformation)은 수학적으로 말하면 $n > m$이라 할 때 $n$차원 공간의 점을 $m$차원 공간의 점으로 바꾸어주는 변환을 말한다. 따라서 이는 매우 방대한 형태의 변환에 해당하나, 여기서는 3차원 공간인 가상의 세상의 점을 2차원 평면인 화면의 점으로 변환해 주는($n = 3$, $m = 2$), 직관적으로 분명한 투영 변환에 대해서 살펴보자.

투영 변환은 투영 참조점(projection reference point, PRP), 투영 평면(projection plane, PP), 그리고 투영선(projector)에 의해 정의된다(그림 2.11). 투영 변환을 정의한다는 것은 3차원 공간의 임의의 점을 2차원 공간의 어디로 투영할 것인지를 명확하게 하는 것이다. 한 점 A가 주어졌을 때 A와 투영 참조점을 연결하는 직선(투영선)과 투영 평면의 교점이 2차원 평면 상의 투영점 $A'$으로 정의된다. 우리가 지금 사용하는 투영은 평면, 직선 등 평면적인 개념을 사용하고 있는데, 컴퓨터 그래픽스 분야에서는 특수한 경우를 제외하고는 이러한 평면 기하 투영(planar geometric projection)이 사용된다.

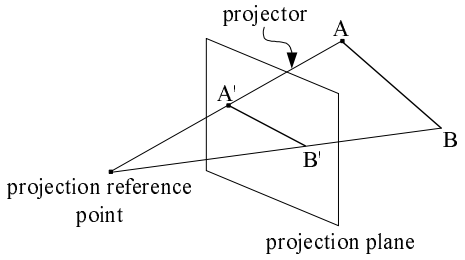
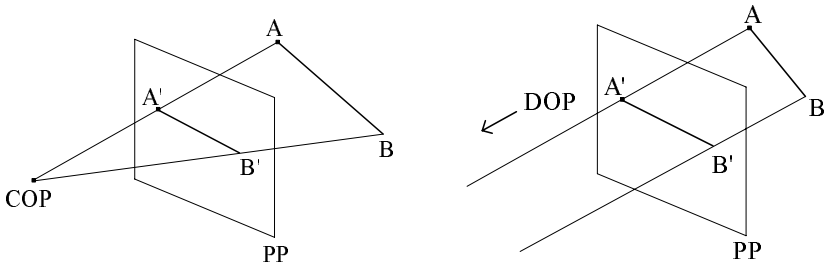



그림 2.11: 투영의 정의

투영은 크게 원근 투영과 평행 투영으로 구별된다. 이러한 분류의 기준을 앞에서 배운 동차 좌표계의 개념을 사용하여 이해해보자. 앞에서 설명한 바와 같이 유클리드 공간 또는 아핀 공간의 관점에서 보면 투영 공간은 아핀 공간의 점(동차 좌표계 $(X, Y, Z, W)$의 $W$가 0이 아닌 점)들과 방향에 해당하는 무한대 점($W = 0$인 점)들로 구성되어 있다고 볼 수가 있다. 투영 참조점의 동차 좌표의 $W$값이 0이 아닐 때, 그러한 변환을 원근 변환(perspective projection)이라 한다(그림 2.12(a)). 즉 원근 변환에서는 투영 참조점이 유클리드 공간의 한 점으로 지정이 되기때문에 (얼마나



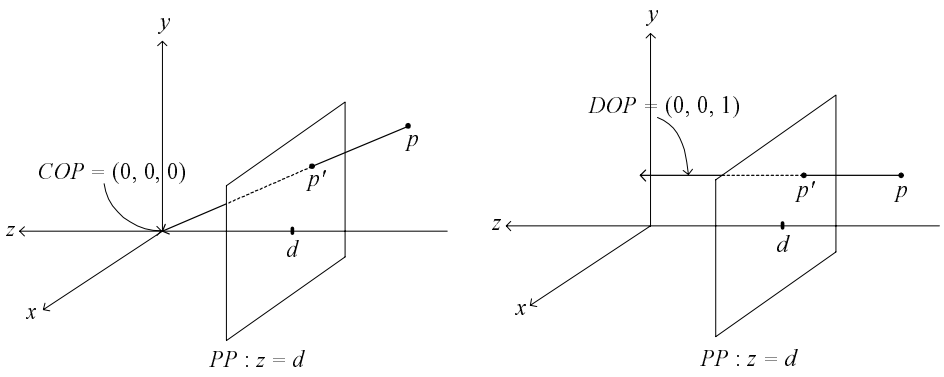

(a) 원근 투영 (b) 평행 투영

그림 2.12: 투영변환의 종류

멀건 간에) 투영 평면으로부터 유한 거리만큼 떨어져 있게 된다. 이 경우 모든 투영선들이 바로 이 점을 향해 날아 들어오기 때문에 원근 투영에서의 투영 참조점 PRP를 투영 중심(center of projection, COP)이라 부르기도 한다.

반면 투영 참조점의 동차 좌표 $W$가 0인 투영을 평행 투영(parallel projection)이라 한다(그림 2.12(b)). 이 경우에는 투영 참조점이 무한대 점이 되어, 투영 평면으로부터 특정 방향으로 무한한 거리만큼 떨어져 있게 된다. 그 결과 원근 투영의 경우 유클리드 공간의 한 점으로 향하던 투영선들이 모두 평행하게 펴지게 된다. 평행 투영에서는 투영 참조점 PRP가 투영이 되는 방향을 나타내기 때문에 이를 투영 방향(direction of projection, DOP)으로 표현하기도 한다. 평행 투영은 투영 방향과 투영 평면의 관계에 따라 직교 투영(orthographic projection)과 경사 투영(oblique projection)으로 나누어진다. 투영 방향이 투영 평면에 수직인 특수한 경우를 직교 투영이라 하고, 나머지 평행 투영은 경사 투영으로 분류된다. 원근 투영과 평행 투영은 그 특징이나 용도에 따라서 더 자세하게 분류가 되는데 그에 대한 설명은 생략하기로 한다.

원근 투영과 평행 투영의 경우에도 앞에서 살펴본 기하 변환처럼 4행 4열의 행렬로 표현할 수 있다. 예를 들어 그림 2.13(a)의 원근 투영에 대한 행렬을 계산해




(a) 원근 투영 (b) 평행 투영

$$\begin{pmatrix} \frac{dx}{z} \\ \frac{dy}{z} \\ d \\ 1 \end{pmatrix} \longleftarrow \begin{pmatrix} x \\ y \\ z \\ \frac{z}{d} \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

(c) 원근 투영 계산

그림 2.13: 투영 변환의 예

보자. 이 예에서는 투영 중심이 원점이고 투영 평면이 $z = d$인 $z$축에 수직인 평면으로 설정이 되어 있다. 지금 임의의 점 $p = (x\ y\ z)^t$를 투영하여 얻어지는 점 $p' = (x'\ y'\ z')^t$의 좌표를 구하려 한다. 이 때 투영 중심과 투영하려는 점을 연결한 투영선은 실수인 매개 변수 $t$를 사용하여 $C(t) = (t \cdot x\ t \cdot y\ t \cdot z)^t$와 같이 나타낼 수 있는데, 여기서 $t$값에 따라 어떤 점인지가 결정이 된다. 지금 문제는 이 직선과 투영 평면과의 교점의 위치에 해당하는 매개 변수 $t_0$를 계산하는 것이다. 이 경우 $t_0 \cdot z = d$라는 관계를 만족해야 하므로 $t_0 = \frac{d}{z}$가 되고, 따라서 투영점 $p'$은 $(\frac{d \cdot x}{z}\ \frac{d \cdot y}{z}\ d)^t$가 됨을 알 수가 있다. 문제는 어떻게 하면 $p'$을 4행 4열 행렬과 $p$의 곱으로 표현할 것인가 하는 것이다. 아핀 변환의 정의를 다시 상기하면 변환된 점의 좌표는 원래의 점의 좌표 값의 선형 결합(즉 좌표 값들에 각각 어떤 상수들을 곱해 더한 형태)으로 표현이 되는 반면, 이 원근 투영의 경우 좌표 값 중의 하나인 $z$가 분

모에 나타나기 때문에 아핀 변환으로 투영을 할 수가 없음을 알 수 있다.

그러면 어떻게 이 문제를 해결할 것인가? 바로 원근 투영을 할 때 유클리드 공간에서 출발하여 투영 공간에 잠시 들렸다가 다시 돌아오는 방식으로 계산을 하면 된다. 투영 공간을 기본 좌표계로 사용한다면 자연스럽게 이 행렬을 유도할 수 있으나, 유클리드 공간에서는 약간의 기법 적용이 필요하다. 즉 유클리드 공간의 좌표와 투영 공간의 좌표의 관계 $(x'\ y'\ z')^t = (\frac{X'}{W'}\ \frac{Y'}{W'}\ \frac{Z'}{W'})^t$를 이용하기 위하여 $X' = x$, $Y' = y$, $Z' = z$, $W' = \frac{z}{d}$와 같이 위의 식을 표현하면 $X'$, $Y'$, $Z'$, $W'$은 $x$, $y$, $z$의 선형 결합으로 표현이 가능해지고, 따라서 이에 대한 원근 투영 행렬 $M_{pers}$는 다음과 같게 된다.

$$M_{pers} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

그림 2.13(c)는 원근 투영을 할 때 유클리드 공간에서$((x\ y\ z\ 1)^t)$ 일단 투영 공간으로 간 후$((x\ y\ z\ \frac{z}{d})^t)$ 다시 유클리드 공간으로 돌아 오는$((\frac{dx}{z}\ \frac{dy}{z}\ d\ 1)^t)$ 계산 과정을 보여주고 있다. 사실 투영 공간에서 보면 단순한 선형 변환이지만 유클리드 공간에서 보면 약간 부자연스러운 변환처럼 보인다.

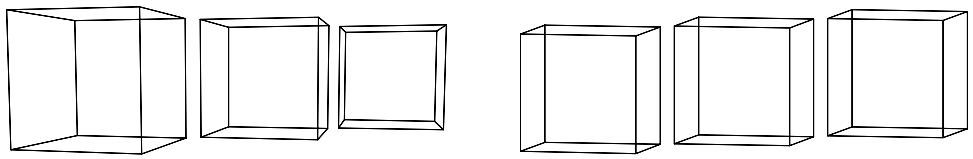
다음 그림 2.13(b)의 평행 투영은 투영 방향이 양의 $z$축 방향인 직교 투영인데 이

때의 변환 행렬 $M_{ortho}$는 다음과 같음을 쉽게 알 수 있다.

$$M_{ortho} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

이 두 예를 통하여 원근 투영과 평행 투영의 일반적인 특징에 대하여 간략히 알아보자. $M_{pers}$과 같은 원근 투영 행렬의 네 번째 행은 더 이상 $(0\ 0\ 0\ 1)$이 아니고, 따라서 원근 투영은 아핀 변환이 아님을 알 수 있다. 따라서 아핀 변환의 기본 성질들이 더 이상 만족이 되지 않는다. 예를 들어 평행한 두 직선을 따라 바라보면서 원근 투영을 하면, 변환 후 두 직선이 한 점으로 수렴하게 되고 따라서 아핀 변환의 특징 중의 하나인 평행성 유지의 성질이 더 이상 만족되지 않음을 알 수 있다. 반면에 평행 투영은 아핀 변환의 한 종류이다. 따라서 평행성 유지나 비율 보존 등의 성질이 그대로 만족한다. 그러면 평행 투영은 강체 변환일까? 이에 대하여 생각해보기 바란다.

또한 $M_{pers}$와 $M_{ortho}$를 보면 이 두 개의 4행 4열 행렬의 계수가 3임을 알 수 있다. 여기서 구한 변환은 3차원 공간에서 2차원 공간으로의 변환으로서 공간의 차원이 하나 줄어들게 되는데, 이러한 기하학적 성질에 대응하여 변환 행렬의 계수가 1만큼 감소하게 된다. 앞에서 고려한 기본 아핀 변환처럼 3차원 공간을 3차원 공간으로 변환을 해주면 그 때의 변환 행렬 계수는 4가 된다. 또한 만약 3차원 공간의 점을 1차원 공간의 직선 상으로 투영을 한다면 투영 행렬의 계수는 2(=3-1)가 감소하여 2가 될 것이다. 위의 두 변환의 경우 행렬의 계수가 4가 아니므로 이 행렬들에는 역행렬이 존재하지 않게 된다. 이는 투영 변환을 되돌릴 수가 없음을 의미하는데,



(a) 원근 투영 (b) 평행 투영

그림 2.14: 원근 투영과 평행 투영

투영 후 투영 평면의 한 점에 무한개의 점이 대응이 되므로 이를 역투영할 수 없음은 직관적으로 명확하다.

아무래도 원근 투영과 평행 투영을 고려할 때 가장 피부에 와 닿는 차이점은 원근 투영을 사용할 때 발생하는 원근 축소(perspective foreshortening) 현상일 것이다. 즉 멀리 있는 물체가 가까이 있는 물체보다 상대적으로 작게 보이게 함으로써 물체간의 거리를 느낄 수 있게 해주는 원근감이 자연스럽게 나타나는데, 이러한 효과가 원근 투영의 정의에서 기인함은 그림 2.12(b)를 다시 보면 분명하다. 그림 2.14은 세 개의 정육면체를 같은 위치에서 투영 종류만 바꿔가면서 투영을 한 모습을 보여주고 있다. 원근 투영을 사용한 왼쪽 그림에서 거리감을 더 잘 느낄 수 있음을 알 수 있다. 반면에 오른쪽의 평행 투영 결과의 이미지를 보면 아핀 변환의 특징인 평행성 유지, 비율의 보존 등의 성질이 잘 나타나 있다. 이렇듯 원근 투영은 우리가 사물을 바라볼 때 자연스럽게 느끼는 원근감 효과를 제공함으로써 현실감이 있는 렌더링을 해야 할 경우 많이 사용이 된다. 한편 평행 투영은 물체의 형태에 대한 정보를 잘 유지하여 주므로 기하 설계 등의 공학 분야에서 널리 쓰이고 있다.

원근 투영의 원근감 생성 효과는 사실 변환 행렬 $M_{pers}$를 사용한 투영 계산을 통해서도 이해할 수 있다. 투영하려는 점에 이 행렬을 곱한 후 다시 유클리드 공간으로 돌아오기 위하여 $x$, $y$, $z$ 값을 $\frac{z}{d}$로 나누어 주는 과정을 생각해보자. 여기서 $\frac{z}{d}$ 값

이 의미하는 것이 무엇일까? 이는 바로 투영하려는 점이 화면에서 얼마나 떨어져 있는가를 나타내는, 정규화가 된 깊이 정보를 제공하는 값으로서 당연히 멀리 떨어져 있을 수록 그 값이 커지게 된다. 따라서 이 값으로 $x$, $y$값을 나눌 경우 멀리 있는 점일 수록 $x$, $y$ 좌표 값이 더 작아지므로 원근감 효과가 발생하는 것이다. 이러한 나눗셈을 원근 나눗셈(perspective division)이라 한다.

OpenGL 기하 파이프라인에서도 이러한 원근 투영과 평행 투영을 사용한다. 그러나 본 절에서 살펴본 투영 변환과 같이 3차원 공간을 2차원 공간으로 투영을 하여 공간에 대한 정보를 투영과 동시에 잃어 버리는 것이 아니라 3차원적인 공간 정보를 계속하여 유지하게 된다. 즉 단순히 물체가 투영 평면의 어느 지점에 투영되는가 하는 2차원적인 투영 정보뿐만 아니라 물체가 화면에서 얼마나 멀리 떨어져 있는가 하는 깊이 정보를 계속해서 유지하여 주는 변환을 사용하기 때문에 계수가 4인 행렬을 사용하게 되고, 그 결과 계속해서 3차원(2+1)적인 정보를 유지하게 된다. 이에 대해서는 3.11절에서 다시 한번 자세히 살펴보겠다.

# 제 3 절 OpenGL에서의 좌표계와 기하 변환

## 3.1 기하 파이프라인

이제 지금까지 살펴본 기하 변환에 대한 지식을 바탕으로 하여 OpenGL에서의 기하 변환에 대하여 알아보자. OpenGL 프로그래밍의 근본적인 목적은 3차원 그래픽스 렌더링 기법을 사용하여 '실시간적으로 사실적인' 영상을 생성하는 것인데, 이를 위하여 여러 가지 부류의 렌더링 계산이 수행되어야 한다고 하였다. 즉 물체가 화면의 어디에 나타나게 할 것인가, 그 물체가 어떤 색깔로 보이게 할 것인가, 멋있는 텍스춰 영상을 물체에 어떻게 붙일 것인가, 어떻게 효과적으로 그림자 효과를 낼 것인가 등 프로그래머가 OpenGL 함수를 통하여 내린 명령에 대한 계산이 수행되어야 한다. OpenGL을 통한 3차원 그래픽스 프로그래밍을 정확하게 하기 위해서는 우선 3차원 뷰잉에 관련된 부분, 즉 기하 물체들이 존재하는 가상의 세상의 어떤 물체가 화면 어디에 나타나게 할 것인가에 관한 기하학적인 계산 모델을 정확하게 이해하고, 또한 어떻게 하면 OpenGL 함수를 사용하여 프로그래머가 원하는 방식으로 3차원 뷰잉 계산을 하도록 할 것인가를 이해하여야 한다. 그림 2.15에는 OpenGL 시스템 아키텍춰 중 이러한 기하 계산 모델에 관련된 부분만 뽑아 만든 기하 파이프라인이 도시되어 있다. 본 3절에서는 바로 이 기하 파이프라인에서 사용되는 계산 모델에 대하여 이해하고, OpenGL 함수를 통하여 이 파이프라인을 정확하게 구동하는 방법에 대하여 배우도록 하겠다.

## 3.2 OpenGL의 뷰잉 모델과 사진 촬영과의 관계

OpenGL에서의 3차원 뷰잉의 과정은 카메라로 사진을 찍는 과정과 유사하다. 우리가 사진을 찍을 때(또는 영화를 찍을 때) 우선 촬영장에 배우와 소품 그리고 조명

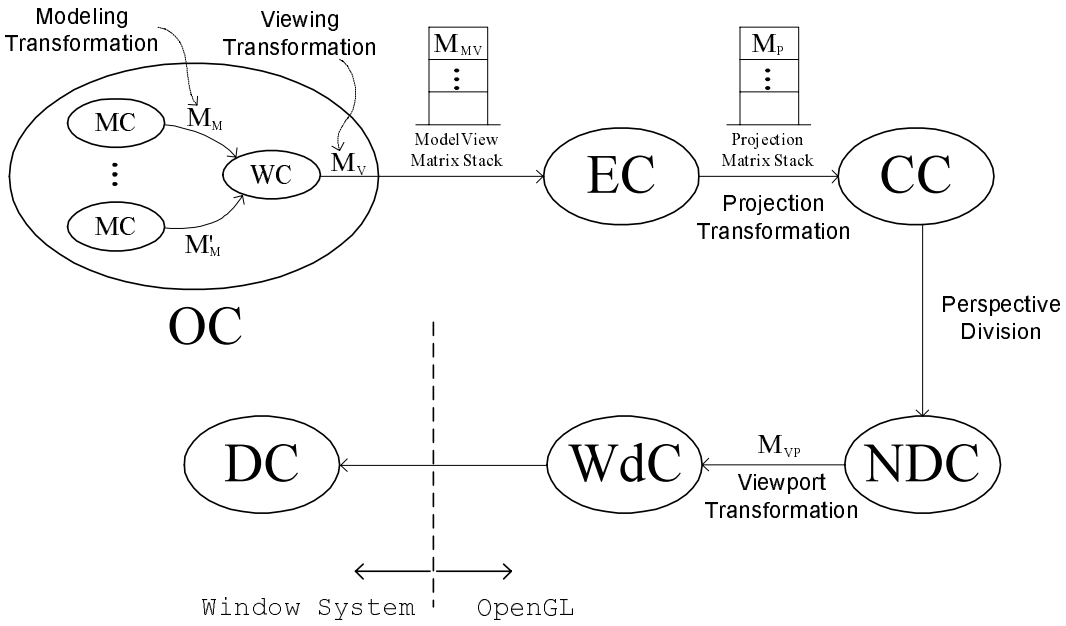



그림 2.15: OpenGL의 기하 파이프라인

| 사진 촬영 | OpenGL 기하 변환 |
|---|---|
| 피사체 및 조명 배치 | Modeling Transformation |
| 카메라의 위치와 방향 설정 | Viewing Transformation |
| 사용할 렌즈 결정 및 촬영 대상 결정 | Projection Transformation |
| 카메라의 셔터를 누름 | |
| 사진의 크기 결정 및 인화 | Viewport Transformation |

표 2.1: OpenGL의 뷰잉 모델과 사진 촬영과의 비교

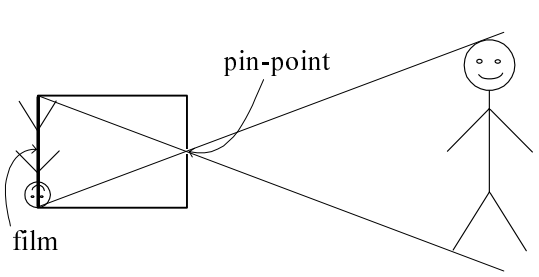
등을 원하는 위치에 배치를 한다. 촬영장을 그래픽스 렌더링에서는 세상(world) 또는 장면(scene)이라고 하는데, 이러한 배치 과정은 OpenGL에서는 모델링 변환(modeling transformation)이라고 하는 기하 변환을 통하여 수행이 된다. 피사체와 조명의 배치가 끝난 후에는 카메라의 위치와 방향을 설정하여야 하는데 OpenGL에서는 뷰잉 변환(viewing tranformation)을 통하여 이러한 작업을 한다.

다음 카메라의 줌 기능을 사용하여 렌즈의 초점 거리를 결정하는데, 이를 통하여 촬영할 대상의 범위를 정하게 된다. 다음 사진을 찍으면 셔터가 열리게 되고 조명에서 출발한 빛이 피사체에 반사가 되어 렌즈를 통하여 들어와 필름을 감광시키

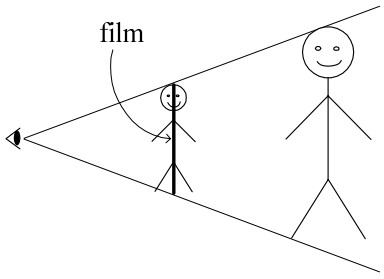
게 된다. 이러한 과정은 OpenGL에서는 개념적으로 투영 변환(projection transformation)을 통하여 수행이 된다. 마지막으로 필름을 현상하여 원하는 크기로 확대한 후 인화를 하는데, 이는 컴퓨터 화면에 얼마나 크게 이미지를 나타낼 것인가에 관한 문제로서 OpenGL에서는 뷰폿 변환(viewport transformation)에 대응이 된다. 물론 이러한 과정이 정확하게 일대일로 대응이 되는 것은 아니지만 개념적으로 상당히 유사한 점이 있기 때문에 이러한 비교를 염두에 두고 OpenGL 프로그래밍을 하면 많은 도움이 될 것이다.

다음 절로 넘어가기 전에 컴퓨터 그래픽스에서 가장 단순하면서도 보편적으로 사용되는 카메라 모델인 바늘 구멍 카메라(pin-point camera) 모델에 대하여 간단히 살펴보자. 누구든지 쉽고 재미있게 실험을 할 수가 있는데, 예를 들어 양철로 만든 직육면체 형태의 녹차 통이 있다고 가정하자. 암실에 들어가 뚜껑을 연 후 한 쪽 면에 필름을 붙이고 뚜껑을 닫은 후 반대 면에 바늘로 구멍을 뚫어 바깥쪽에 검은 테이프를 붙인다. 이 통을 밖으로 가져 나와 원하는 피사체 방향으로 놓고 테이프를 제거한 후 수십 분 정도 나둔 다음 다시 테이프를 붙여 암실에 가져와 현상, 확대, 인화를 하면 마치 고급 카메라로 촬영한 것과 같은 정교한 사진이 나온다.

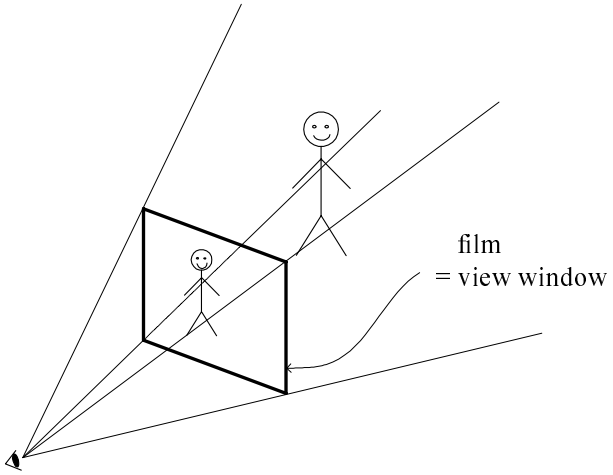
바로 이것이 컴퓨터 그래픽스에서 사용되는 가장 원시적인 카메라 모델로 개념적으로 그림 2.16(a)와 같이 도시할 수 있다. 이 그림은 녹차 통의 옆모습을 보여주고 있는데 잘 알다시피 빛이 바늘 구멍을 통하여 들어와 필름에 상이 거꾸로 맺히게 된다. 이 때 필름을 뒤집어 앞쪽으로 옮기면 이러한 카메라 모델을 그림 2.16(b)와 같이 생각할 수가 있다. 이를 다시 3차원적으로 도시하면 그림 2.16(c)에서와 같이 그래픽스 렌더링에서 가장 기본이 되는 카메라 모델을 얻게 된다. 즉 바늘 구멍에 해당하는 눈의 위치와 필름에 해당하는 뷰 윈도우가 결정이 되면 앞에서 투영 변환에 관한 절에서 설명한 것과 같은 투영이 일어나게 되는데, 이로부터 투영 변환과




(a) '녹차 통' 카메라




(b) 바늘 구멍 카메라 모델




(c) 3차원 카메라 모델

그림 2.16: 바늘 구멍 카메라 모델

바늘 구멍 카메라 모델과의 관계를 이해할 수가 있다. 물론 실제로 사용하는 카메라는 빛이 통과하는 구멍, 즉 렌즈가 어느 정도의 면적을 가지기 때문에, 그러한 결과로 피사계 심도가 생기게 된다. 바늘 구멍 카메라 모델은 어떻게 보면 이론적으로만 존재하는 비현실적인 모델이기 때문에, 고급 렌더링 기법에서는 실제의 렌즈를 사용한 것과 같은 사실적인 영상을 생성하는 방법을 사용한다.

## 3.3 OpenGL의 기하 계산 관련 함수

기하 파이프라인에 대하여 본격적으로 살펴보기 전에 이에 관련된 OpenGL 함수들에 대하여 간략히 살펴보자. 우선 어떤 종류의 데이터들이 OpenGL의 기하 파이프라인을 통하여 흘러갈까? 앞에서도 말한 바와 같이 파이프라인에 입력으로 흘러들어가는 데이터는 그리려고 하는 물체들, 즉 그것들을 구성하는 다각형, 선분, 점과 그것들에 대한 속성(attribute)들이다. 다각형, 선분은 꼭지점들로 구성되어 있으므로 가장 기본적인 데이터 요소는 $(x, y, z, w)$ 형태의 꼭지점이고, 그와 함께 그에 대한 성질들, 법선 벡터, 색상, 텍스춰 좌표 등이 프로그래머가 설정하는 데로 각각 $(nx, ny, nz, nw)$, $(r, g, b, a)$, $(s, t, r, q)$ 형태로 꼭지점에 붙어 흘러가게 된다. 기하 파이프라인 관점에서 볼 때 가장 중요한 계산은 3차원 공간의 점에 대한 좌표 변환 계산이라 할 수 있는데, OpenGL에는 이러한 계산을 효율적으로 할 수 있도록 시스템 아키텍춰 및 함수가 설계되어 있다. 3차원 그래픽스 프로그래밍에 있어 중요한 것은 사용 API의 구조를 정확히 이해하고, 이를 가장 효율적으로 구동할 수 있도록 프로그램을 작성하여야 한다는 점이다. 무엇보다도 프로그래머의 입장이 아니라 시스템 입장에서 프로그래밍을 하여야 한다는 사실은 이해하여야 한다.

그림 2.15에 도시된 기하 파이프라인을 보면 모델뷰 행렬 스택(ModelView matrix stack)와 투영 행렬 스택(Projection matrix stack)이라고 하는 두 개의 행렬 스

택(matrix stack)이 있음을 알 수가 있다. 이 두 가지 행렬 스택은 꼭지점 좌표 변환에 필요한 행렬을 동적으로 저장하는데 쓰이는데, 이에 대한 사용법의 정확한 이해가 무엇보다도 정확한 프로그래밍에 필수적이라 할 수 있다. 실제로 OpenGL에서는 이 두 행렬 스택과 함께 텍스춰 좌표 변환에 필요한 텍스춰 행렬 스택(texture matrix stack, 그리고 1.2 버전에 새롭게 추가된 OpenGL 시스템 아키텍춰의 한 부분인 이미징 서브셋(imaging subset)에서 제공하는 색상 변환 기능에 쓰이는 색깔 행렬 스택(color matrix stack) 등 모두 네 가지 종류의 행렬 스택이 제공되는데, 본 2장에서는 우선 모델뷰 행렬 스택과 투영 행렬 스택의 활용에 대하여 살펴보도록 하자.

표 2.2에는 기하 파이프라인과 관련한 함수들이 요약이 되어 있는데, 이 함수들은 기본적으로 행렬 스택에 대하여 서로 다른 방식으로 영향을 미친다. 스택(stack)은 컴퓨터학에서 사용되는 가장 중요한 자료 구조의 하나로서 데이터 요소가 한쪽 방향으로만 들어 갔다 나올 수 있는 구조를 가진다. 따라서 이 자료 구조에서는 항상 가장 늦게 저장된 데이터가 가장 먼저 나오는 특성을 가진다. 이러한 성질을 LIFO(Last-In First-Out)이라고 하는데, 푸쉬(push)와 팝(pop)이라는 연산을 통하여 구현이 된다. 푸쉬는 데이터를 집어넣는 연산이고 팝은 마지막으로 들어간 데이터를 빼내는 연산인데, 가장 늦게 들어가 현재 스택에 있는 원소를 스택의 탑(top)에 있는 원소라 한다.

3차원 그래픽스 프로그래밍에서도 스택 연산이 중요하게 쓰인다. 일반적으로 그래픽스 프로그래밍 API의 문맥에서의 스택의 사용은 일반적으로 우리가 알고 있는 스택의 사용과 약간의 차이가 있다. 우선 스택에 저장되는 데이터의 형태는 3차원 기하 변환을 저장하기 위한 4행 4열 행렬이다. OpenGL에서는 이 2차원 행렬을 16개의 원소를 가지는 1차원 배열로 저장을 하는데, 중요한 것은 1차원으로 저장

| OpenGL 함수 | 용 도 |
|---|---|
| void glMatrixMode(GLenum mode); | 현재 행렬 스택의 지정 |
| void glLoadMatrixd(const GLdouble *m);<br>void glLoadMatrixf(const GLfloat *m); | 행렬 m을 현재 행렬로 지정 |
| void glLoadIdentity(void); | 단위 행렬을 현재 행렬로 지정 |
| void glMultMatrixd(const GLdouble *m);<br>void glMultMatrixf(const GLfloat *m); | 행렬 m을 현재 행렬의 오른쪽에<br>곱해 현재 행렬로 지정 |
| void glTranslated(GLdouble x, GLdouble y,<br>GLdouble z);<br>void glTranslatef(GLfloat x, GLfloat y,<br>GLfloat z); | 해당 이동 변환 행렬을<br>현재 행렬의 오른쪽에 곱함 |
| void glScaled(GLdouble x, GLdouble y,<br>GLdouble z);<br>void glScalef(GLfloat x, GLfloat y,<br>GLfloat z); | 해당 크기 변환 행렬을<br>현재 행렬의 오른쪽에 곱함 |
| void glRotated(GLdouble angle, GLdouble x,<br>GLdouble y, GLdouble z);<br>void glRotatef(GLfloat angle, GLfloat x,<br>GLfloat y, GLfloat z); | 해당 회전 변환 행렬을<br>현재 행렬의 오른쪽에 곱함 |
| void glOrtho(GLdouble left, GLdouble right,<br>GLdouble bottom, GLdouble top,<br>GLdouble zNear, GLdouble zFar); | 해당 직교 투영 변환 행렬을<br>현재 행렬의 오른쪽에 곱함 |
| void glFrustum(GLdouble left, GLdouble right,<br>GLdouble bottom, GLdouble top,<br>GLdouble zNear, GLdouble zFar); | 해당 원근 투영 변환 행렬을<br>현재 행렬의 오른쪽에 곱함 |
| void gluPerspective(GLdouble fovy, GLdouble aspect,<br>GLdouble zNear, GLdouble zFar); | 해당 원근 투영 변환 행렬을<br>현재 행렬의 오른쪽에 곱함 |
| void gluLookAt(GLdouble ex, GLdouble ey,<br>GLdouble ey, GLdouble cx, GLdouble cy, GLdouble cz,<br>GLdouble ux, GLdouble uy, GLdouble uz); | 해당 뷰잉 변환 행렬을<br>현재 행렬의 오른쪽에 곱함 |
| void glViewport(GLint x, GLint y,<br>GLsizei width, GLsizei height); | 해당 뷰폿 변환 행렬을<br>현재 행렬의 오른쪽에 곱함 |
| void glPushMatrix(void); | 현재 행렬 스택을 한 번 푸쉬함 |
| void glPopMatrix(void); | 현재 행렬 스택을 한 번 팝함 |

표 2.2: 기하 파이프라인 관련 주요 OpenGL 함수

할 때 C/C++ 프로그래머들에게 익숙한 행-우선(row-major) 순서가 아니라, 열-우선(coloum-major) 순서를 사용한다는 것이다. 예를 들어 GLfloat m[16];과 같이 행렬을 선언하면 이는 m[6]은 2행 3열이 아니라 3행 2열에 있는 원소를 저장한다.

항상 어떤 순간에 스택의 탑에 있는 행렬을 현재 행렬(current matrix)이라고 하는데, OpenGL에서는 여러 개의 스택이 존재하기 때문에 현재 행렬을 정확하게 지칭을 하려면 현재 행렬 스택(current matrix stack)을 지정하여야 한다[5]. glMatrixMode(mode); 함수의 목적이 바로 현재 행렬 스택을 지정하는 것인데, GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE, GL_COLOR 중 원하는 스택에 해당하는 상수를 사용하여 현재 행렬 스택으로 지정할 수 있다. 예를 들어 glMatrixMode(GL_MODELVIEW);와 같이 함수를 호출할 경우 모델뷰 행렬 스택이 현재 스택으로 지정이 되며, 그 결과 자동적으로 현재 이 스택의 탑에 있는 행렬이 현재 행렬로 지정된다. 이렇게 한 스택이 현재 스택으로 지정이 되면 그 순간부터 다른 행렬 스택이 현재 스택으로 지정이 될 때까지 표 2.15에 나열된 기하 변환에 관련된 모든 OpenGL 함수들은 현재 스택에 영향을 미치게 된다. 행렬 스택에는 초기 상태로 각 스택마다 한 개씩의 행렬이 올려져 있고, 그 내용은 단위 행렬로 지정이 되어 있으며 초기 행렬 모드는 GL_MODELVIEW로 지정되어 있다.

$$m = \begin{bmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{bmatrix}$$

그림 2.17: 변환 행렬 $m$의 저장

표 2.15에 있는 함수들이 스택에 영향을 미치는 방식은 함수에 따라 다른데, glLoadMatrix*(*m);와 glLoadIdentity();함수 같은 경우에는 현재 행렬의 내용을 각각 행렬 m과 단위 행렬로 탑재하게 된다. 또한 glMultMatrix*(*m); 함수는 그래픽스 프로그래밍의 한 특징을 잘 나타내고 있는데, 바로 행렬 m을 현재 행렬의 오른

[5] 편의상 현재 스택이라고도 하겠다.

쪽에 곱해 그 값을 현재 행렬의 내용으로 대체하게 된다.

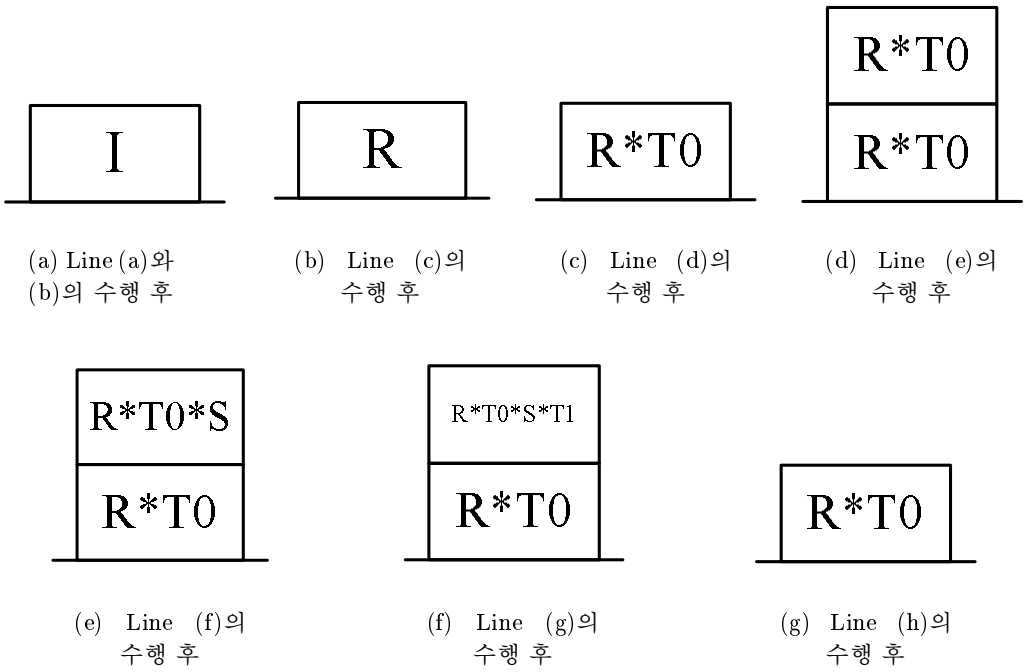
한편 이 표의 glTranslate*(*);부터 glViewport(*);까지의 함수들은 모두 glMultMatrix*(*) 함수에 기반을 두고 있다. 이 함수들을 호출할 경우 해당 목적에 맞는 행렬 m을 계산하여 glMultMatrix*(m) 함수를 수행하는 것과 같은 효과가 있게 된다. 예를 들어 glTranslatef(3.0, 1.0, 5.0);과 같은 문장을 수행하면 $T(3.0, 1.0, 5.0)$에 해당하는 행렬이 계산이 되어 현재 행렬의 오른쪽에 곱해진다.

표의 마지막에 있는 glPushMatrix();와 glPopMatrix(); 함수들은 그 이름이 의미하듯이 현재 행렬 스택에 대하여 각각 푸쉬와 팝 연산을 수행한다. glPushMatrix() 함수의 경우 새로운 행렬이 푸쉬가 되어 현재 행렬이 되는데, 특이한 것은 바로 직전의 현재 행렬의 내용이 새로운 현재 행렬로 복사가 된다는 점이다. glPopMatrix() 함수의 경우 현재 행렬을 팝하게 되고 그 결과 바로 아래의 행렬이 현재 행렬로 바뀌게 된다. 이 두 함수는 다음에 여러 예를 통하여 자연스럽게 알게 되겠지만 현재 행렬의 내용을 안전하게 보관하고 또 추후에 사용을 할 수 있는 역할을 하며, 이를 통하여 여러 가지 형태의 동적 프로그래밍을 할 수가 있게 된다.

스택이라는 자료 구조를 구현할 때 결정하여야 할 점 중의 하나는 과연 스택의 크기를 어떻게 정할 것인가 하는 것이다. OpenGL 1.2의 경우 모델뷰 행렬 스택의 경우 최소한 깊이가 최소한 32개 이상, 그리고 다른 스택의 경우 두 개 이상이 되도록 규정되어 있는데, 사용하는 OpenGL 라이브러리에 따라 실제 크기가 다를 수가 있다. 마지막으로 지금까지의 내용을 간단한 예를 통하여 확인하기 위하여 아래의 코드와 그림 2.18에 도시된 각 함수 수행 후의 행렬의 내용을 비교해보기 바란다.

**프로그램 예 2.1** 스택의 사용 예.

```
glMatrixMode(GL_MODELVIEW); // Line (a)
glLoadIdentity(); // Lind (b):  I = unit matrix
```




(a) Line (a)와 (b)의 수행 후

(b) Line (c)의 수행 후

(c) Line (d)의 수행 후

(d) Line (e)의 수행 후

(e) Line (f)의 수행 후

(f) Line (g)의 수행 후

(g) Line (h)의 수행 후

그림 2.18: 모델뷰 행렬 스택의 내용

```
glRotatef(90.0, 0.0, 1.0, 0.0); // Line (c):  R          =
R(90, 0, 1, 0)
glTranslatef(2.0, 0.0, 0.0); // Line (d):  T0 = T(2, 0, 0)
glPushMatrix(); // Line (e)
glScalef(2.0, 2.0, 2.0); // Line(f):  S = S(2, 2, 2)
glTranslatef(-2.0, 3.0, 0.0); // Line (g):  T1 = T(−2, 3, 0)
draw_object();
glPopMatrix(); // Line (h)
```

## 3.4 물체 좌표계와 윈도우 좌표계

이제 본격적으로 OpenGL 기하 파이프라인에서의 좌표 변환에 대하여 살펴보자. 이제부터 설명할 내용을 보다 정확하게 이해하기 위하여 그림 2.15과 표 2.1에 있는 내용을 항상 생각을 하면서 비교하기 바란다. 그림 2.15에 도시되어 있는 바와 같이 OpenGL에서의 기하 변환은 물체 좌표계(Object Coordinates, OC)라고 부르는 좌표계에서 출발하여 윈도우 좌표계(Window Coordinates, WdC)라고 하는 좌표계에 이르게 된다. 윈도우 좌표계는 프로그래머가 자신이 만들고 있는 이미지를 도시하기 위하여 원하는 크기의 윈도우를 설정하였을 때, 그 윈도우 안에서의 위치를 정의하기 위한 좌표계이다.

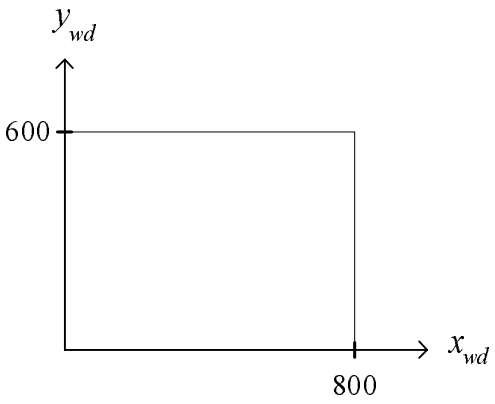



그림 2.19: 윈도우 좌표계

즉 예를 들어 가로 800 화소, 세로 600 화소의 윈도우에 그림을 그리기 원할 경우 그림 2.19과 같이 윈도우에 대한 좌표계가 결정이 된다. 전에 기술한 바와 같이 OpenGL의 윈도우 좌표계에서는 윈도우의 왼쪽 아래의 모서리가 원점이 되고 오른쪽으로 갈수록 $x_{wd}$의 값이 증가하고, 또한 위로 갈수록 $y_{wd}$의 값이 커진다. 윈도우 좌표계는 그림을 그리려고 하는 그림판, 또는 사진을 인화하려는 인화지에 대한 좌표계이며, 이 좌표계를 통하여 결과적으로 생성되는 그림의 정확한 위치 설정이 가능해진다.

그러면 OpenGL 기하 파이프라인의 다른 쪽 끝인 물체 좌표계는 과연 무엇을 위한 좌표계일까? 물체 좌표계를 개념적으로 모델링 좌표계(Modeling Coordinats, MC)와 세상 좌표계(World Coordinates, WC)로 나누어 생각을 하면 좀 더 쉽게 기

하 파이프라인을 이해할 수가 있다. 표 2.1에 요약되어 있는 바와 같이 OpenGL 프로그래밍 과정을 개념적으로 사진을 찍는 과정과 비교하여 생각을 할 수가 있다고 하였는데, 세상 좌표계는 우리가 존재하는 세상에 대한 좌표계라 할 수 있다. 즉 사진을 찍을 때 피사체들을 배치한다고 하였는데 그들의 위치, 방향, 크기 등을 정확하게 기술하기 위해서는 가상의 세상에 대한 좌표계 설정이 필요하고, 바로 그것을 위하여 사용하는 좌표계가 세상 좌표계인 것이다. 다시 말해서 세상 좌표계는 무대 또는 촬영장에 대한 좌표계인 셈이다.

피사체와 조명등의 배치를 위해서는 우선 피사체에 해당하는 기하 물체들을 만들어야 한다. 예를 들어 토이 스토리라는 영화를 만들기 위해서는 우선 각각의 장난감 또는 등장 인물에 해당하는 물체를 만들어야 한다. 이러한 작업은 기하 모델링이라는 과정을 통하여 이뤄지는데, 앞에서 말한 바와 같이 OpenGL에서는 기본적으로 다면체 모델을 사용하여 물체를 표현하게 된다. 따라서 토이 스토리에 나오는 우디라는 주인공에 해당하는 기하 모델을 정확하게 표현하기 위해서는, 물체를 이루는 각 꼭지점에 대한 좌표계의 설정이 필요하다. 이렇게 개개의 물체를 모델링할 때 사용되는 좌표계를 모델링 좌표계라 한다. 물론 공통의 세상 좌표계를 사용하여 모든 물체를 한 좌표계에서 만들 수도 있으나, 일반적으로는 각 물체를 그에 해당하는 모델링 좌표계에서 만든 후 모델링 변환(modeling transformation)이라 하는 좌표 변환을 통하여 세상 좌표계로 배치하는 방식을 택한다. 이에 대한 이유는 뒤에서 상세하게 설명을 하겠다.

당분간은 기하 파이프라인을 쉽게 이해를 하기 위하여 모델링 좌표계는 없다고 가정하고 OpenGL의 물체 좌표계는 단순히 세상 좌표계로 이루어져 있다고 생각하자. 이러할 경우 OpenGL 기하 파이프라인의 목적은 세상 좌표계에 존재하는 물체들을 윈도우 좌표계가 나타내는 그림판의 어디에 어떠한 방향으로, 어떤 크기로 그

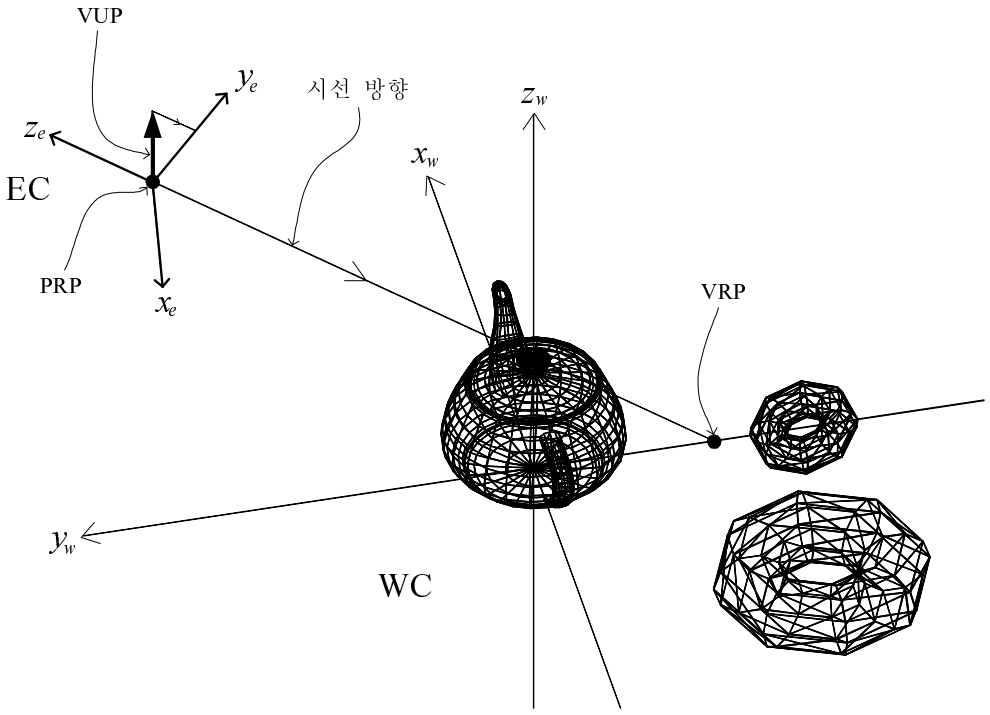



그림 2.20: 세상 좌표계에서의 카메라의 설정

릴 것인가를 결정하는 것이라 할 수 있다.

## 3.5 세상 좌표계에서의 카메라 설정

가상의 세상에 피사체들의 배치가 끝났을 때 다음으로 해야 하는 작업은 세상 좌표계에 대하여 카메라의 위치와 방향을 설정하는 일이다. 그림 2.20에서와 같이 $x_w$, $y_w$, $z_w$축으로 정의되는 세상에 한 개의 물주전자(teapot)와 두 개의 도우넛(큰 것과 작은 것 각각을 bdonut, sdonut이라 하자.)이 있다고 하자. 앞에서 잠깐 언급한 바와 같이 일반적으로 좋은 방법은 아니나 각 물체를 이루는 꼭지점 좌표들이 세상 좌표계를 기준으로 설정되어 있다고 가정하자.

**잠깐 2.4** 다면체 모델을 효과적으로 파일에 저장하기 위하여 여러 가지 기법이 존재한다. 여기서는 문제를 간단하게 하기 위하여 가장 단순한 방법을 사용하도록 하

겠다. 즉 다면체 모델로 표현된 물체는 다각형들의 리스트로 표현하고, 각 다각형은 그 다각형의 꼭지점의 개수와 각 꼭지점의 좌표를 나열함으로써 기술할 수가 있다. 다음은 한 물체에 대한 저장 내용을 보여주고 있는데,

```
3215
3
0.000000 -6.000000 1.200000
0.000000 -6.848528 0.848528
0.235114 -6.794510 0.794510
3
0.235114 -6.794510 0.794510
0.235114 -6.000000 1.123607
0.000000 -6.000000 1.200000
4
0.235114 -6.000000 1.123607
0.235114 -6.794510 0.794510
0.380423 -6.653089 0.653089
0.380423 -6.235181 1.010134
    ⋮
```

우선 전체 다각형의 개수(3215)가 주어지고 그 다음에는 물체를 구성하는 각 다각형에 대하여 꼭지점의 개수와 꼭지점들을 나열함으로써 전체 다면체 모델이 기술된다. 여기서 중요한 것은 다각형의 꼭지점을 나열하는 순서에 일관성이 있어야 한다는 점이다. 한 기하 물체를 바깥쪽에서 바라볼 때 다각형의 꼭지점을 시계 방향으로 나열할 수도 있고, 또는 반시계 방향을 나열을 할 수도 있을 것이다. 반드시 어느 방향을 사용해야 한다는 규칙은 없으나, 한 물체를 이루는 다각형을 나열할 때 시계 방향이건 반시계 방향이건 모든 다각형에 대하여 일관성이 있어야 한다. 이는 OpenGL 프로그래밍에 있어 상당히 중요한 요구 사항이며, 또한 다른 그래픽스 작업에 있어서도 방향성에 대하여 일관성을 유지하는 것이 바람직하다.

아래의 프로그램 예에는 당분간 우리가 사용할 다면체 모델에 대한 자료 구조와 입력 함수, 그리고 OpenGL 함수를 사용한 출력 함수가 정의되어 있다. 다시 한번

강조하지만 glBegin(*)과 glEnd() 함수 사이에 glVertex*(*) 함수로 꼭지점을 나열함으로써 주어진 물체를 그리라는 명령을 내릴 수가 있다. 이러한 명령에 의해 꼭지점 좌표 및 그에 연관된 정보가 OpenGL 기하 파이프라인으로 흘러 들어가게 된다.

**프로그램 예 2.2** teapot의 입출력 함수.

```
#define MAX_VERTICES 8

typedef struct {
 int nvertex;
 float poly[MAX_VERTICES][3];
} polygon; // a simple polygon format

polygon *teapot; // pointer to teapot
int npolytp; // # of polygons in teapot

void read_teapot(void) {
 FILE *fp;
 int i, j, k;

 if ((fp = fopen("teapotw.dat", "r")) == NULL) {
  printf("Can not read the file teapotw.dat");
  exit(-1);
 }
 fscanf(fp, "%d", &npolytp);
 teapot = (polygon *) malloc(sizeof(polygon)*npolytp);

 for (i = 0; i < npolytp; i++) {
  fscanf(fp, "%d", &teapot[i].nvertex);
  for (j = 0; j < teapot[i].nvertex; j++) {
   for (k = 0; k < 3; k++)
    fscanf(fp, "%f", &teapot[i].poly[j][k]);
  }
 }
 fclose(fp);
```

```
}

void draw_teapot(void) {
 int i, j;

 for (i = 0; i < npolytp; i++) {
  glBegin(GL_POLYGON);
  for (j = 0; j < teapot[i].nvertex; j++) {
   glVertex3f(teapot[i].poly[j][0], teapot[i].poly[j][1],
              teapot[i].poly[j][2]);
  }
  glEnd();
 }
}
```

세상 좌표계에서 카메라를 설정하는 방법은 몇 가지가 있는데, 가장 간단한 방법은 void gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez, GLdouble cx, GLdouble cy, GLdouble cz, GLdouble ux, GLdouble uy, GLdouble uz); 함수를 사용하는 것이다. 이 함수를 호출하기 위해서는 아홉 개의 인자가 필요한데, 이들은 3차원 공간에서의 두 개의 좌표와 한 개의 벡터를 나타낸다. 우선 ex, ey, ez는 카메라의 위치를 지정하는데 사용된다. 그림 2.20에서 PRP가 바로 카메라의 위치가 되는데 편의상 PRP $\equiv (e_x\ e_y\ e_z)^t$와 같이 표현하자. 정확히 말해서 PRP, 즉 투영 참조점(projection reference point, PRP)은 카메라의 위치라기보다는 투영을 하는데 기준이 되는 점이라 하는 것이 정확한 표현이라 할 수가 있다. 다음 cx, cy, cz는 카메라가 어느 방향을 향해 바라볼 것인가를 결정하는 인자로서, 그림의 VRP의 좌표가 된다. VRP $\equiv (c_x\ c_y\ c_z)^t$는 뷰 참조점(view reference point, VRP)이라는 의미를 가지는데, 글자 그대로 카메라에서 바라보는데 참조하는 점으로서 PRP에서 VRP로의 직선이 카

메라에서 세상을 바라보는 시선의 방향이 된다. 일반적으로 VRP는 렌더링하고 싶은 대상의 가운데 부근에 설정을 한다.

마지막으로 세 개의 인자 ux, uy, uz는 카메라를 통하여 세상을 바라볼 때, 어느 방향이 위쪽 방향인지를 결정하는데 사용된다. 즉 PRP와 VRP를 통하여 시선의 방향이 결정이 되면, 이 시선 방향에 수직인 평면에 상이 맺히게, 즉 투영이 된다. 카메라에 대하여 정확하게 설정을 하려면 이 두 인자만으로는 충분하지 않다. 즉 카메라를 PRP에서 VRP를 바라보도록 놓았을 때 시선 방향에 해당하는 직선 둘레로 빙글빙글 돌릴 수가 있다. 이는 아직 카메라에 대한 위치와 방향이 아직 완전히 결정되지 않았음을 의미하는데, 카메라의 정확한 설정을 위하여 추가적인 값이 필요하다. OpenGL에서는 벡터 (ux, uy, uz)를 통하여 어느 방향이 카메라를 기준으로 하여 위쪽 방향인지를 설정함으로써 정확하게 카메라의 위치와 방향에 대한 정의를 마치게 된다. 이러한 벡터를 뷰 상향 벡터(view-up vector, VUP)라 한다. 조금 전에 기술한 바와 같이 시선 방향에 수직인 평면에 상이 맺히는데, 이 2차원 평면에 방향성을 결정하기 위하여 위쪽 방향(일반적으로 $y$축 방향)을 설정하면 자동적으로 오른쪽에 방향 (일반적으로 $x$ 축 방향)이 결정이 된다. 따라서 위쪽 방향을 결정하기 위한 VUP $\equiv (u_x\ u_y\ u_z)^t$로 시선 방향에 수직인 벡터를 사용해야 하지만, 이는 프로그래머가 정확하게 수직인 벡터를 계산해야 하는 부담을 주기 때문에 OpenGL에서는 대략적으로 위쪽에 해당하는 벡터를 사용하여 설정한다. 그림 2.20에서 벡터 VRP를 PRP를 지나고 시선 방향에 수직인 평면에 투영을 함으로써 생성되는 벡터가 앞으로 생성하게 될 이미지의 위쪽 방향이 되며, 자동적으로 오른쪽 방향도 결정이 되게 된다.

이렇게 함으로써 카메라의 위치(PRP), 카메라가 바라보는 방향(그림 2.20에서 $z_e$축의 반대 방향), 영상의 위쪽 방향($y_e$축 방향), 그리고 상의 오른쪽 방향($x_e$축 방

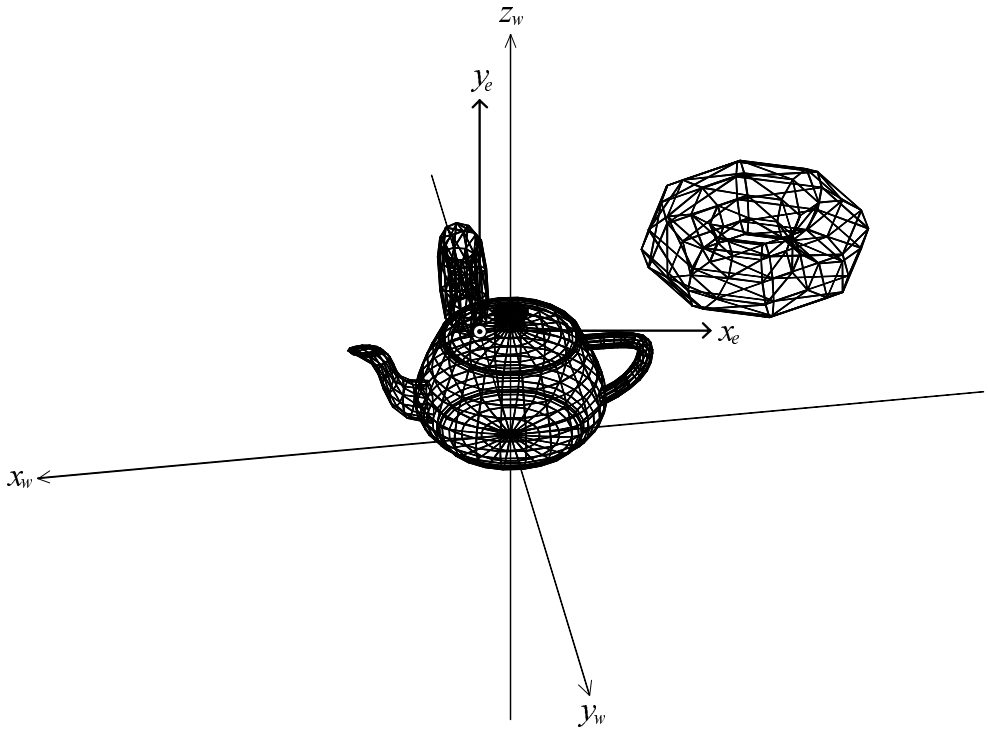
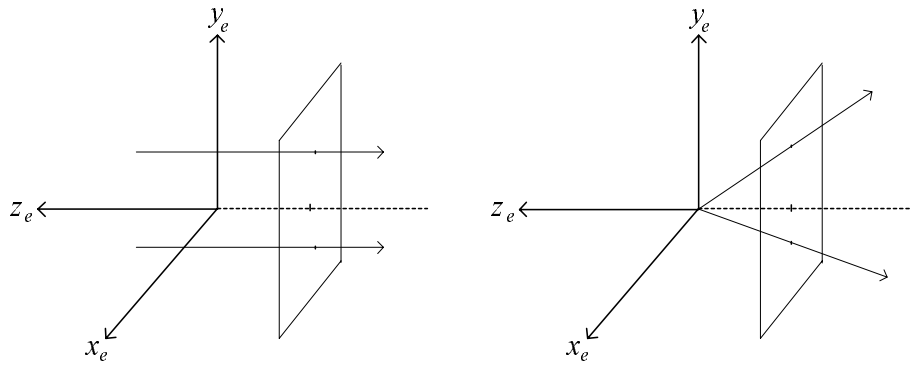



그림 2.21: 카메라에서 바라본 세상

향)이 분명하게 결정이 된다. 그림 2.21에는 이렇게 정의된 카메라에서 바라본 세상의 모습이 도시되어 있다.

## 3.6 눈 좌표계와 뷰잉 변환

아직 수학적으로 정확하게 유도를 하지는 않았지만, 카메라의 위치와 방향을 설정함으로써 세상 좌표계에 하나의 또 다른 좌표계가 생성이 된다. 그림 2.20에서 PRP가 원점에 해당하고 $x_e$, $y_e$, $z_e$축 방향에 의해 세 개의 서로 수직인 좌표축이 형성이 되는데, 이러한 좌표계를 눈 좌표계(Eye Coordinates, EC)라 한다. 눈 좌표계는 카메라 좌표계(Camera Coordinates)라 부르기도 하는데, 이는 이 좌표계가 카메라를 기준으로 설정이 된 좌표계이기 때문이다.




(a) 직교 투영 (b) 원근 투영

그림 2.22: 눈 좌표계

그림 2.22에는 OpenGL에서의 직교 및 원근 투영 각각에 대하여 눈 좌표계와의 관계가 도시되어 있다. 우선 항상 카메라가 세상을 바라보는 방향은 눈 좌표계의 음의 $z_e$축 방향이고, 투영은 $z_e$축에 수직인 평면에 이루어진다. 그림 (a)의 직교 투영의 경우 투영의 방향은 양의 $z_e$축 방향(즉 DOP $= (0\ 0\ 1)^t$)이고, 그림 (b)의 원근 투영의 경우에는 눈 좌표계의 원점을 향해 투영(즉 COP $= (0\ 0\ 0)^t$)이 된다. 또한 $x_e$축은 오른쪽 방향, $y_e$축은 위쪽 방향을 가리키도록 설정이 되어 있다. 카메라가 세상 좌표계에서 임의의 위치와 방향을 갖도록 설정할 수 있다는 점을 생각하면, 눈 좌표계는 카메라를 기준으로 하여 상당히 정규화가 된 좌표계라 할 수 있다. 세상 좌표계에서 꼭지점의 좌표가 주어졌을 때 그 점이 생성하려고 하는 이미지의 어느 부분에 나타나리라 하는 것을 예상하는 것은 쉽지가 않다. 반면에 그 점을 눈 좌표계를 기준으로 바꾸어 주면 어느 정도 화면에서의 위치를 파악할 수가 있다. 예를 들어 두 점이 있을 때 한 점의 $x_e$ 좌표가 다른 점의 $x_e$ 좌표보다 더 크다면, 이로부터 전자가 후자의 오른쪽에 위치하게 될 것이라는 사실을 예측할 수가 있다.

앞에서 강조한 바와 같이 3차원 뷰잉의 근본 목적은 카메라의 특성을 결정하였을 때 물체 좌표계를 기준으로 하여 기술된 각 꼭지점에 해당하는 윈도우 좌표계에

서의 좌표 값을 구하는 것이다. 그림 2.15의 기하 파이프라인에서 매번 다음 좌표계로 옮겨갈 때마다 좌표 값의 의미가 윈도우 좌표계에서의 의미($x_{wd}$: 윈도우 기준점으로부터 오른쪽으로 몇 화소만큼 떨어져 있는가?, $y_{wd}$: 윈도우 기준점으로부터 위쪽으로 몇 화소만큼 떨어져 있는가?, $z_{wd}$: 화면으로부터 얼마나 깊이 들어가 있는가?)에 조금씩 수렴하게 된다.

이와 같이 프로그래머가 gluLookAt(*) 함수를 통하여 눈 좌표계를 결정하면, 세상 좌표계를 기준으로 표현된 물체의 꼭지점들의 좌표가 눈 좌표계를 기준으로 변환이 된다. 이러한 좌표 변환을 뷰잉 변환(viewing transformation)이라 하는데, 이 변환에서는 물체가 움직이거나 회전하거나 하는 것이 아니라 물체는 가만히 있고, 그 물체의 꼭지점 좌표를 표현하는데 사용되는 기준 좌표계만 바꾸는 것이다. 뷰잉 변환은 아핀 변환으로서 4행 4열 행렬 $M_v$로 표현할 수 있는데, 세상 좌표계의 모든 꼭지점에 $M_v$를 곱하면 대응되는 눈 좌표계에서의 좌표를 얻을 수가 있다.

## 3.7 뷰잉 변환의 유도

### 3.7.1 눈 좌표계의 결정

그림 2.23은 gluLookAt(*) 함수를 사용하여 카메라를 설정할 때 눈 좌표계를 설정하기 위한 값들, 즉 세상 좌표계에서의 한 점 PRP와 역시 세상 좌표계에서의 세 개의 벡터 **u**, **v**, **n**을 어떻게 계산하는지를 잘 보여주고 있다. 우선 gluLookAt(*) 함수의 인자들인 PRP $= (e_x\ e_y\ e_z)^t$, VRP $= (c_x\ c_y\ c_z)^t$, VUP $= (u_x\ u_y\ u_z)^t$가 결정이 되면, 1. VPN $\equiv$ PRP $-$ VRP, 2. U $\equiv$ VUP $\times$ VPN, 3. V $\equiv$ VPN $\times$ U 순서대로 세 개의 벡터 U, V, VPN을 계산한다(여기서 $\times$는 벡터의 외적 연산을 뜻한다.). 이 때 VPN은 뷰 평면 법선 벡터(view plane normal)를 나타내는 기호로서, 시선의 반대 방향, 즉 눈 좌표계의 $z_e$축 방향이 되고, VUP과 VPN의 외적을 통하여 얻어지는 U

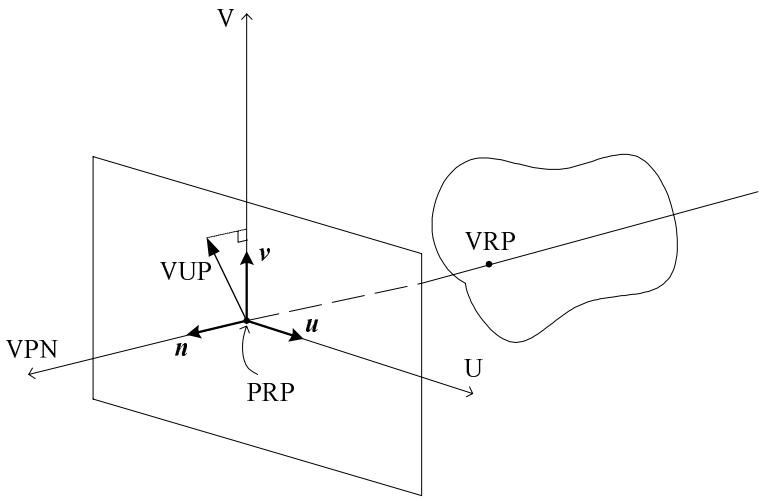



그림 2.23: 눈 좌표계의 결정

벡터는 오른쪽 방향인 $x_e$축 방향, 그리고 VPN과 U의 외적의 결과 얻어지는 V 벡터는 위쪽 방향, 즉 $y_e$축 방향이 된다. 이 세 벡터들의 길이를 1로 만든 벡터를 각각 **u**, **v**, **n**이라고 할 때 이 벡터들은 다음과 같이 결정된다: $\mathbf{u} = \frac{\mathrm{U}}{|\mathrm{U}|} \equiv (u_x\ u_y\ u_z)^t$, $\mathbf{v} = \frac{\mathrm{V}}{|\mathrm{V}|} \equiv (v_x\ v_y\ v_z)^t$, $\mathbf{v} = \frac{\mathrm{VPN}}{|\mathrm{VPN}|} \equiv (n_x\ n_y\ n_z)^t$. 여기서 **v**는 VUP 벡터를 PRP와 VRN에 의해 결정이 되는 VPN 벡터에 수직인 평면에 수선을 내렸을 때 생성되는 벡터와 평행함을 쉽게 알 수가 있고, 따라서 VUP 벡터는 원래의 목적을 달성하게 된다. 이렇게 함으로써 한 점 PRP와, 서로 수직이고 길이가 1인, 다시 말해서 정규직교계를 형성하는, 세 개의 벡터 **u**, **v**, **n**이 결정되는데, 바로 이것들이 세상 좌표계를 기준으로 한 눈 좌표계의 정의가 된다.

### 3.7.2 $M_v$의 계산

이제 눈 좌표계가 gluLookAt(*) 함수의 인자들을 사용하여 수학적으로 정확하게 정의가 되었으므로, 세상 좌표계를 기준으로 하여 주어진 점을 눈 좌표계를 기준으로 변환할 수 있다(그림 2.24). 재차 강조하지만 여기서 중요한 사실은 점이 위치가 바뀌는 것이 아니라 단지 그 점의 좌표를 기술하는데 기준이 되는 좌표계만 바뀐다는

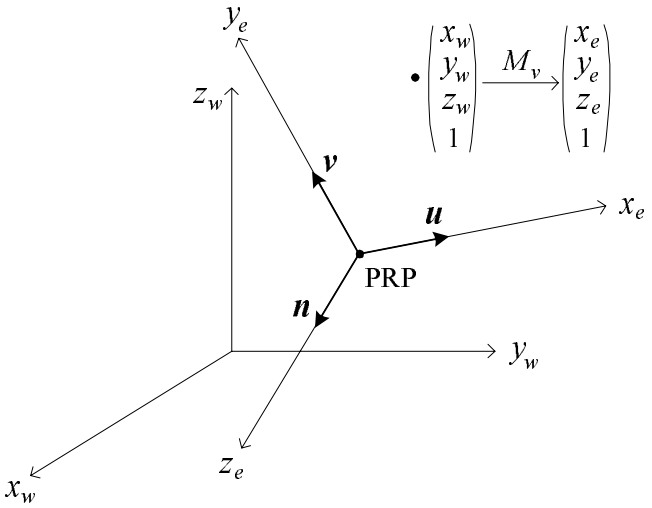



그림 2.24: 뷰잉 변환

점이다. 그림의 점 $(x_w\ y_w\ z_w\ 1)^t$은 세상 좌표계를 기준으로 표현된 좌표로서 이는 세상 좌표계에서 다음과 같은 의미를 가진다.

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + x_w \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + y_w \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + z_w \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{2.3}$$

다시 말해서 세상 좌표계의 원점 $(0\ 0\ 0)$에서 출발하여 $x_w$축 방향인 $(1\ 0\ 0)^t$ 방향으로 $x_w$만큼, 그리고 마찬가지로 $(0\ 1\ 0)^t$ 방향으로 $y_w$만큼, $(0\ 0\ 1)^t$ 방향으로 $z_w$만큼 가면 그 점이 나온다는 것을 의미한다. 마찬가지로 또 다른 좌표계인 눈 좌표계를 기준으로 하여 같은 점을 표현 하였을 때의 좌표값이 $(x_e\ y_e\ z_e\ 1)^t$라 하면 이는 세

상 좌표계의 관점에서 다음과 의미를 가지게 된다.

$$\begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} + x_e \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} + y_e \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + z_e \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \tag{2.4}$$

사실 이 두 식 (2.3)와 (2.4)는 세상 좌표계에서 같은 점을 나타내므로 두 벡터는 같은 값이어야 한다. 따라서 두 좌표간에는 다음과 같은 관계가 만족되어야 한다.

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} + \begin{bmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{bmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \end{pmatrix}$$

여기서 우리가 원하는 것은 세상 좌표계에서의 좌표가 주어졌을 때 눈 좌표계에서의 좌표를 어떻게 구하는가 하는 것인데, 이를 위하여 이 식을 정리하려면 위의 3행 3열 행렬의 역행렬을 계산하여야 한다. 그런데 이 행렬은 길이가 1이고 서로 수직인 열 벡터들로 구성되어 있으므로 직교 행렬이 되고, 따라서 이 행렬의 전치 행렬이 역행렬이 된다.

$$\begin{aligned} \begin{pmatrix} x_e \\ y_e \\ z_e \end{pmatrix} &= \begin{bmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{bmatrix}^{-1} \left( \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} - \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} \right) \\ &= \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{bmatrix} \left( \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} - \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} \right) \end{aligned}$$

오른쪽 항에서 $\begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{bmatrix} \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} \equiv \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$ 라 할 경우 위의 식은 다음과 같이 정리가 되는데,

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} = \begin{bmatrix} u_x & u_y & u_z & -t_x \\ v_x & v_y & v_z & -t_y \\ n_x & n_y & n_z & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix}$$

이 4행 4열이 바로 뷰잉 변환에 대한 행렬 $M_v$가 된다. □

행렬 $M_v$는 기하학적으로 좀 더 분명한 의미를 갖도록 다음과 같이 두 개의 행렬의 곱으로 분해할 수가 있다.

$$M_v = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \equiv R \cdot T(-e_x, -e_y, -e_z)$$

즉 뷰잉 변환 행렬 $M_v$는 회전 변환에 해당하는 행렬 $R$과 이동 변환 행렬 $T(-e_x, -e_y, -e_z)$의 곱으로 표현이 될 수 있다.

세상 좌표계에서 눈 좌표계로의 변환을 유도하는 또 다른 방법은 그림 2.24에서 눈 좌표계의 좌표축의 원점을 세상 좌표계의 원점으로 이동하여($T(-e_x, -e_y, -e_z)$), 이동된 좌표축을 세상 좌표계의 $x_w$, $y_w$, $z_w$축 둘레로 적절한 각도로 회전을 하여 두 좌표축이 일치가 되도록 하는 것이다. 이 때 $R$이 바로 해당 회전 변환의 행렬의

곱과 일치하게 된다. 어떤 방법을 사용하건 같은 내용의 행렬을 얻게 되는데, 뷰잉 변환은 앞에서 설명한 강체 변환의 전형적인 예라 할 수 있다.

한 예로 카메라를 설정하기 위하여 다음과 같이 함수를 호출하였을 때의 뷰잉 변환 행렬을 구해보자.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(2.0, 8.0, 8.0, 0.0, -4.0, 0.0, 0.0, 2.0, 0.0);
```

이 경우 PRP $= (2.0\ 8.0\ 8.0)^{\mathrm{t}}$, VRP $= (0.0\ -4.0\ 0.0)^{\mathrm{t}}$, VUP $= (0.0\ 2.0\ 0.0)^{\mathrm{t}}$ 이므로, VPN $=$ PRP $-$ VRP $= (2.0\ 12.0\ 8.0)^{\mathrm{t}}$, U $=$ VUP $\times$ VPN $= (-24.0\ 4.0\ 0.0)^{\mathrm{t}}$, V $=$ VPN $\times$ U $= (-32.0\ -192.0\ 296.0)^{\mathrm{t}}$ 이 된다. 따라서 이 벡터들을 정규화하면 새로운 좌표축의 방향에 해당하는 단위 벡터들은 다음과 같다.

$$\begin{aligned} \mathbf{u} &= (-0.98639\ 0.16440\ 0.0)^t \\ \mathbf{v} &= (-0.09033\ -0.541967\ 0.835532)^t \\ \mathbf{n} &= (0.13736\ 0.82416\ 0.54944)^t \end{aligned}$$

따라서

$$M_v = \begin{bmatrix} -0.98639 & 0.16440 & 0.0 & 0 \\ -0.09033 & -0.541967 & 0.835532 & 0 \\ 0.13736 & 0.82416 & 0.54944 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -2.0 \\ 0 & 1 & 0 & -8.0 \\ 0 & 0 & 1 & -8.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \square$$

**잠깐 2.5** 지금까지 뷰잉 변환에 대한 변환 행렬의 계산 방법에 대하여 알아 보았는데, 과연 gluLookAt(*) 함수의 수행 결과 OpenGL 시스템에서 내부적으로 어떠

한 일이 일어날까? 이것을 이해하기 위하여 우선 그림 2.15의 기하 파이프라인을 다시 살펴 보자. 이 그림은 glVertex*(*) 함수에 의해 기술된 점의 좌표가 OpenGL 기하 파이프라인을 따라 어떠한 과정을 거쳐 좌표 변환이 되는가를 보여주고 있는데, 이러한 과정에서 세 가지의 중요한 행렬이 좌표 변환에 영향을 미치고 있다. 첫째가 모델뷰 행렬 스택의 탑에 있는 행렬 $M_{MV}$인데, 일반적으로 이 행렬은 $M_{MV} = M_V \cdot M_M$과 같이 모델링 변환 행렬과 뷰잉 변환 행렬의 곱으로 표현된다. 현재 편의상 우리가 모델링 좌표계를 생각하지 않고 있으나, 일반적으로는 프로그래머가 glVertex*(*) 함수를 사용하여 모델링 좌표계에서의 꼭지점 좌표를 기술하였을 때 그 것이 $M_{MV}$에 곱해져 모델링 좌표계에서 세상 좌표계를 거쳐 눈 좌표계로 변환이 되는 것이다. 두 번째 행렬은 투영 행렬 스택의 탑에 있는 행렬 $M_P$인데 뒤에서 상세히 설명을 하겠지만 눈 좌표계로 변환된 점의 좌표에 이 행렬이 곱해져 절단 좌표계(Clip Coordinates, CC)를 거쳐 정규 디바이스 좌표계(Normalized Device Coordinates, NDC)로 변환이 되고, 다음 마지막으로 뷰폿 변환(Viewport Transformation)에 해당하는 행렬 $M_{VP}$에 의해 OpenGL 기하 파이프라인에서의 마지막 좌표계인 윈도우 좌표계로 변환이 된다.

따라서 개념적으로 생각하면 OpenGL 프로그래밍에 있어 3차원 뷰잉을 위한 기하 변환 관련 부분 코드의 목적은 렌더링 하고자 하는 물체를 이미지 윈도우의 어느 부분에 어떤 모양으로 그리게 할 것인가를 결정하는 것이고, OpenGL 시스템 관점에서 본다면 결국 이를 위하여 OpenGL 함수를 사용하여 원하는 기하 변환이 이루어지도록,

1. 적절한 모델링 변환과 뷰잉 변환을 나타내는 모델뷰 행렬 $M_{MV}$를 모델뷰 행렬 스택의 탑에 올려놓고,

2. 원하는 투영 변환에 해당하는 투영 행렬 $M_P$를 투영 행렬 스택의 탑에 올려 놓은 다음,

3. 마지막으로 뷰폿 변환에 해당하는 $M_{VP}$ 행렬을 적절하게 설정하는 것이다라고 할 수 있다.

실제로 gluLookAt(*) 함수를 수행시키면 1. 위에서 설명한 방식으로 변환 행렬이 계산이 된 후, 2. 이 행렬이 현재 행렬, 즉 현재 행렬 스택의 탑에 있는 행렬의 오른쪽에 곱해진다. 이 함수의 목적은 뷰잉 행렬 $M_V$를 설정하는 것이므로, 현재 행렬 스택이 모델뷰 행렬 스택으로 지정되어 있지 않다면 먼저 glMatrixMode(GL_MODELVIEW); 문장을 수행하여 현재 형렬 스택을 모델뷰 행렬 스택으로 바꾸어 주어야 한다. 초기에는 스택의 행렬이 단위 행렬로 초기화가 되어 있으나, 반복해서 렌더링을 할 경우 glLoadIdentity() 함수를 수행 시켜 단위 행렬로 초기화 한 후 gluLookAt(*) 함수를 호출하는 것이 좋은 방법이라 하겠다. 다음에 설명을 하겠지만 이 행렬 스택에는 뷰잉 변환 행렬 $M_V$와 모델링 변환 행렬 $M_M$이 $M_{MV} = M_V \cdot M_M$과 같은 형태로 저장이 되어야 한다. 항상 변환 행렬은 현재 행렬의 오른쪽에 곱해지므로, 먼저 수행이 되는 변환에 해당하는 행렬이 나중에 곱해져야 한다. 따라서 모델뷰 행렬 스택을 설정을 하려면 우선 뷰잉 변환에 해당하는 코드가 먼저 나온 후 모델링 변환에 해당하는 코드가 나와야 한다.

아래의 프로그램 예 2.3에는 OpenGL에서의 기하 변환에 관련된 전형적인 코드가 예시되어 있는데, 크게 뷰폿 변환, 투영 변환, 뷰잉-모델링 변환으로 이루어져 있음을 알 수 있다. glViewport(*) 함수의 수행 결과 $M_{VP}$의 내용이 설정이 되고, 다음 세 줄의 투영 변환 관련 문장의 수행 결과 원하는 $M_P$가 투영 행렬 스택의 탑에 탑재된다. 그 다음에 나오는 코드가 뷰잉 변환과 모델링 변환에 관련된 부분이다.

이 예에서는 `draw_object`$i$`()` $(i = 1, 2, 3, 4)$ 함수를 통하여 네 개의 물체를 그리고 있다. 일반적으로 각 물체들은 서로 독립적으로 설계되기 때문에, 서로 다른 모델링 좌표계를 사용하고 따라서 물체마다 적절한 모델링 변환을 사용하여 세상 좌표계에 배치를 하여야 한다. 이 때 한 정지 이미지에 대하여 카메라는 고정이 되어 있으므로 공통의 뷰잉 변환을 사용하고, 단지 해당 물체에 대하여 적절한 모델링 변환을 수행하면 된다. 3.3절에서 설명한 기하 변환 관련 함수들의 용도와 수행 원리를 잘 이해를 했다면 각 `draw_object`$i$`()` 함수의 수행 직전의 모델뷰 행렬 스택의 내용을 어렵지 않게 알 수 있을 것이다. 바로 각 함수 수행 직전에 이 스택의 탑에 있는 행렬이 해당 물체를 위한 모델뷰 행렬 $M_{MV}$가 되는 것이다.

**프로그램 예 2.3** 3차원 뷰잉 관련 코드.

```
glViewport(0, 0, 400, 300); // Viewport transformation

glMatrixMode(GL_PROJECTION); // Projection transformation
glLoadIdentity();
glOrtho(-5.0, 5.0, -5.0, 5.0, 0.0, 1000.0);

glMatrixMode(GL_MODELVIEW); // Viewing transformation
glLoadIdentity();
gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

glPushMatrix();
  glRotatef(angle, 0.0, 1.0, 0.0); // Modeling trans. 1
  ⋮
  draw_object1();
  glPushMatrix();
    glScalef(2.0, 2.0, 2.0); // Modeling trans. 2
    ⋮
    draw_object2();
    glRotatef(40.0*angle, 1.0, 0.0, 0.0); // Modeling
trans. 3
```

```
    ⋮
    draw_object3();
  glPopMatrix();
  glTranslatef(0.15, 0.3, 0.0); // Modeling trans. 4
  ⋮
  draw_object4();
glPopMatrix();
```

## 3.8 카메라의 속성 결정 및 투영 변환

### 3.8.1 카메라의 속성

표 2.1에 요약되어 있는 바와 같이 세상 좌표계에서 카메라의 위치와 방향의 설정 작업이 끝나면 다음에는 사용할 카메라의 속성을 결정해야 한다. 요즘 일반화된 줌-렌즈를 장착한 카메라로 사진을 찍을 때 카메라를 피사체 방향으로 고정을 시킨 후 줌-렌즈를 적절히 조절하여 사진에 찍힐 영역을 얼마나 넓게 할 것인지, 또는 얼마나 좁게 할 것인지를 정하게 된다. 이는 사용 렌즈의 초점 거리를 결정하는 과정이라 할 수가 있는데, 그래픽스 렌더링 과정에서도 마찬가지로 이와 유사한 계산이 수행되어야 한다. 다음 셔터를 누르면 광원에서 출발한 빛이 피사체에 반사가 되어 카메라 안으로 들어와 필름을 감광 시키게 된다. OpenGL 기하 파이프라인에서의 투영 변환(projection transformation)이 이러한 과정(카메라의 초점 거리 설정 및 촬영)에 대응이 되는데, 정확한 비교라고는 할 수 없으나 개념적으로 상당히 유사하다고 할 수 있다. 요약을 하면 OpenGL에서의 투영 변환은 사진에 나타날 정확한 대상을 결정한 후 필름으로 투영시키는 과정이라 할 수 있다.

### 3.8.2 직교 투영

다른 그래픽스 API처럼 OpenGL에서도 평행 투영과 원근 투영 두 가지 종류의 투영을 모두 지원하는데, 평행 투영의 경우 기본적으로 직교 투영을 위한 함수만 제공한다. 이는 OpenGL을 사용해서는 경사 투영을 할 수 없음을 의미하는 것은 아니다. 다만 경사 투영을 하려면 변환 행렬을 직접 계산하여 glMultMatrix*(*) 함수를 사용하여 모델뷰 행렬 스택에 올려야 한다. OpenGL에서의 직교 투영 함수는 void glOrtho(GLdouble l, GIdouble r, GLdouble b, GLdouble t, GLdouble n, GLdouble f); 인

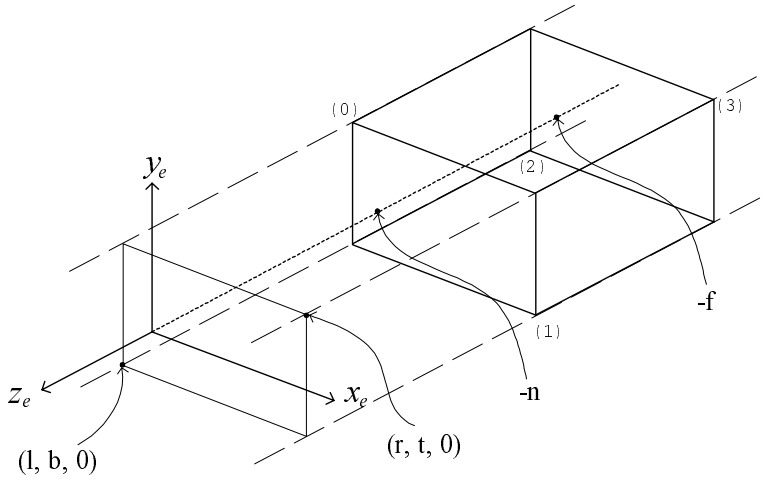



그림 2.25: glOrtho(l, r, b, t, n, f) 함수의 인자들의 의미

데 각 인자들의 의미가 그림 2.25에 설명되어 있다.

투영 변환에 관련된 카메라의 속성은 세상 좌표계에서 설정을 한 카메라를 통하여 바라보이는 공간에서 결정을 하는 것이 자연스럽고, 따라서 이 함수들의 인자들은 그러한 공간인 눈 좌표계를 기준으로 하여 설정된다. 앞에서 기술한 바와 같이 눈 좌표계에서 카메라는 음의 $z_e$축 방향을 향해 바라보고 있다. 직교 투영은 우리에게 더 익숙한 원근 투영의 특수한 경우라 할 수 있는데, 원근 투영에서처럼 바늘구멍 카메라의 바늘 구멍에 해당하는 눈이 아핀 공간의 한 점인 원점이 아니라 양의 $z_e$축 방향으로 무한대 거리만큼 떨어져 있는 투영 공간의 점 $(0\,0\,1\,0)^t$에 있게 된다. 따라서 투영은 양의 $z_e$축 방향으로 평행하게 이루어진다. glOrtho(*) 함수의 첫 네 인자 l (left), r (right), b (bottom), t (top)는 눈 좌표계 공간의 $x_e - y_e$ 평면 상에서 두 모서리 (l, b, 0)과 (r, t, 0)에 의해 결정이 되는 사각형을 정의한다. 이 사각형과 투영 방향에 의해 무한대로 뻗은 직육면체 형태의 파이프가 생성되는데 바로 이 안에 있는 물체들만 투영이 된다.

일반적으로 렌더링 계산을 할 때는 이 안에 있는, $z_e$ 좌표가 음의 무한대에서 양의 무한대에 걸쳐 있는, 모든 물체를 투영을 하는 것이 아니고, 일정 범위 안에 있

는 물체에 대해서만 투영을 한다. 이는 계산 효율 및 정확도에 큰 영향을 미치는데 자세한 내용은 뒤에서 다시 다루도록 하겠다. glOrtho(*) 함수의 마지막 두 인자 n (near)과 f (far)는 바로 그러한 범위를 결정하는데 사용된다. 우선 앞의 네 개의 인자에 의해 결정되는 무한 영역을 $z_e$축에 수직인 $z_e = -n$ 평면으로 앞에서 자르고, 다음 $z_e = -f$ 평면으로 뒤에서 자르면 그림 2.25에서와 같이 직육면체 형태의 일정 영역이 결정된다. 이러한 영역을 뷰잉 볼륨(viewing volume), 또는 뷰 볼륨(view volume)이라 하는데, 이 영역 밖에 있는 물체들은 잘려 나가고 안에 있는 물체, 다시 말해서 물체를 이루는 점, 선분, 다각형들만 투영이 된다. 사실 투영 평면의 위치는 중요하지는 않지만 편의상 $z_e = -n$ 평면 위의 사각형이 상이 맺히는 필름에 해당한다고 하자. 이 때 이 사각형의 가로-세로 비율이 $r-l : t-b$이 됨을 기억하기 바란다.

앞에서 뷰잉 볼륨을 결정하기 위해서 사용하는 두 평면을 각각 앞 절단 평면(front clipping plane)과 뒤 절단 평면(back clipping plane)이라 하는데, 한 가지 주의하여야 할 점은 n과 f가 실제로 평면이 있는 지점의 $z_e$ 좌표가 아니라 원점에서 시선의 방향(음의 $z_e$축 방향)으로 떨어진 거리를 나타낸다는 사실이다. 사실 눈 좌표계는 오른손 좌표계보다는 왼손 좌표계가 더 자연스럽고, 실제로 이전의 그래픽스 라이브러리에서는 이 지점에서 왼손 좌표계를 사용하고는 했는데, 그러한 잔재가 OpenGL에 남아 있다고 할 수 있다. OpenGL 프로그래밍에 있어 가능한 한 두 절단 평면의 거리는 짧게 하는 것이 좋다. 즉 렌더링을 하고자 하는 물체를 모두 포함하는 한 두 평면의 폭을 최대로 좁히는 것이 좋은데, 이는 불필요한 물체를 잘라내 투영 변환 이후의 계산량을 줄일 수 있을 뿐만 아니라, 한정된 깊이 버퍼의 용량을 최대한으로 사용할 수 있게 해준다.

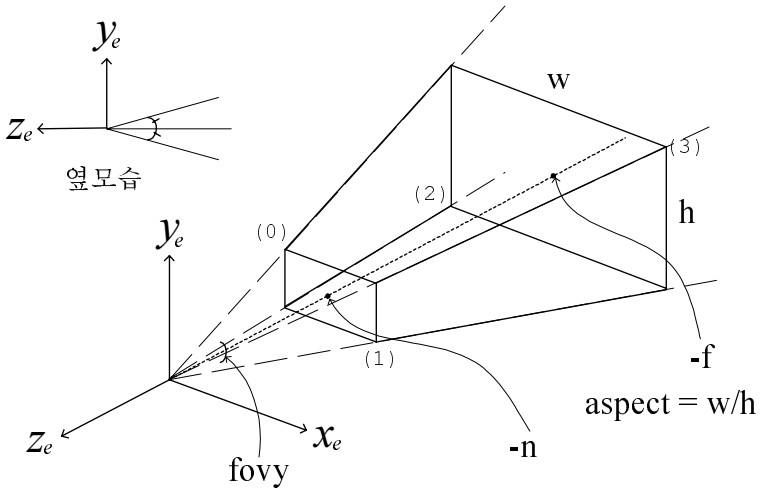



그림 2.26: gluPerspective(fovy, asp, n, f) 함수의 인자들의 의미

### 3.8.3 원근 투영

OpenGL에서는 원근 투영을 위하여 void glFrustum(GLdouble l, GLdouble r, GLdouble b, GLdouble t, GLdouble n, GLdouble f); 함수와 void gluPerspective(GLdouble fovy, GLdouble asp, GLdouble n, GLdouble f); 함수를 제공한다. 여기서는 직관적으로 사용하기가 더 쉬운 gluPerspective(*) 함수에 대하여 설명을 하도록 하겠다. 그림 2.26에는 이 함수의 네 개의 인자들이 무엇을 의미하는지가 설명되어 있다. 우선 옆모습 그림에서와 같이 원근 투영의 뷰잉 볼륨을 정의하는 위, 아래 평면이 $z_e$축에 대하여 대칭으로 벌어지는데, 첫 번째 인자 fovy가 이 각도를 나타낸다. 다음 왼쪽, 오른쪽으로 벌어지는 평면의 각도를 fovx라 할 때 이 각도는 fovy와 두 번째 인자 asp (aspect)에 의해 fovx = asp·fovy와 같이 결정이 된다. 여기서 이 두 각도는 얼마나 넓게 세상을 바라볼 것인가를 결정하는데, 이러한 각도를 피사계 범위(field of view)라 한다. 일단 이 두 각도에 의하여 네 개의 평면이 결정이 되면, 이들에 의해 무한대로 뻗어나가는 피라미드 형태의 사각뿔이 형성된다. 다음 직교 투영에서와 같이 마지막 두 인자 n과 f에 의해 정의되는 절단 평면으로 앞뒤에서 자르면 위 부분이 잘려나

간 피라미드 형태의 영역이 형성되는데, 이 영역이 바로 원근 투영에서의 뷰잉 볼륨이 된다. gluPerspective(*) 함수는 상하좌우 대칭인 뷰잉 볼륨을 정의하는데, 직교 투영에서와 같이 편의상 앞 절단 평면 상의 사각형에 상이 맺힌다고 가정하자. 이 사각형의 가로, 세로 길이를 각각 $w$, $h$라 하면 인자 asp는 asp $= \frac{w}{h}$와 같이 상이 맺히는 필름의 가로-세로 비율을 의미하게 된다.

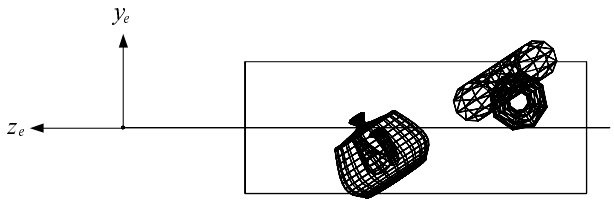
원근 투영은 눈이 아핀 공간의 한 점인 원점에 있는 경우로서 투영의 결과 원근감이 생기게 된다. 이는 우리가 실제로 물체를 바라볼 때 멀리 있는 물체가 상대적으로 작게 보이는 것과 같은 자연스러운 현상이므로 일반적으로 가상의 세상의 장면을 렌더링 할 때 원근 투영을 사용한다. 한 가지 주의를 하여야 할 점은 피사계 범위를 필요 이상 크게 해서는 안 된다는 사실이다. 이 값이 클 경우 더 넓은 영역의 물체가 필름으로 투영이 되고 그 결과 거리에 따른 물체의 심한 왜곡 현상이 발생하게 된다. 이것은 광각 렌즈를 사용할 때, 또는 극단적인 경우로 어안 렌즈를 사용할 때 물체가 뒤틀어지는 것과 같은 이유이다. 따라서 자연스러운 원근감을 원한다면 비교적 작은 피사계 범위 값을 사용하는 것이 필요하다.

요약을 하면 OpenGL에서의 투영 변환을 위해서는 우선 1. 직교 투영과 평행 투영 중 어떤 종류의 투영을 사용할 것인가와 2. 실제로 필름에 투영되는 영역인 뷰잉 볼륨을 어떻게 정의할 것인가를 결정해야 한다. glOrtho(*) 함수, gluPerspective(*) 함수, 그리고 glFrustum(*) 함수 등을 수행하면 다른 기하 변환 관련 함수들과 같이 해당하는 투영 행렬 $M_P$를 계산하여 현재 행렬의 오른쪽에 곱해준다. 따라서 이 함수들을 원래의 목적대로 사용을 하기 위해서는 다음과 같이 현재 스택을 투영 행렬 스택으로 바꾸어 주어야 한다.

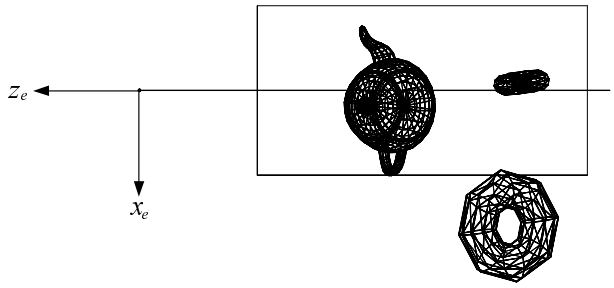
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

### 3.8.4 투영 변환의 예

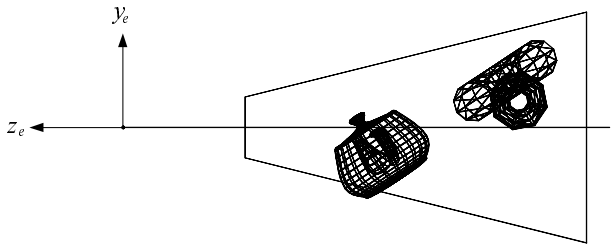
130쪽의 예에서 설정한 뷰잉 변환에 이어서 다음과 같은 투영 변환에 대하여 생각해보자. 첫 번째 예는 glOrtho(-3.6, 3.6, -2.7, 2.7, 5, 19.0);와 같이 직교 투영을 하는 경우이고, 두 번째 예는 gluPerspective(28.0, 800.0/600.0, 5.0, 19.0);처럼 원근 투영을 하는 경우이다. 그림 2.27은 이 두 문장의 수행 결과 생성되는 뷰잉 볼륨을 보여주고 있고, 그림 2.28은 그러한 뷰잉 볼륨에 대하여 절단을 한 후의 모습을 보여준다. 마지막으로 그림 2.29는 각 투영에 대하여 뷰잉 볼륨 안의 기하 프리미티브들이 필름에 투영이 된 모습을 보여주고 있는데, 원근 투영을 사용할 경우 원근감을 분명하게 느낄 수가 있다.



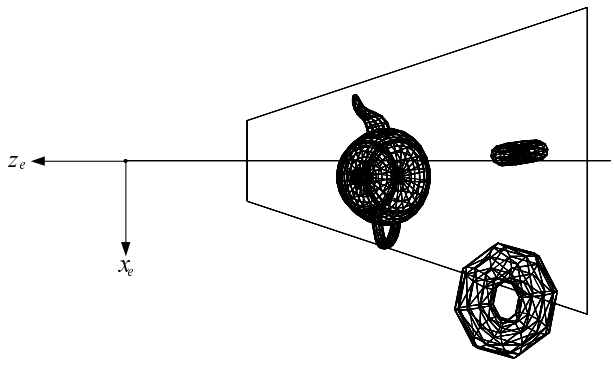

(a) 직교 투영 - 옆에서 본 모습




(b) 직교 투영 - 위에서 본 모습

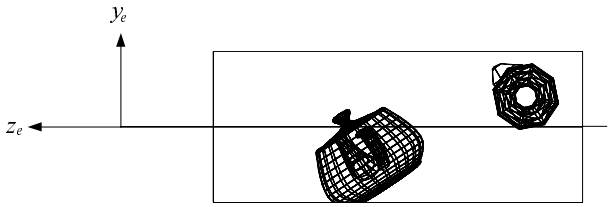



(c) 원근 투영 - 옆에서 본 모습

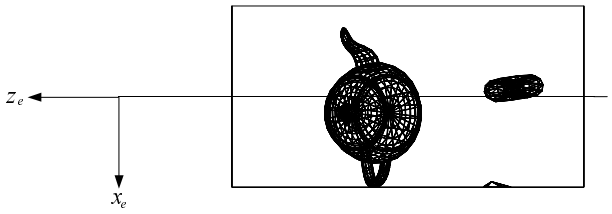



(d) 원근 투영 - 위에서 본 모습

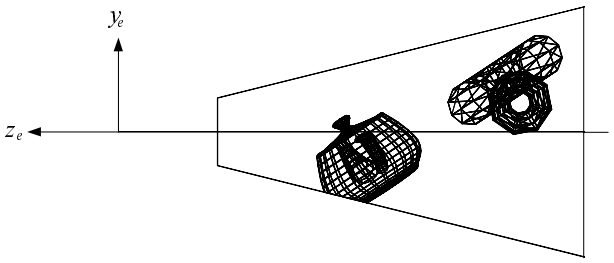
그림 2.27: 뷰잉 볼륨의 설정



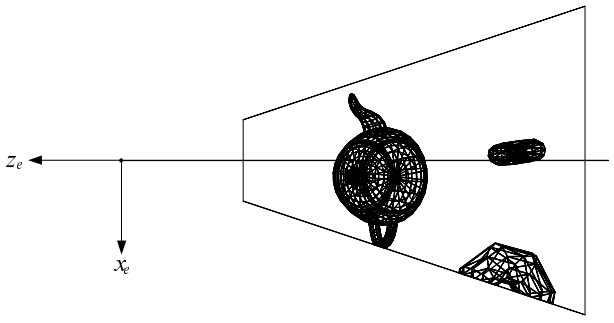

(a) 직교 투영 - 옆에서 본 모습




(b) 직교 투영 - 위에서 본 모습

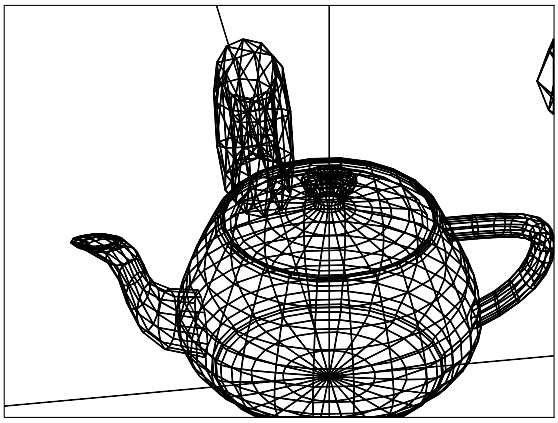



(c) 원근 투영 - 옆에서 본 모습

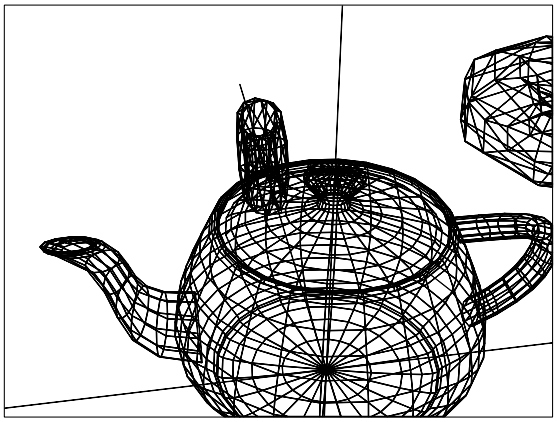



(d) 원근 투영 - 위에서 본 모습

그림 2.28: 뷰잉 볼륨에 대한 절단



(a) 직교 투영



(b) 원근 투영

그림 2.29: 필름에 투영이 된 모습

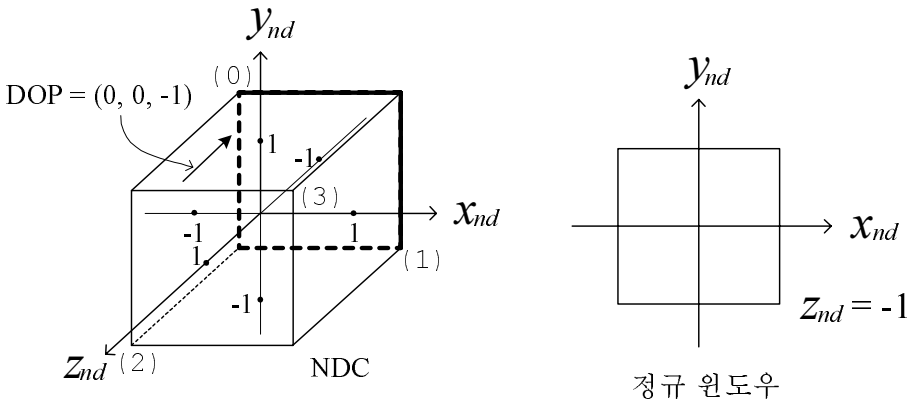



그림 2.30: 정규 디바이스 좌표계의 정의

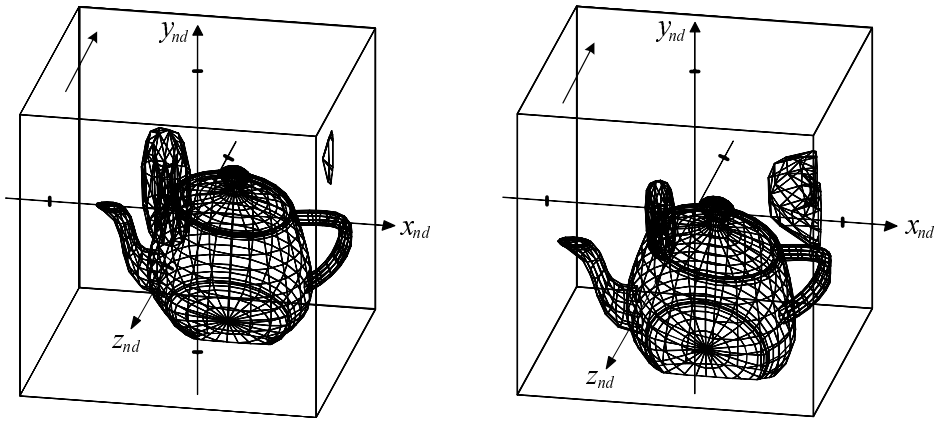
## 3.9 정규 디바이스 좌표계와 뷰 매핑

앞 절에서는 프로그래머의 관점에서 OpenGL의 투영 변환에 대하여 알아 보았는데, OpenGL 시스템을 보다 정확하게 이해를 하려면 투영 변환과 관련하여 시스템 내부에서 일어나는 일을 상세하게 파악하는 것이 중요하다. 그림 2.15의 기하 파이프라인을 다시 살펴보면 눈 좌표계에 다다른 점들이 투영 변환의 결과 절단 좌표계를 거쳐 정규 디바이스 좌표계(Normalized Device Coordinates)로 변환되어 가는 것을 알 수가 있다. 이러한 과정을 이해하기 위하여 우선 정규 디바이스 좌표계에 대하여 알아보자. 그림 2.30에는 정규 디바이스 좌표계라는 새로운 공간에 각 변의 길이가 2이고 중심이 원점인 정육면체가 도시되어 있다. 정규 디바이스 좌표계에서는 이 육면체 안의 공간만 고려를 한다. 즉 이 좌표 공간은 $[-1,1]\times[-1,1]\times[-1,1]$ 영역 안에서만 정의가 되고, 또한 그 안의 점들은 $\mathrm{DOP} = (0\ 0\ -1)^t$ 방향으로 $z_{nd} = -1$ 평면 상의 정사각형(굵은 선으로 표시된 사각형)으로 직교 투영이 되는 특징을 가진다.

앞에서 살펴본 바와 같이 눈 좌표계에서 투영 변환에 관련된 속성을 정의한 결과 그에 따른 뷰잉 볼륨이 결정이 되었다. 그 다음에 개념적으로 필름에 해당하는

앞쪽의 사각형에 투영이 일어난다고 하였는데, 실제로 OpenGL 시스템 내부에서는 약간 다른 방식으로 계산이 수행된다. 즉 어떤 투영을 사용하건 간에 물체를 구성하는 점들이 직접 2차원 평면 상으로 투영이 되는 것이 아니라, 뷰잉 볼륨에 해당하는 눈 좌표계 공간의 특정 영역이 바로 정규 디바이스 좌표계의 정육면체 영역으로 매핑이 된다. 이러한 과정을 뷰 매핑(view mapping)이라 하는데, 아직 3차원 공간으로부터 2차원 평면으로 투영이 되는 것이 아니고 3차원 공간의 특정 영역이 또 다른 3차원 공간의 대응 영역과 일치가 되도록 좌표계간에 변환이 일어나는 것이다. 그림 2.25와 2.26의 뷰잉 볼륨을 보면 모서리를 구별하기 위하여 (0)부터 (4)까지 번호를 붙였는데, 각 뷰잉 볼륨의 모서리 번호가 그림 2.30의 정육면체의 모서리 번호와 일치하도록 공간의 매핑이 일어난다. 이렇게 눈 좌표계 공간이 정규 디바이스 좌표계 공간으로 매핑이 되는 과정에서 공간의 찌그러짐 현상이 일어나게 된다.

3.8.4절의 예에서와 같이 투영 변환을 위한 OpenGL 함수를 수행시켰을 때 각 투영에 해당하는 뷰 매핑이 일어난다. 그림 2.31은 두 가지 투영 종류에 따른 뷰 매핑 이후의 정규 디바이스 좌표계의 모습을 보여주고 있다. 앞에서 언급한 바와 같이 일단 이 공간으로 변환이 되면, 각 프리미티브들을 이루는 점들은 안쪽의 정사각형으로 직교 투영이 된다. 그림 2.32는 이 정육면체를 옆에서 본 모습과 위에서 본 모습을 보여주고 있는데, 그림 2.28의 해당 그림과 비교를 해보면 뷰 매핑의 과정을 좀 더 명확히 이해를 할 수 있을 것이다. 직교 투영의 경우(그림 (a)와 (b))는 비교적 단순한 변환임을 알 수가 있는데, 기본적으로 한 직육면체에서 정육면체로의 변환인 것이다. 원근 투영의 경우(그림 (c)와 (d)) 직교 투영의 경우와는 달리 좀 더 복잡한 변환을 사용하여 뷰 매핑이 일어난다. 무엇보다도 사다리꼴의 단면이 정사각형 형태의 단면으로 매핑이 일어나고 있는데, 눈 좌표계 공간에서 원점을 향해 날아들던 투영선 들이 뷰 매핑의 결과 정규 디바이스 좌표계 공간에서는 음의 $z_{nd}$ 축

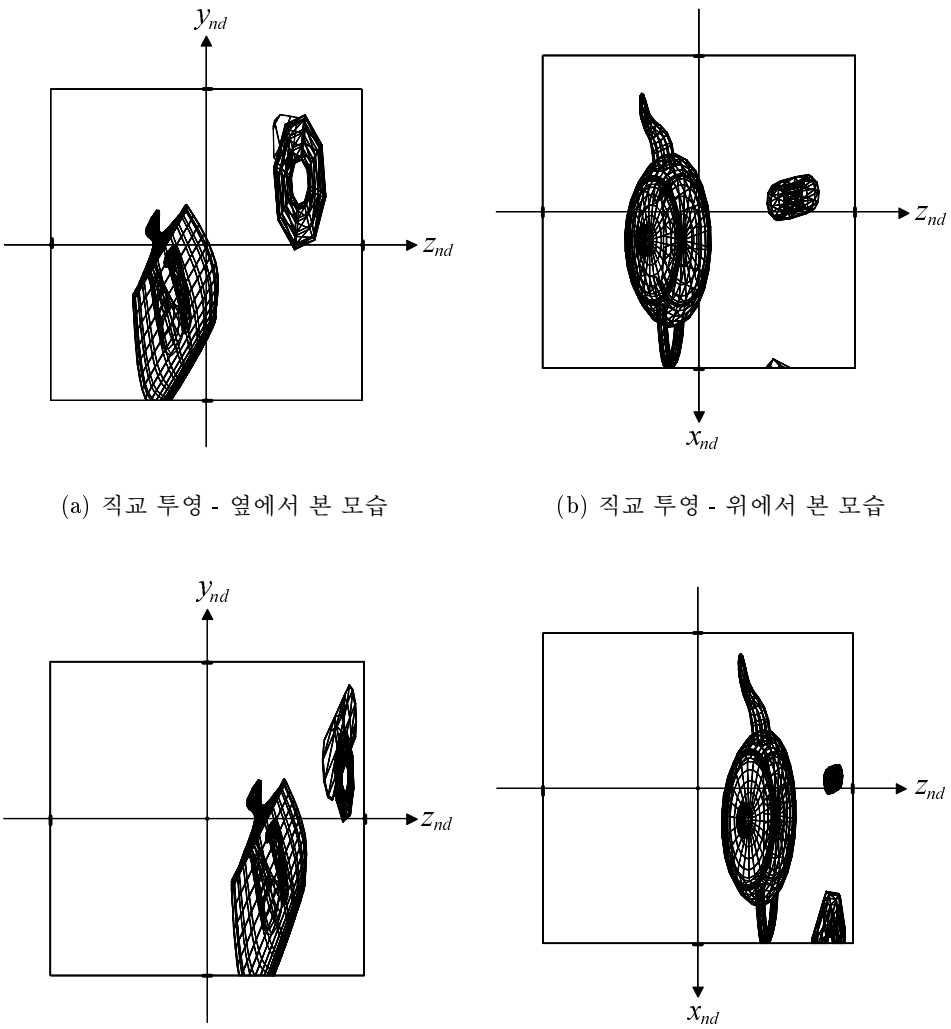



(a) 직교 투영 (b) 원근 투영

그림 2.31: 뷰 매핑 후의 정규 디바이스 좌표계 공간의 모습

과 평행한 방향으로 퍼지게 된다. 기하학적으로 생각을 해보면 투영선들을 서로 평행하게 만들어 주기 위하여 투영 참조점을 아핀 공간의 한 점인 원점 $(0\ 0\ 0\ 1)^t$에서 아핀 공간에는 존재하지 않는 투영 공간의 점 $(0\ 0\ 1\ 0)^t$으로 옮긴 후 $z$ 좌표의 방향을 반대로 바꾼 것임을 알 수가 있다. 여기서 눈 좌표계 공간에서는 양의 $z_e$축 방향으로 투영이 일어나나 정규 디바이스 좌표계에서는 음의 $z_{nd}$축 방향으로 투영이 일어난다는 사실을 명심하기 바란다.

이렇듯 원근 투영의 경우 투영 참조점이 원점에서 $z_e$축을 따라 무한한 거리만큼 떨어진 곳으로 옮겨짐에 따라 투영선이 평행해지고 따라서 그 결과 뷰잉 볼륨의 앞쪽 공간이 상대적으로 뒤쪽 영역보다 늘어나게 된다. 그림 2.32(c)와 (d)를 보면 정규 디바이스 좌표계 공간에서는 상대적으로 뷰잉 볼륨의 앞쪽에 있는 물체가 커지고 뒤쪽의 물체는 작아지는 현상을 쉽게 볼 수가 있다. 이는 원근 투영의 가장 큰 특징인 원근감이 생성되는 과정이라 할 수가 있다. 또 하나 정규 디바이스 좌표계 공간의 정사각형 영역을 보면 원근 투영을 사용할 때 물체들이 상대적으로 오른쪽으로, 즉 양의 $z_{nd}$ 방향으로 밀린 것을 알 수가 있다. 이는 투영 변환에서 사용하는



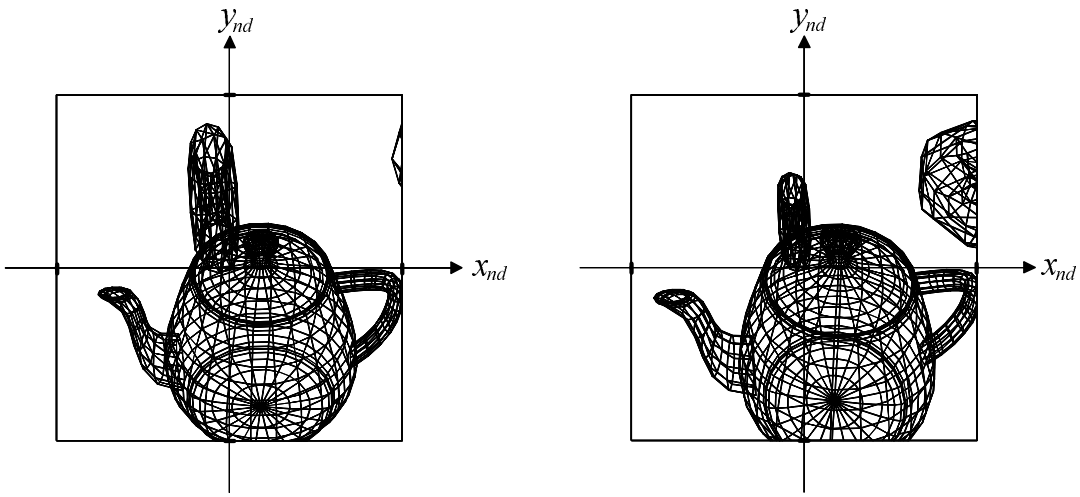

(a) 직교 투영 - 옆에서 본 모습

(b) 직교 투영 - 위에서 본 모습

(c) 원근 투영 - 옆에서 본 모습

(d) 원근 투영 - 위에서 본 모습

그림 2.32: 뷰 매핑 후의 모습




(a) 직교 투영 (b) 원근 투영

그림 2.33: 정규 윈도우로의 투영

변환 함수의 특징으로서 3.11.2절에서 자세하게 살펴보겠다.

그림 2.33은 기하 물체들이 $z_{nd} = -1$의 평면 상의 정사각형 영역에 투영된 모습을 보여주고 있다. 이 영역은 $x_{nd}$, $y_{nd}$축 각각 -1에서 1사이의 영역으로 정의된 정규 윈도우(normalized window)라 하는 정사각형인데, 이는 일종의 '정규화'된 필름이라고 할 수 있다. 그림 2.29의 내용과 비교를 해보면 $x$축 방향으로 축소가 되었음을 알 수가 있다. 이는 두 사각형의 가로-세로 비율이 다르기 때문인데, 큰 문제는 아니다. 정규 디바이스 좌표계의 정규화된 필름은 단지 필름일 뿐이고 실제로 사진을 인화지에 적절히 확대하여 인화를 할 때, 그래픽스 프로그래밍 관점에서 이야기하면 윈도우를 생성하여 뷰폿이라고 하는 영역으로 그림을 그릴 때, 다시 한번 적절한 가로-세로 비율을 갖도록 변환이 일어나게 된다.

## 3.10 OpenGL의 좌표계 및 기하 변환의 의미

이 시점에서 우리가 현재 다루고 있는 OpenGL에서의 3차원 뷰잉 과정에 대하여

다시 한번 개괄적으로 살펴보자. 앞에서도 기술한 바와 같이 뷰잉의 목적은 렌더링하고자 하는 물체의 각 꼭지점들을 프로그래머가 설정한 윈도우의 어느 지점에 떨어지게 할 것인가를 결정하는 것이다. 물체를 구성하는 프리미티브들의 꼭지점 좌표들이 기하 파이프라인을 통하여 OpenGL 뷰잉의 최종 좌표계인 윈도우 좌표계에 이르렀을 때, 무엇보다도 $x_{wd}$ 좌표는 윈도우의 기준점인 왼쪽 아래 모서리에서 몇 화소 떨어졌는지, 그리고 $y_{wd}$ 좌표는 위쪽으로 몇 화소만큼 떨어졌는지를 의미한다. 물론 윈도우 좌표계로 프리미티브에 대한 데이터가 흘러왔을 때 단지 윈도우에서의 위치인 $(x_{wd}, y_{wd})$ 좌표 외에도 더 많은 정보가 같이 붙어오게 되는데 이에 대해서는 추후에 자세하게 설명하겠다.

어떻게 보면 OpenGL에서의 기하 변환은 꼭지점의 모델링 좌표계, 또는 세상 좌표계에서의 의미가 점진적으로 윈도우 좌표계에서의 의미로 바꾸어지는 과정이라 할 수 있다. 모델링 좌표계에서의 어떤 물체의 꼭지점 좌표의 의미란 그 물체를 설계하는 공간의 기준점에서 얼마나 떨어져 있는가하는 것이다. 예를 들어 설계 기준 단위가 센티미터일 수도 있고 인치일 수도 있을 것이다. 세상 좌표계에서의 좌표의 의미는 무대에 해당하는 가상의 세상에서의 위치에 대한 정보인데, 예를 들어 세상의 기준점에서 동쪽으로 몇 미터, 남쪽으로 몇 미터, 그리고 위로 몇 미터 떨어져 있는가에 대한 정보를 제공한다. OpenGL에서 물체 좌표계라고 총칭하는 이 두 좌표계에서의 좌표는 기하 변환의 결과 최종적으로 생성되는 윈도우안에서의 좌표와는 일반적으로 아무런 상관 관계가 없다. 점들이 일단 눈 좌표계로 변환이 되면 좌표값들이 좀 더 윈도우 좌표계에서와 비슷한 의미를 같는다. 즉 눈 좌표계는 카메라를 기준으로 한 좌표계로서 한 점 $(x_e\ y_e\ z_e)^t$이 주어졌을 때 $x_e$와 $y_e$는 각각 카메라의 위치를 기준으로 하여 오른쪽으로, 그리고 위쪽으로 얼마나 떨어져 있는가를 의미한다. 따라서 눈 좌표계 공간에서의 좌표의 의미가 윈도우 좌표계 공간에서의 좌

표의 의미와 유사하게 됨을 알 수 있다. 한편 $z_e$ 좌표는 그 점이 카메라로부터 얼마나 멀리 떨어져 있는가를 나타내는데, 그 값이 작아질 수록 더 멀리 떨어진 것을 의미한다.

눈 좌표계에서 정규 디바이스 좌표계로 변환이 되면 좀 더 좌표의 의미가 윈도우 좌표계에서의 의미와 유사하게 된다. 정규 디바이스 좌표계의 경우 좌표의 의미가 눈 좌표계와 비슷한데, 차이는 그 이름이 나타내듯이 정규 디바이스 좌표계에서는 좌표 값들이 적절하게 정규화가 되어 있다는 점이다. 즉 $x_{nd}$와 $y_{nd}$는 눈 좌표계에서의 값과 비슷한 의미를 갖지만 그 값이 -1과 1 사이로 정규화가 되어 있다. 눈 좌표계에서 어떤 점의 $x_e$, $y_e$ 좌표 값이 각각 38.5와 -13.2라 할 때 아직 그 값을 가지고는 이 점이 화면의 어느 위치에 나올지 정확하게 알기가 힘들다. 반면에 정규 디바이스 좌표계에서의 $x_{nd}$, $y_{nd}$ 좌표의 값을 알면 대략적으로 화면상의 뷰폿 안에서의 위치에 대한 비율을 알 수 있을 것이다. 또한 $z_{nd}$ 좌표도 눈 좌표계에서와 같이 그 점이 투영면에서 얼마나 떨어져 있는가 하는 깊이 정보(depth information)를 나타내게 된다. 이 경우 그 값이 마찬가지로 -1과 1사이로 정규화가 되어 있는데, 눈 좌표계와는 달리 그 값이 커질 수록 멀리 떨어져 있음을 의미한다. 즉 $z_{nd}$ 값이 -1일 때가 가장 가깝고 1일 때가 가장 멀리 떨어져 있게 된다. 이 깊이 정보는 추후 은면 제거, 즉 앞에 물체에 가려 보이지 않는 프리미티브들을 제거할 때 중요하게 쓰인다.

재차 강조하고자 하는 것은 OpenGL에서의 투영 변환의 결과 우리가 직관적으로 생각하는 것처럼 3차원 공간의 점이 2차원 평면으로 투영이 되는 것이 아니라, 투영 변환 후에도 3차원적인 정보, 즉 정규 윈도우에서의 위치인 $(x_{nd}, y_{nd})$와 깊이 정보 $z_{nd}$를 보존한다는 점이다. 이러한 정보들은 3.13에서 설명할 OpenGL 기하 파이프라인에서의 마지막 변환인 뷰폿 변환에 의하여 최종적으로 윈도우 좌표계로 변

환이 된다.

## 3.11 투영 변환의 유도

이제 OpenGL에서의 투영 변환이 수학적으로 어떻게 수행이 되는지를 이해하기 위하여 각 투영 종류에 따른 변환 행렬을 유도하여 보자.

### 3.11.1 직교 변환 $M_{ortho}$의 유도

glOrtho(*) 함수를 사용한 변환은 쉽게 유도할 수가 있다. 직교 변환에 대한 뷰 매핑 과정을 다시 생각을 해보면, 직교 변환은 눈 좌표계 공간에서 두 모서리 $(l, b, -n)$과 $(r, t, -f)$에 의해 정의되는 직육면체 영역을 정규 디바이스 좌표계에서 두 점 $(-1, -1, -1)$과 $(1, 1, 1)$에 의해 정의되는 정육면체가 서로 일치하도록 해주는 크기 변환임을 알 수 있다. 한 가지 주의할 점은 그림 2.25와 그림 2.30에서의 대응 모서리((0)-(4))를 보면 알 수 있듯이 $z$축의 방향이 바뀌게 된다는 사실이다.

투영 변환은 다음과 같은 기본 변환의 합성으로 유도할 수 있다.

1. 우선 눈 좌표계에서의 직육면체의 중점 $(\frac{l+r}{2}\ \frac{b+t}{2}\ \frac{-n-f}{2})$이 원점이 되도록 이동 변환을 한다. $\Rightarrow T(-\frac{l+r}{2}, -\frac{b+t}{2}, \frac{n+f}{2})$

2. 다음 이 직육면체의 크기가 정규 디바이스 좌표계에서의 정육면체의 크기와 일치하도록 크기 변환을 한다. $\Rightarrow S(\frac{2}{r-l}, \frac{2}{t-b}, \frac{2}{f-n})$

3. 마지막으로 $z$축의 방향을 바꾸기 위하여 크기 변환의 일종이라 할 수 있는 반사 변환을 수행한다. $\Rightarrow S(1, 1, -1)$

이 세 개의 변환을 순서대로 합성하면 직교 변환 행렬 $M_{ortho}$은 다음과 같이 얻을 수 있다.

$$
\begin{aligned}
M_{ortho} &= S(1,1,-1)\cdot S(\frac{2}{r-l},\frac{2}{t-b},\frac{2}{f-n})\cdot T(-\frac{l+r}{2},-\frac{b+t}{2},\frac{n+f}{2}) \\
&= \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \square
\end{aligned}
$$

참고로 이 행렬의 네 번째 행을 보면 직교 투영 변환은 아핀 변환으로서 평행성이 유지가 되고, 또한 비율이 보존되는 변환임을 알 수 있다. 따라서 이 변환으로는 물체의 거리에 따른 원근감을 생성할 수 없음은 분명하다. 또한 이 행렬의 계수는 102쪽에서 유도한 직교 투영 행렬의 계수와는 달리 4이기 때문에 변환 후에도 3차원적인 정보를 계속하여 유지할 수가 있는 것이다.

### 3.11.2 원근 변환 $M_{pers}$의 유도

gluPerspective(*) 함수에 해당하는 변환 행렬은 직교 투영 변환보다 복잡한 유도 과정을 거친다. 원근 투영에 대한 변환을 원근 변환(perspective transformation)이라 하는데, 이 변환은 아래에서 보는 바와 같이 직교 투영과는 다른 형태의 비아핀 변환에 속한다. 변환 행렬 $M_{pers}$은 눈 좌표계의 점 $(x_e\ y_e\ z_e\ 1)^t$을 정규 디바이스 좌표계의 점 $(x_{nd}\ y_{nd}\ z_{nd}\ 1)^t$으로 적절히 변환을 시켜주어야 하는데 이에 대한 유도는 크게 두 부분으로 나누어진다.

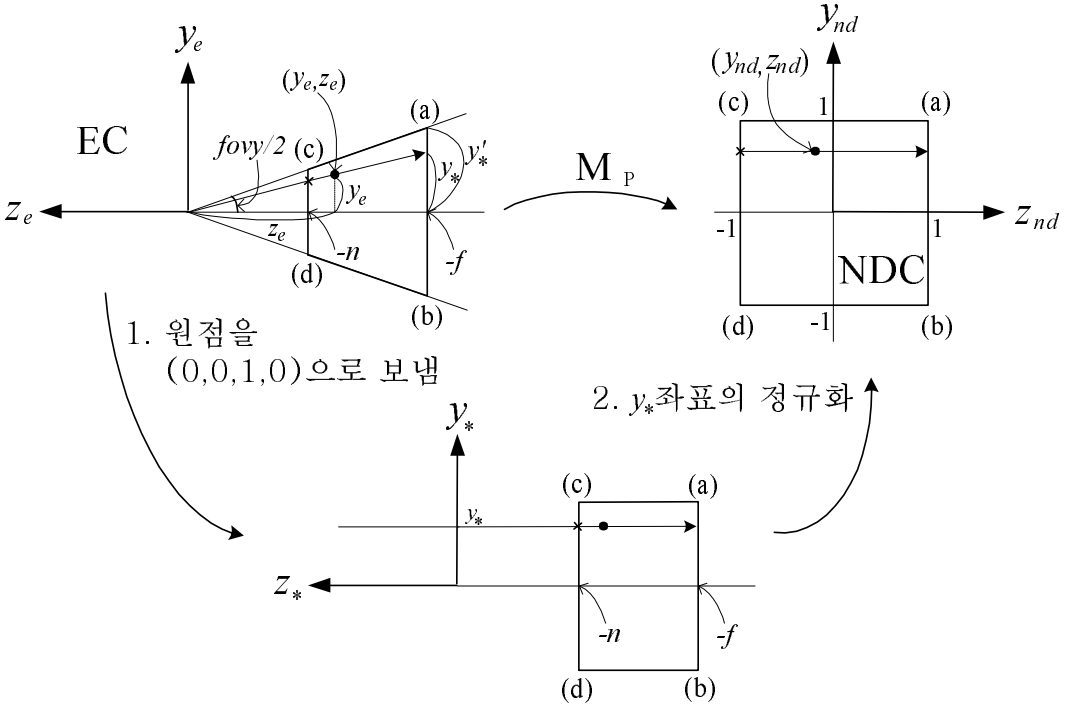



그림 2.34: 원근 변환 행렬의 유도

1. [$x_{nd}$와 $y_{nd}$의 유도] 우선 정규 디바이스 좌표계에서의 $y_{nd}$값을 눈 좌표계에서의 좌표 값 $x_e$, $y_e$, $z_e$와 gluPerspective(*) 함수의 인자 $fovy$, $asp$, $n$, $f$의 함수로 표현을 해보자. 이에 대한 유도는 다시 두 단계로 나누어 생각을 하면 편하다. 그림 2.34는 눈 좌표계(EC)와 정규 디바이스 좌표계(NDC) 공간의 뷰 매핑을 하려고 하는 두 개의 대응 영역을 옆에서 바라본 모습을 보여주고 있다. $y_{nd}$ 값을 유도하기 위해서는 우선 눈 좌표계의 원점을 무한대 점 $(0\ 0\ 1\ 0)^t$으로 보내는 작업이 필요하다. 즉 원점을 $z_e$축을 따라 무한한 거리만큼 떨어진 곳으로 보내면, 눈 좌표계 공간에서의 모든 투영선들이 수평으로 눕게 되는데(그림 아래쪽 $y_*z_*$ 좌표계), 이 때 $(y_e, z_e)$를 지나는 투영선이 $y_*$축과 교차하는 지점의 높이를 $y_*$라 하자. 눈 좌표계에 대한 그림을 보면 알 수 있듯이 $y_*$에

대하여 다음과 같은 관계를 생각할 수 있다.

$$y_e : z_e = y_* : -f$$

따라서 $y_* = -f \cdot \frac{y_e}{z_e}$이 됨을 알 수 있는데, 만약 이 점이 뷰잉 볼륨 안에 존재한다면 $y_*$의 값은 $-y'_*$와 $y'_*$사이의 값을 가질 것이다. $y'_*$ 값은 $y'_* = tan(\frac{fovy}{2}) \cdot f$가 되는데, $y_{nd}$ 값의 경우 -1과 1사이의 구간으로 정규화가 되어 있으므로 해당 $y_{nd}$ 좌표는 $y_*$를 $y'_*$에 대하여 정규화를 함으로써 얻을 수 있다.

$$y_{nd} = \frac{y_*}{y'_*} = \frac{cot(\frac{fovy}{2}) \cdot y_e}{-z_e} \tag{2.5}$$

$x_{nd}$ 값도 유사한 방법으로 유도를 할 수 있다. 이 경우 $x_* = -f \cdot \frac{x_e}{z_e}$와 $x'_* = asp \cdot tan(\frac{fovy}{2}) \cdot f$가 되고 따라서 $x_{nd}$는 다음과 같이 표현할 수가 있다.

$$x_{nd} = \frac{x_*}{x'_*} = \frac{\frac{cot(\frac{fovy}{2})}{asp} \cdot x_e}{-z_e} \tag{2.6}$$

2. [$z_{nd}$**의 유도**] 앞에서는 식 (2.5)와 (2.6)에서처럼 $x_{nd}$와 $y_{nd}$의 값을 적절하게 유도를 하였으나 여기서는 약간의 문제가 발생하게 된다. 우리가 유도하려는 원근 변환은 변환 행렬 $M_{pers}$에 눈 좌표계의 점 $(x_e\ y_e\ z_e\ 1)^t$을 곱해 정규 디바이스 좌표계의 점 $(x_{nd}\ y_{nd}\ z_{nd}\ 1)^t$을 구하는 형태를 취한다. 그렇다면 두 좌표계의 좌표 값 사이에는 선형적인 관계가 존재해야 할 것 같으나, 위에서 유도한 두 식을 살펴보면 $z_e$가 분수식의 분모에 나타나기 때문에 선형 관계를 만족시키지 않고 있다. $z_{nd}$를 유도할 때 만족 시켜야하는 필요 조건은 무엇일까? 무엇보다도 눈 좌표계 $z_e$축 구간 $[-n, -f]$가 정규 디바이스 좌표계의 $z_{nd}$축 구

간 $[-1,1]$과 일대일 대응을 하도록 하는 것이다. 이러한 변환을 하기 위해서는 선형 변환을 비롯한 여러 가지 형태의 변환을 사용할 수 있으나, 여기서는 앞에서 언급한 문제를 같이 해결할 수 있는 방향으로 적절하게 유도를 하여야 한다. 원근 변환에서는 $z_{nd}$는 상수 $\alpha$, $\beta$에 대하여 $z_{nd} = \alpha + \frac{\beta}{z_e}$의 형태의 변환을 사용한다. 이는 언뜻 보면 그 이유가 분명하지 않은 것처럼 보이나, 사실은 지금처럼 유클리드 공간에서가 아니라 투영 공간에서의 선형 변환을 생각해보면 어렵지 않게 이해를 할 수 있다. 어쨌든 이러한 형태를 사용한다고 했을 때 눈 좌표계 공간에서의 구간 $[-n,-f]$와 정규 디바이스 좌표계의 $[-1,1]$ 사이의 매핑을 적절하게 결정해야 한다. 상수 $\alpha$와 $\beta$를 구하기 위하여 구간의 양 끝점에서의 요구 조건을 사용한다. 이 과정에서 $z$축 방향이 바뀐다는 사실을 다시 상기하면 $1 = \alpha + \frac{\beta}{-f}$와 $-1 = \alpha + \frac{\beta}{-n}$의 조건을 만족시켜야 함을 알 수 있다. 이 두 식을 풀어 $\alpha$, $\beta$를 계산하면 $\alpha = \frac{f+n}{f-n}$, $\beta = \frac{2 \cdot n \cdot f}{f-n}$가 되고, 따라서 이 값을 대입하여 정리하면 $z_{nd}$에 대한 식은 다음과 같이 표현된다.

$$z_{nd} = \frac{-\frac{f+n}{f-n} z_e - \frac{2nf}{f-n}}{-z_e} \tag{2.7}$$

이제 $x_{nd}$, $y_{nd}$, $z_{nd}$에 대한 식을 유도를 하였는데, 이를 선형적인 형태로 표현을 하려면 100쪽에서 투영 변환을 유도할 때 사용한 기법을 적용하여야 한다[6]. 식 (2.6), (2.5), (2.7)를 보면 공통적으로 분자는 모두 변수 $x_e$, $y_e$, $z_e$에 대하여 선형적으로 표현이 되어 있고, 분자는 공통적으로 $-z_e$임을 알 수 있다. 이는 앞에서 동차 좌표계에 관한 설명을 할 때 언급한 투영 공간의 점과 아핀 공간의 점의 관계를 떠올리게

[6] 다시 한번 강조하면 이는 특별한 기교가 아니고 투영 공간에서의 선형 변환으로부터 자연스럽게 생각할 수 있는 것이다. 지금의 문제는 투영 공간에서 자연스럽게 선형적으로 표현할 수 있는 변환을 제한된 아핀 공간에서 유도를 하기 때문에 부자연스럽게 보이는 것일 뿐이다.

한다. 즉 $x_c = \frac{cot(\frac{fovy}{2})}{asp} \cdot x_e$, $y_c = cot(\frac{fovy}{2}) \cdot y_e$, $z_c = -\frac{f+n}{f-n} \cdot z_e - \frac{2nf}{f-n}$, $z_c = -z_e$와 같이 새로운 변수 $x_c$, $y_c$, $z_c$, $w_c$를 도입하면 이 변수들간의 관계는 다음과 같이 행렬과 벡터의 곱으로 표현할 수 있다.

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{bmatrix} \frac{cot(\frac{fovy}{2})}{asp} & 0 & 0 & 0 \\ 0 & cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} \equiv M_{pers} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} \quad (2.8)$$

이 때 일반적으로 원근 변환 시 $w_c$값은 1이 아니므로 $(x_c\ y_c\ z_c\ w_c)^t$는 $(x_e\ y_e\ z_e\ 1)^t$이 투영 변환된 점의 투영 공간에서의 좌표라 할 수 있다. 따라서 다시 아핀 공간으로 돌아오기 위해서는 $x_c$, $y_c$, $z_c$를 $w_c$로 나누어 주어야 하는데, 그 때의 값 $x_{nd} = \frac{x_c}{w_c}$, $y_{nd} = \frac{y_c}{w_c}$, $z_{nd} = \frac{z_c}{w_c}$가 원근 투영된 점의 정규 디바이스 좌표계에서의 좌표가 된다. 바로 위의 식에서의 행렬이 바로 우리가 구하려고 하는 원근 변환에 대한 변환 행렬 $M_{pers}$가 된다. □

## 3.12 OpenGL 원근 변환의 몇 가지 특징

이제 지금까지 유도한 원근 변환의 몇 가지 특징에 대하여 고려해보자. 아핀 공간인 눈 좌표계의 점 $(x_e\ y_e\ z_e\ 1)^t$에 대하여 원근 투영 변환을 하기 위하여 행렬 $M_{pers}$를 곱한 결과로 생성되는 점 $(x_c\ y_c\ z_c\ w_c)^t$는 투영 공간에서의 동차 좌표의 형태로 표현이 된다. 이러한 점들의 공간을 OpenGL에서는 절단 좌표계(Clip Coordinates, CC)라 하는데, 다시 아핀 공간인 정규 디바이스 좌표계로 돌아오기 위하여 동차 좌표의 $w_c$를 1로 만들어 준다. 이 과정에서 $x_c$, $y_c$, $z_c$를 $w_c$로 나누게 되는데, 이러한 연산을 원근 나눗셈(perspective division)이라 하였다. 원근 투영 계산은 바로 이렇게

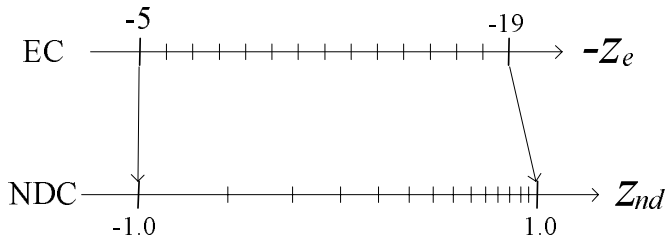



그림 2.35: $z$축 방향으로의 공간의 찌그러짐

'원근 변환 행렬의 곱셈과 원근 나눗셈'의 두 단계의 계산을 통하여 수행이 된다(그림 2.15 참조). 원근 나눗셈이라는 수학적인 연산은 투영 변환에 있어 중요한 의미를 가진다. 우선 나눗셈에 사용되는 $w_c$값의 의미를 살펴보면, $w_c = -z_e$는 변환하려고 하는 점 $(x_e\ y_e\ z_e\ 1)^t$이 카메라의 기준점으로부터 얼마나 떨어져 있는가 하는 깊이 정보를 나타내고 있다. 이 값은 점이 카메라의 앞쪽에 있다면 양수가 되고, 따라서 $x_c$와 $y_c$를 $w_c$로 나눌 경우 멀리 있는 점일 수록 큰 값으로 나누는 결과가 발생한다. 즉 점이 카메라에서 멀리 떨어져 있을 수록 나눗셈의 결과 생성되는 $x_{nd}$와 $y_{nd}$ 값은 더 작아진다. 따라서 원근 나눗셈은 물체가 멀리 있을수록 상대적으로 작게 보이는, 즉 원근감 생성의 기초가 되는 연산임을 알 수 있다. 다르게 표현하면 이 연산은 눈 좌표계 공간에서 한 점으로 모이던 투영선을 평행하게 펴주는 기하학적인 의미를 가진다고 볼 수가 있다.

다음 $z_c$를 $w_c$로 나눌 때 어떠한 현상이 발생하는지를 살펴보자. 위에서의 유도 과정을 보면 알 수 있듯이 $z_c$와 $z_e$사이에는 선형적인 관계가 존재하나, $z_c$를 $w_c$로 나누어 생성되는 $z_{nd}$를 보면 더 이상 $z_{nd}$와 $z_e$사이에는 선형적인 관계가 성립하지 않는다(식 (2.7) 참조). 이러한 비선형적인 관계로 인하여 공간이 $z$축 방향으로 부자연스럽게 왜곡되는 결과를 낳는다. 식 (2.7)의 그래프를 그려보면 쉽게 알 수 있듯이 $z_e$의 값이 일정한 속도로 변할 때 $z_{nd}$의 변화 속도는 일정하지가 않다. 그림 2.35는 $n = 5$, $f = 19$일 때, 눈 좌표계와 정규 디바이스 좌표계의 $z$축 간의 관계를 보여주

고 있다. $z_e$축에서 일정한 간격으로 값이 변할 때 $z_{nd}$축을 보면 처음에는 빨리 변하다가 점점 그 속도가 느려지게 됨을 관찰할 수 있다. 직관적으로 보면 원근 투영의 결과 눈 좌표계의 공간이 정규 디바이스 좌표계로 변환되면서 공간이 앞쪽으로 밀리는 결과를 낳는다. 그림 2.32(c)와 (d)를 다시 보면 뷰잉 볼륨 안의 물체들이 앞쪽으로 밀려간 것이 명확하게 보이고 있다. 이는 분명히 우리가 원하는 현상은 아니고, 바라지 않는 몇 가지 부작용을 낳게 되는데 그에 대한 적절한 대응이 필요하다.

정규 디바이스 좌표계에서는 깊이 정보를 내포하고 있는 $z_{nd}$값이 -1과 1사이의 구간으로 정규화 되어 있다(각각 눈 좌표계에서의 앞 절단 평면과 뒤 절단 평면에 대응). 이 깊이 정보는 계속되는 OpenGL 계산을 통하여 일반적으로 0과 1사이의 구간으로 매핑이 되고(물론 glDepthRange(*) 함수로 매핑 구간을 바꿀 수 있으나), 그 후 깊이 버퍼의 해상도에 맞게 정수 값으로 변환되어 깊이 버퍼에 저장이 된다. 깊이 버퍼에는 한 화소 당 보통 24비트에서 32비트 사이의 비트가 할당이 된다. 예를 들어 24비트가 할당이 되면 $z_{nd}$값이 존재하는 $[-1, 1]$ 구간이 $2^{24}$개의 점으로 균일하게 샘플링이 되어 저장이 된다. 그림 2.35를 다시 보면 정규 디바이스 좌표계 공간에서 균일한 간격으로 샘플링을 한다는 것은 물체가 존재하는 눈 좌표계 공간의 입장에서 봤을 때 앞쪽이 상대적으로 촘촘하게 샘플링이 되고 뒤로 갈수록 덜 촘촘하게 샘플링이 됨을 의미한다. 이러한 샘플링 간격의 불균일성은 후에 설명할 은면 제거 과정에서 사용되는 깊이 버퍼링의 정확도에 안 좋은 결과를 낳을 수가 있다. 따라서 프로그래머의 관점에서 볼 때 이 문제를 극복하기 위하여 할 수 있는 것은 불필요하게 앞 절단 평면과 뒤 절단 평면의 간격을 벌이지 말고, 가능한 한 투영 변환의 인자 $n$과 $f$ 값의 간격이 좁아지도록 노력하여야 한다.

깊이 정보를 샘플링 할 때의 정밀도는 깊이 버퍼에 할당된 메모리의 크기에 의존하는데, 이러한 메모리의 사용에는 제한이 있으므로 샘플링 되는 구간의 폭을 좁

혀 정확도를 높이려는 노력이 필요하다. 특히 $n$이 0에 가까운 값이라면 $z_e$가 0에 가까운 점도 절단되지 않고 투영이 되면서 원근 나눗셈 계산에서 절대값이 매우 작은 $w_c = -z_e$로 나눗셈이 일어난다. 우리가 잘 알다시피 절대값이 매우 작은 값으로 나눗셈을 할 경우 그 계산이 수치적으로 매우 불안정해지므로, 렌더링 인자 설정에 문제가 없는 한 $n$은 가능한 한 크게, 즉 앞 절단 평면은 앞쪽으로 충분히 미는 것이 필요하다. 또한 $z$축 방향으로의 공간의 왜곡은 뒤에서 설명할 꼭지점에 연관된 여러 데이터 값의 보간에 있어 부정확한 결과를 낳게 되는데 이에 대해서는 기하 프리미티브의 래스터화 과정을 설명할 때 다시 고려하겠다.

마지막으로 투영 변환과 관련하여 한 가지 더 언급을 하면, 원근 투영의 경우 변환되는 점이 투영 계산 과정에서 아핀 공간(눈 좌표계)에서 출발하여 일단 투영 공간(절단 좌표계)으로 갔다가 다시 아핀 공간(정규 디바이스 좌표계)으로 돌아온다. 직교 투영을 포함한 평행 투영의 경우 어떠한 지를 살펴 볼 필요가 있다. 앞에서 직교 투영 변환 행렬의 유도 과정에서 직교 투영은 아핀 변환임을 알 수 있었는데, 사실 모든 평행 투영이 아핀 변환이다. 즉 투영 공간을 거쳐갈 필요가 없이 직접 아핀 공간(눈 좌표계)에서 아핀 공간(정규 디바이스 좌표계)으로 좌표가 변환이 된다. 어떻게 보면 평행 투영과 원근 투영은 꼭지점의 좌표가 서로 다른 길을 거쳐 정규 디바이스 좌표계로 흘러간다고 할 수 있다. 따라서 이전에 많이 사용되었던 HP의 Starbase와 같은 그래픽스 라이브러리의 기하 파이프라인에서는 개념적으로 이 부분에서 두 갈래로 갈라졌다가 다시 합치게 된다. OpenGL의 경우에는 그림 2.15에서 같이 직교 투영과 원근 투영 모두 같은 길을 따라 데이터가 흘러간다. 사실 아핀 공간은 투영 공간에 포함되기 때문에[7], OpenGL에서는 직교 투영의 경우에도 개념적으로 투영 공간인 절단 좌표계를 거쳐간다고 생각을 한다. 이렇게 함으로써 시스

[7] 아핀 공간의 점들은 $w$ 좌표 값이 1인 투영 공간의 점들로 구성되어 있다.

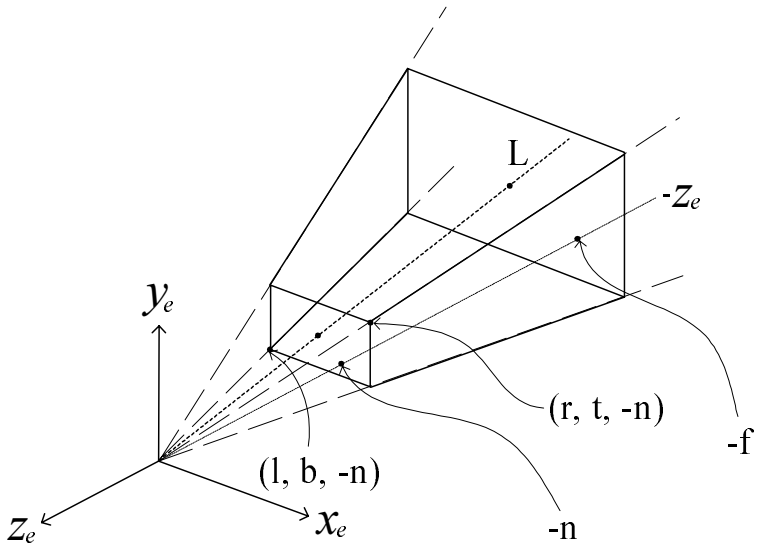



그림 2.36: glFrustum(l, r, b, t, n, f) 함수의 인자들의 의미

템 구현 입장에서 보면 각 경우를 나누어 생각을 할 필요가 없이 두 변환 종류에 대하여 모두 같은 방식으로 관련 부분을 구현할 수 있다는 장점이 있게 된다. 다음에 설명하겠지만 평행 투영과 원근 투영 모두 바로 이 절단 좌표계에서 뷰잉 볼륨에 대한 절단이 일어난다.

**잠깐 2.6** 지금까지 OpenGL에서의 원근 투영에 대한 함수로서 gluPerspective(*)에 대하여 알아보았다. 실제로 OpenGL에서 제공하는 기본적인 원근 투영 함수는 void glFrustum(GLdouble l, GLdouble r, GLdouble b, GLdouble t, GLdouble n, GLdouble f);으로서 gluPerspective(*) 함수보다 더 다양한 투영을 할 수 있게 해준다. 그림 2.36은 이 함수의 인자 값들이 어떤 의미를 가지는가를 보여주고 있다.

우선 앞 절단 평면인 $z_e = -n$ 평면 상에 $(l, b, -n)$과 $(r, t, -n)$ 두 점에 의하여 하나의 윈도우가 정의되고, 이에 따라 원점과 이 윈도우에 의해 무한히 뻗어나가는 사각뿔이 결정된다. 다음 두 절단 평면 $z_e = -n$과 $z_e = -f$로 $z_e$축에 수직으로 앞뒤에서 자르면 이 함수에 의해 정의되는 뷰잉 볼륨이 결정이 된다. 이 함수의 경

우 gluPerspective(*) 함수와는 달리 원점과 윈도우의 중점을 연결하는 직선(그림에서의 직선 L)이 반드시 $z_e$축가 일치할 필요가 없다. 따라서 glFrustum(*) 함수가 gluPerspective(*) 함수보다 더 넓은 범위의 원근 투영을 지원한다고 할 수 있다.

이 함수에 대한 변환 행렬 $M_{frus}$는 어떻게 유도할 수 있을까? 사실 $M_{frus}$는 $M_{pers}$를 기반으로 하여 쉽게 유도가 가능하다. 우선 glFrustum(*) 함수에 정의되는 뷰잉 볼륨을 gluPerspective(*) 함수에 의해 결정되는 뷰잉 볼륨의 형태로 변환할 수 있다면(이 때의 변환 행렬을 $M'$이라 하자.), $M_{frus} = M_{pers} \cdot M'$이 될 것이다. 따라서 문제는 어떻게 하면 이 두 개의 뷰잉 볼륨간의 매핑 변환을 구할 수 있을 것인가 하는 것인데, 이 변환은 다음과 같은 필요 조건을 만족시켜야 한다. 첫째로, 변환 전의 한 투영선 상에 존재하는 모든 점이 변환 후에도 같은 투영선 상에 있어야 하고, 또한 당연히 점들간의 전후 관계가 보존이 되어야 한다. 둘째로, 변환 전후의 두 뷰잉 볼륨에 포함되는 투영선들이 일치하여야 한다. 이러한 요구 사항은 3차원 공간에서의 쉬어링 변환을 사용하여 만족시킬 수가 있다. 즉 $x_e$축과 $y_e$축 각 방향으로 적절히 쉬어링을 하면 그림 2.36의 직선 L이 $z_e$축에 일치하게 만들 수가 있는데, 이에 해당하는 변환 행렬은 다음과 같다.

$$M' = \begin{bmatrix} 1 & 0 & \frac{r+l}{2n} & 0 \\ 0 & 1 & \frac{t+b}{2n} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

따라서 $M_{frus}$는 다음과 같이 표현된다.

$$M_{frus} = M_{pers} \cdot M' = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \square \quad (2.9)$$

설명의 편의상 glFrustum(*) 함수에 대응되는 투영 변환 행렬을 gluPerspective(*) 함수의 변환 행렬로 표현을 하였는데, 앞에서 언급한 바와 같이 glFrustum(*) 함수가 OpenGL 시스템에서 제공하는 더 기본적인 함수이다. 실제로 gluPerspective(*) 함수는 glFrustum(*) 함수를 사용하여 다음과 같이 구현이 되는데 그 내용이 맞는지 확인해보기 바란다.

```
#define PI 3.14159265
void gluPerspective(GLdouble fovy, GLdouble asp,
         GLdouble n, GLdouble f)
{
  GLdouble ll, rr, bb, tt;


  tt = n*tan(fovy*PI/360.0);
  bb = -tt;
  ll = bb*asp;
  rr = tt*asp;


  glFrustum(ll, rr, bb, tt, n, f);
}
```

## 3.13 윈도우 좌표계와 뷰폿 변환

이제 OpenGL 기하 파이프라인에서의 마지막 변환인 정규 디바이스 좌표계에서 윈도우 좌표계로의 변환에 대하여 알아보자. 물체 좌표계에서 출발한 다면체 모델의 기하학적인 정보, 즉 점, 선분, 다각형 등의 프리미티브들의 꼭지점들이 정규 디바이스 좌표계에 이르게 되면 그 좌표들은 -1과 1사이로 정규화된 값을 가지게 된다. 정규 디바이스 좌표계의 $[-1,1] \times [-1,1] \times [-1,1]$ 영역은 일종의 '정규화된 3차원 필름'으로 볼 수가 있다. $x_{nd}$와 $y_{nd}$ 좌표는 해당 점이 정규화된 2차원 윈도우의 어디에 떨어질 것인가 하는 정보를 제공한다. 이러한 2차원적인 정보와 함께 또 하나의 좌표인 $z_{nd}$는 그 점이 화면으로부터 얼마나 멀리 떨어져 있는가 하는 정보를 포함하고 있다. 이 값은 추후에 여러 꼭지점들이 같은 좌표 $(x_{nd}, y_{nd})$의 점에 투영이 될 때 어떤 점이 가깝고 어떤 점이 먼 지를 결정하는데 사용이 된다.

투영 변환의 결과 기하 물체들이 3차원 필름으로 현상이 되었을 때, 이제 사진 촬영의 마지막 단계에 해당하는 사진의 확대 및 인화 과정이 남아 있게 된다. OpenGL 관점에서 생각하면 이 과정은 프로그래머가 윈도우를 화면에 띄운 후 그 윈도우 안에 실제로 그림을 도시할 영역을 결정하여 정규 디바이스 좌표계의 내용을 이 영역으로 매핑하는 과정에 해당한다. 이렇게 3차원 필름이 확대되어 인화되는 윈도우 안의 영역을 뷰폿(viewport)이라 하는데, OpenGL에서 뷰폿은 void glViewport(GLint x, GLint y, GLsizei w, GLsizei h); 함수를 사용하여 결정한다. 그림 2.37을 보면 쉽게 알 수 있듯이 (x, y)는 설정하려는 뷰폿의 왼쪽 아래 모서

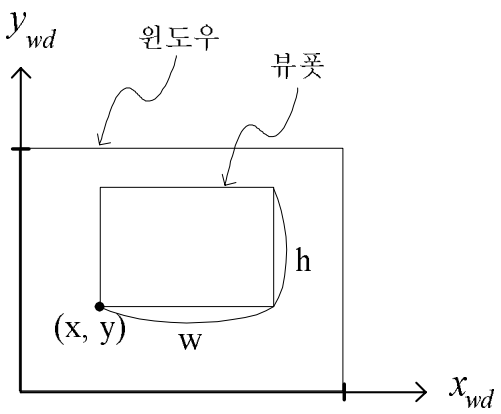



그림 2.37: 뷰폿의 설정

리의 화소의 위치이고, 이 점을 기준으로 하여 왼쪽으로 w (width) 화소만큼의 폭, 위쪽으로 h (height) 화소만큼의 높이를 설정함으로써 뷰폿이 결정된다.

한 가지 언급하고자 하는 것은 화면에 띄우려는 윈도우 크기의 결정은 OpenGL 시스템이 아니라 사용 윈도우 시스템의 소관이라는 사실이다. 즉 어떤 윈도우스 환경에서 OpenGL 프로그래밍을 하건 윈도우의 생성 및 화면으로의 매핑은 사용하는 윈도우 시스템의 윈도우 서버가 담당한다. OpenGL 시스템에서는 사용자가 윈도우 프로그래밍을 통하여, 또는 윈도우 시스템의 사용을 통하여 윈도우의 크기를 결정하였을 때, 윈도우 시스템으로부터 윈도우 크기에 대한 정보를 넘겨받아 사용을 하게 된다. 이러한 윈도우 크기에 대한 정보는 OpenGL 관점에서 보면 사용하려는 인화지의 크기에 해당하는 정보이기 때문에, 윈도우의 크기나 가로-세로 비율과 같은 정보들이 OpenGL 시스템 쪽에서 사용이 된다.

정규 디바이스 좌표계에서 윈도우 좌표계로의 변환을 뷰폿 변환(viewport transformation)이라 하는데, glViewport(*) 함수를 사용하여 뷰폿을 설정하면 정규 디바이스 좌표계 공간의 위치 정보 $(x_{nd}, y_{nd})$와 윈도우 좌표계에서의 위치 정보 $(x_{wd}, y_{wd})$간의 매핑이 결정된다. 이는 2차원 그래픽스에서 널리 사용되는 윈도우간의 매핑으로서, 두 점 $(x, y)$와 $(x+w, y+h)$에 의해 정의되는 윈도우에서 $(-1, -1)$과 $(1, 1)$에 의해 정의되는 윈도우로의 매핑에 해당한다. 뷰폿의 중점을 $(o_x, o_y)$라 하면, 즉 $o_x = x + \frac{w}{2}$, $o_y = y + \frac{h}{2}$라 하면 두 좌표계간의 관계는 다음과 같이 된다.

$$x_{wd} = \frac{w}{2} \cdot x_{nd} + o_x, \qquad y_{wd} = \frac{h}{2} \cdot y_{nd} + o_y \tag{2.10}$$

뷰폿 변환의 마지막 과정으로 $[-1, 1]$ 사이로 정규화된 깊이 정보 $z_{nd}$를 어떻게 변환할 것인가를 결정해야 한다. 우선 윈도우 좌표계에서 $z_{wd}$ 좌표는 어떤 범위의 값을

가질까? 윈도우 좌표계에서의 깊이 정보를 나타내는 이 값은 0과 1사이의 값을 가진다. 0이 화면에 가까운 쪽, 1이 먼 쪽에 해당하기 때문에, 윈도우 좌표계에서는 암시적으로 왼손 좌표계를 사용하는 셈이다. void glDepthRange(GLclampd n, GLclampd f); 함수는 두 $z$ 좌표간의 변환에 영향을 미치는데, 이 함수를 호출하면 정규 디바이스 좌표계에서의 $[-1, 1]$ 구간이 윈도우 좌표계의 [n, f] 구간으로 매핑이 된다. 따라서 $z$ 좌표간의 변환은 다음과 같이 표현되고,

$$z_{wd} = \frac{f-n}{2} \cdot z_{nd} + \frac{f+n}{2}$$

그 결과 뷰폿 변환 행렬 $M_{VP}$는 다음과 같이 표현할 수가 있다.

$$M_{VP} = \begin{bmatrix} \frac{w}{2} & 0 & 0 & o_x \\ 0 & \frac{h}{2} & 0 & o_y \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \square$$

여기서 $n$과 $f$값은 OpenGL 시스템에서 각각 0과 1로 초기화되어 있어 $[0, 1]$ 구간으로 매핑이 되는데, 이것이 보편적인 상황이기 때문에 일반적으로 glDepthRange(*) 함수는 잘 사용하지 않는다. 뷰폿 행렬의 결과 생성되는 $z_{nd}$ 좌표가 만약 0보다 작거나 1보다 크다면[8], 각각 0과 1 값으로 변환이 된다.

이제 요약을 하면 뷰폿 변환을 통하여 꼭지점 좌표를 윈도우 좌표계로 변환하였을 때, $(x_{wd}, y_{wd})$는 윈도우에서의 꼭지점의 위치 정보를 나타내고, 구간 [0, 1] 사이의 값을 가지는 $z_{wd}$는 꼭지점의 깊이 정보를 나타내게 된다. 이 값은 추후 깊이 버

[8] 이러한 상황은 glDepthRange(*) 함수를 사용하지 않으면 발생하지 않는다.

퍼의 용량에 따라 적절히 무부호 정수로 변환이 되어 깊이 버퍼링 등에 사용이 된다. 인화지에 해당하는 윈도우 좌표계 공간으로 변환이 된 후에도 역시 꼭지점에 대한 3차원적인 정보를 유지함을 알 수가 있다.

## 3.14 간단한 뷰잉을 사용하는 프로그램 예

아래에 있는 프로그램 예 2.4는 지금까지 살펴 본 3차원 뷰잉 과정을 OpenGL 함수들을 사용하여 코딩을 한 프로그램의 일부를 보여주고 있다(예제 프로그램 2.A참조). `render()` 함수는 디스플레이 컬백 함수로 등록된 `display()` 함수가 호출하는 함수로서, 화면의 내용을 다시 그려야 할 때마다 수행이 된다. 이 함수를 보면 뷰잉에 관련된 코드는 크게 세 부분으로 나누어져 있다. 먼저 뷰폿 변환이 설정이 되는데, 이 프로그램에서는 고정된 크기($800 \times 600$)를 가지는 윈도우의 전체 영역에 143쪽의 그림 2.29와 같은 내용의 그림이 그려진다.

다음은 투영 변환에 관련된 부분으로서 전역 변수 `ortho` 값에 따라 어떤 종류의 투영을 사용할 지를 결정한다. 이 변수는 키보드 컬백 함수로 등록된 `keybord(*)` 함수에서 설정이 되는데, 사용자가 'o' 키를 누르면 참, 'p' 키를 누르면 거짓 값을 갖도록 해준다. 이 변수 값을 설정한 후 `glutPostRedisplay()` 함수를 통하여 디스플레이 컬백 함수를 호출을 하고, 그 결과 `render()` 함수에서 해당하는 투영 변환을 사용하여 다시 그림을 그리게 된다.

마지막으로 뷰잉 변환에 대한 코드가 나오는데, `gluLookAt(*);` 문장을 수행시킨 직후에 우리가 머리 속에서 생각하고 있는 3차원 뷰잉에 대한 개념적인 내용이 OpenGL 함수들을 통하여 변환 행렬로 적절히 변환이 되어 각자 OpenGL 시스템에서의 적절한 위치에 저장이 된다. 지금 당분간 모델링 좌표계는 고려를 하고 있지 않기 때문에 이제 물체를 그릴 준비가 다 되었고, 따라서 세상 좌표계를 기준으

로 기술된 기하 물체를 그리라는 명령을 `draw_world()` 함수를 통하여 내리면 기하 물체에 대한 데이터들이 렌더링 파이프라인으로 흘러가게 된다.

일반적으로 3차원 뷰잉과 관련한 코드는 '뷰폿 변환 코드', '투영 변환 코드', '뷰잉 변환 및 모델링 변환 코드' 등 크게 세 부분으로 나누어진다. 지금까지 설명한 바와 같이 이 부분들은 107쪽의 그림 2.15에 있는 기하 파이프라인에서 각각 뷰폿 변환 행렬 $M_{VP}$, 투영 행렬 스택(Projection Matrix Stack), 그리고 모델뷰 행렬 스택(ModelView Matrix Stack)의 내용을 설정함으로써 원하는 렌더링을 하기 위한 준비 작업을 하는 것을 목표로 한다. 이 세 가지 종류의 코드는 상태 기계인 OpenGL 시스템의 상태를 서로 독립적으로 설정을 하기 때문에, 사실 그들간의 순서는 중요하지가 않지만 일반적으로 '뷰폿 변환' → '투영 변환' → '뷰잉 변환 및 모델링 변환'의 순서로 코딩을 한다. 물론 곧 보겠지만 프로그램 수행 시 투영에 관련된 렌더링 인자가 동적으로 바뀐다면 각 투영에 대한 코드의 수행 순서는 바뀌게 될 것이다.

한편 뷰폿 변환과 모델링 변환간에는 순서가 매우 중요하다. 이 두 변환은 하나의 행렬 스택을 공유하고, 또한 그 스택에는 $M_{MV} = M_V \cdot M_M$과 같은 내용이 저장되어야 하기 때문에 스택의 성질상 항상 모델링 변환 이전에 뷰잉 변환에 관한 코드가 수행이 되어야 한다. 참고로 `render()` 함수의 마지막 문장에서 사용한 void glFlush(void); 함수는 일반적으로 한 장면의 그림을 렌더링하는데 필요한 명령을 다 내린 후 호출을 한다. 보통 OpenGL 함수들을 사용한 명령들은 렌더링 계산의 효율을 위해서 그래픽스 버퍼에 순서대로 저장이 되어 있다가 시스템이 준비가 되면 함께 수행이 되는데, 쉽게 생각하면 이 함수의 목적은 더 이상 기다리지 말고 가능한 한 빨리 수행을 하도록 OpenGL 시스템에 요청을 하는 것이다라고 할 수 있다.

**프로그램 예 2.4** 간단한 OpenGL 뷰잉 예 1.

```
void keyboard(unsigned char key, int x, int y) {
 switch (key) {
  case 'q':
   exit(0);
   break;
  case 'o':
   ortho = 1;
   glutPostRedisplay();
   break;
  case 'p':
   ortho = 0;
   glutPostRedisplay();
   break;
 }
}

void render(void) {
 glClearColor(0.0, 0.0, 0.0, 1.0);
 glClear(GL_COLOR_BUFFER_BIT);

 glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

 glViewport(0.0, 0.0, 800.0, 600.0); // Viewport trans.

 glMatrixMode(GL_PROJECTION); // Projection trans.
 glLoadIdentity();
 if (ortho)
  glOrtho(-3.6, 3.6, -2.7, 2.7, 5, 19.0);
 else
  gluPerspective(28.0, 4.0/3.0, 5.0, 19.0);

 glMatrixMode(GL_MODELVIEW); // Viewing trans.
 glLoadIdentity();
 gluLookAt(2.0, 8.0, 8.0, 0.0, -4.0, 0.0, 0.0, 0.0, 2.0);

 draw_world();
```

```
  glFlush();
}
```

이제 약간 발전된 형태의 OpenGL 뷰잉을 생각해 보자(예제 프로그램 2.A참조). 이번에는 사용자가 윈도우를 띄웠을 때, 전체 윈도우를 네 부분으로 나누어 제1사분면에는 직교 투영을 사용한 그림을, 그리고 제3사분면에는 원근 투영을 사용한 그림을 그리도록 코딩이 되어 있다(그림 2.38). 특히 사용자가 윈도우의 크기를 임의로 바꿀 때마다 적절한 비율이 유지가 되도록 렌더링을 다시 하기 때문에, 윈도우의 가로-세로 비율이 4:3이 아닌 경우 그림이 찌그러지게 된다. 프로그램 예 2.5에는 주요 함수들이 주어져 있다. 우선 `reshape(*)` 함수는 리세이프 컬백 함수로 등록이 되어 있다. 따라서 사용자가 윈도우의 크기를 변경할 때마다[9], 윈도우 시스템이 이 함수의 두 인자 `width`와 `height`를 새로운 윈도우 크기로 지정하여 이 함수를 호출해 준다. 그 결과 현재 사용하고 있는 윈도우의 크기를 저장하기 위한 전역 변수 `W`와 `H`가 항상 정확한 정보를 유지하게 된다. 보통 윈도우의 크기가 변하면 윈도우의 내용을 다시 그려야 하기 때문에 디스플레이 컬백 함수가 호출이 되고, 그 결과 `render()` 함수를 통하여 화면의 그림을 다시 렌더링 하게 된다.

`init_OpenGL()` 함수는 그 이름이 나타내듯이 처음 프로그램이 실행될때 적절하게 OpenGL 상태를 초기화한다. 이렇게 필요한 OpenGL 상태를 설정한 후 디스플레이 컬백 함수가 `render()` 함수를 호출할 때마다 매번 전체 상태를 다시 초기화할 필요가 없이, `render()` 함수에서는 동적으로 변하는 상태만 바꾸어 렌더링을 한다. 이 예에서는 뷰잉 변환은 고정이 되어 있으므로 동적으로 변하는 뷰폿 변환과 투영 변환과는 달리 `render()` 함수에서 다시 지정을 할 필요가 없다. 이 함수에서

[9] 윈도우가 처음 화면에 매핑이 될 때도 크기가 바뀌는 것으로 간주된다.

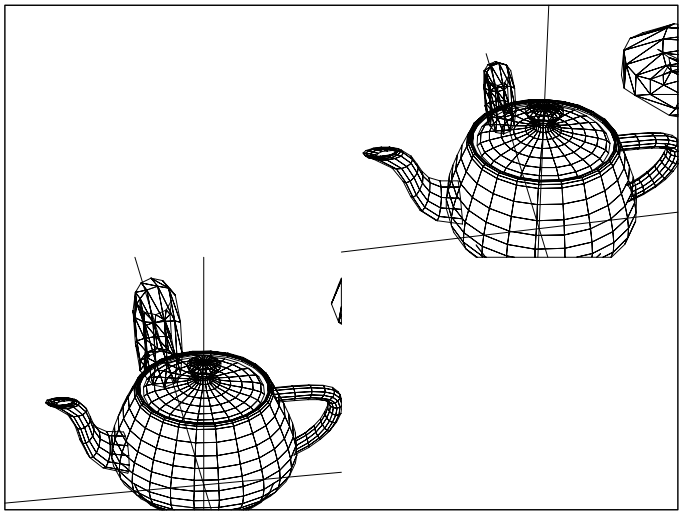


그림 2.38: 프로그램 예 2.5의 수행 결과

는 그림을 두 번 그리는데 glViewport(*) 함수로 뷰폿을 설정한 후 투영 행렬 스택을 적절히 사용하여 원하는 렌더링을 하게 된다. 이 때 이 행렬 스택의 내용이 어떻게 동적으로 변하는지를 생각해 보기 바란다.

**프로그램 예 2.5** 간단한 OpenGL 뷰잉 예 2.

```
int W, H;
      ⋮
void reshape(int width, int height) {
  W = width, H = height;
}

void init_OpenGL(void) {
  glClearColor(0.0, 0.0, 0.0, 1.0);
     ⋮
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-3.6, 3.6, -2.7, 2.7, 5, 19.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(2.0, 8.0, 8.0, 0.0, -4.0, 0.0, 0.0, 0.0, 2.0);
```

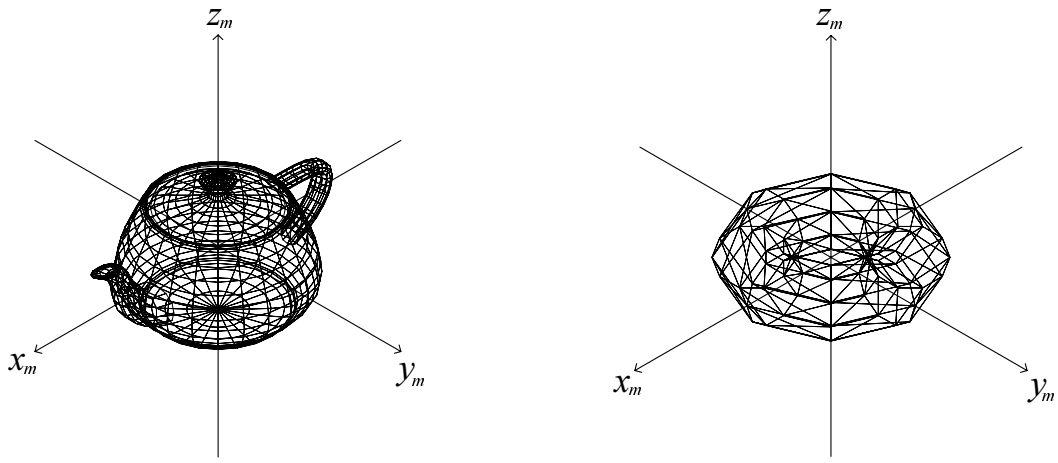
```
}

void render(void) {
 glClear(GL_COLOR_BUFFER_BIT);

 glViewport(0.0, 0.0, W/2, H/2);
 draw_world(); // The 3rd quadrant

 glViewport(W/2, H/2, W/2, H/2);
 glMatrixMode(GL_PROJECTION);
 glPushMatrix();
 glLoadIdentity();
 gluPerspective(28.0, 4.0/3.0, 5.0, 19.0);
 draw_world(); // The 1st quadrant
 glPopMatrix();
 glFlush();
}
```

## 3.15 모델링 좌표계와 모델링 변환

앞에서도 언급한 바와 같이 OpenGL 물체 좌표계는 개념적으로 모델링 좌표계와 세상 좌표계로 구성되어 있다고 생각하면 뷰잉 관련 프로그래밍을 쉽게 할 수가 있다. 지금까지는 설명의 편의상 모델링 좌표계는 존재하지 않고, 물체 좌표계는 가상의 세상이 존재하는 세상 좌표계라 가정하고 기하 파이프라인에 대하여 알아보았다. 실제로 모델링 좌표계와 그에 따른 모델링 변환은 효과적인 OpenGL 프로그래밍을 하는데 있어 상당히 중요한 역할을 한다. 여기서는 가장 기본적인 형태의 모델링 변환에 대하여 살펴보고, 4절의 프로그래밍 예에서 보다 유용한 형태의 모델링 변환에 대하여 알아보겠다.

118쪽의 그림 2.20에 도시된 세상 좌표계 공간을 다시 고려하자. 현재 가상의 세상에는 물주전자(teapot) 한 개와 똑같이 생긴 두 개의 도우넛(각각 bdonut와 sdonut)이 배치되어 있다. 앞에서는 단순히 각 다면체 모델들을 구성하는 꼭지점들이 모두 세상 좌표계를 기준으로 기술되어 있다고 가정하였다. 일반적으로는 기하 모델을 설계할 때 각 물체들에 대하여 자신의 고유한 모델링 좌표계를 사용한다. 따라서 여기서는 그림 2.39에서와 같이 물주전자에 대한 다면체 모델 한 개와 도우넛에 대한 모델 한 개를 각각의 모델링 좌표계에서 만들고, 이들을 모델링 변환을 통하여 적절히 세상에 배치를 하는 방식을 택하려 한다.

현재의 예에서는 자신의 모델링 좌표계를 기준으로 설계된 물 주전자를 $z_m$축으로 1.5만큼 위로 올려 세상에 배치를 하였다($M_M = T(0.0, 0.0, 1.5)$). 한편 큰 도우넛은 모델링 좌표계의 도우넛 모델을 두 배로 확대한 후, $x$, $y$, $z$축으로 각각 -6, -5, 1만큼 이동을 하여 세상으로 배치를 하였고($M_M = T(-6, -5, 1) \cdot S(2, 2, 2)$), 작은 도우넛의 경우 모델링 좌표계에서 $z = x$평면에 대하여 도우넛을 반사 시킨 후, 좌표를 바꾸어 $y$축 방향으로 -6만큼 이동하여 배치를 하였다($M_M = T(0, -6, 0) \cdot$




(a) 물주전자 (b) 도우넛

그림 2.39: 모델링 좌표계에서 설계된 두 개의 다면체 모델

$R(-45, 0, 1, 0) \cdot S(1, 1, -1) \cdot R(45, 0, 1, 0))$. 이렇게 개념적으로 사진 촬영의 피사체 배치에 해당하는 과정을 정확하게 수행하기 위해서는 OpenGL 함수를 사용하여 모델링 변환에 관한 행렬을 모델뷰 행렬 스택에 적절히 올려주어야 한다. 다시 말해서 사용하는 뷰잉 변환 행렬이 $M_V$이고 현재 그리려는 물체에 대한 모델링 변환 행렬이 $M_M$이라면, 모델뷰 행렬 스택의 탑에 행렬 $M_{MV} = M_V \cdot M_M$의 내용을 탑재한 후 그 물체의 모델링 좌표계에서의 모델을 사용하여 물체를 그려야 한다. 이러한 작업을 세상에 존재하는 모든 물체에 대하여 반복을 해야 하는데, 모델링 변환과 뷰잉 변환은 행렬 스택을 공유하고 따라서 서로 직접적으로 영향을 미치기 때문에 조심하여야 한다.

예제 프로그램 2.B에는 이러한 모델링 변환을 사용하는 OpenGL 프로그램이 주어져 있다. 이 프로그램에서는 우선 gluLookAt(*) 함수를 사용하여 뷰잉 변환 행렬 $M_V$를 모델뷰 행렬 스택의 탑에 올려 놓은 후, 프로그램 예 2.6에 주어진 `draw_world()` 함수를 통하여 모델링 변환을 하고 있다. gluLookAt(*) 함수를 수

행하면 이제 모델링 변환을 하기 위한 준비를 마친 셈인데, `draw_world()` 함수를 살펴보면 우선 `draw_axes()` 함수를 통하여 세상 좌표계의 좌표축을 그리고 있다. 이 때에는 특별히 어떤 모델링 변환을 하지 않으므로 이 함수에서 사용하는 꼭지점의 좌표는 세상 좌표계에서의 좌표라고 생각하면 된다. 실제로 좌표축은 세상 좌표계 공간에 존재하기 때문에 모델링 변환이 필요가 없다.

다음 물주전자를 그리기 위해서는 모델뷰 행렬 스택에 $M_V \cdot T(0.0, 0.0, 1.5)$를 올린 후 `draw_object(teapot, npolytp);` 문장을 수행 시켜야 한다. 이 때 $M_V$의 내용을 보존하기 위하여 먼저 `glPushMatrix()` 함수를 통하여 $M_V$를 현재 행렬에 복사한 후, `glTranslatef(0, 0, 1.5);` 문장을 통하여 원하는 모델뷰 행렬을 설정하게 된다. 물체를 그린 후에는 현재 행렬의 내용 $M_V \cdot T(0, 0, 1.5)$가 더 이상 필요가 없으므로 `glPopMatrix()` 함수를 통하여 제거를 한다. 따라서 모델뷰 행렬 스택은 바로 직전에 `glPushMatrix()` 함수를 호출하기 직전의 상태로 다시 돌아오게 된다.

**프로그램 예 2.6** 모델링 변환 예.

```
void draw_world(void) {
  draw_axes();

  glMatrixMode(GL_MODELVIEW);

  glColor3f(1.0, 1.0, 0.0);
  glPushMatrix();
    glTranslatef(0.0, 0.0, 1.5);
    draw_object(teapot, npolytp); // draw teapot
  glPopMatrix();

  glColor3d(1.0, 0.0, 1.0);
  glPushMatrix(); // Line (a)
    glTranslatef(-6.0, -5.0, 1.0);
    glScalef(2.0, 2.0, 2.0); // Line (b)
```

```
    draw_object(donut, npolydn); // draw bdonut
  glPopMatrix();

  glColor3d(0.0, 1.0, 1.0);
  glPushMatrix();
    glTranslatef(0.0, -6.0, 0.0);
    glRotatef(-45.0, 0.0, 1.0, 0.0); // x<->z
    glScalef(1.0, 1.0, -1.0);
    glRotatef(45.0, 0.0, 1.0, 0.0);
    draw_object(donut, npolydn); // draw sdonut
  glPopMatrix();
}
```

다음 큰 도우넛을 그리기 위해서 모델뷰 행렬 스택의 탑에 $M_V \cdot T(-6,-5,1) \cdot S(2,2,2)$를 올려야 하기 때문에, `Line (a)`부터 `Line (b)`까지에서와 같이 OpenGL 함수를 부르게 된다. 재차 강조를 하거니와 현재 행렬의 내용에 영향을 미치는 OpenGL 함수를 호출하면 항상 해당 행렬이 현재 행렬의 오른쪽에 곱해지므로, 직관적으로 먼저 수행이 되어야 하는 크기 변환에 대응되는 OpenGL 함수가 그 다음에 수행되는 이동 변환 관련 함수보다 나중에 수행되어야 한다. 마지막으로 작은 도우넛을 위한 모델뷰 행렬 $M_V \cdot T(0,-6,0) \cdot R(-45,0,1,0) \cdot S(1,1,-1) \cdot R(45,0,1,0)$을 탑재하기 위해 역시 역순으로 OpenGL 함수를 불러 준비를 마친 후 물체를 그리고 있다. OpenGL에서 glPushMatrix() 함수는 뷰잉을 위한 기하 변환이 동적으로 변할 때 주어진 행렬의 내용을 보존하기 위하여 사용된다. 반대로 glPopMatrix() 함수는 더 이상 현재 행렬의 내용이 필요하지 않을 때 그것을 제거하기 위하여 사용된다.

그러면 과연 약간은 복잡하게 보이는 모델링 변환을 사용하는 이유는 무엇일까? 우선 첫 번째 장점으로는 렌더링 프로그램 수행 시 메모리에 올려야 하는 다면체 모델의 양을 줄일 수 있다는 점을 들 수가 있다. 물론 항상 해당하는 사실은 아니나

위의 예에서 보듯이, 모델링 좌표계를 사용하지 않는 기존의 프로그램에서는 세상 좌표계를 기준으로 기술된 물주전자 모델 한 개와 도우넛 모델 두 개를 메모리에 올려야만 했다. 하지만 도우넛의 경우 실제로 동일한 물체이기 때문에 모델링 좌표계를 사용할 경우 모델을 한 번만 올린 후 적절한 모델링 변환을 사용함으로써 렌더링에 필요한 데이터의 양을 줄일 수 있다.

어떻게 보면 요즈음의 PC나 워크스테이션에는 과거와는 달리 상당한 크기의 메모리를 장착하기 때문에 이것이 메모리 사용에 있어 큰 절약은 아닌 것처럼 보일 수도 있다. 그러나 복잡한 장면, 예를 들어 5000개의 의자가 있는 극장 내부를 렌더링 한다고 할 때, 그리고 모든 의자가 같은 모양이고 각 의자가 1500개의 다각형으로 모델링 되어 있다면, 위와 같이 모델링 변환을 사용하는 것과 단순히 모든 의자에 대한 모델을 세상에 직접 배치하는 것에는 큰 차이가 있게 된다. 모델링 변환을 사용하는 것에 대한 또 다른 이점으로 한 물체를 구성하는 요소들간의 계층적 구조를 효과적으로 이용할 수 있다는 점을 들 수가 있는데, 이에 대해서는 4절에서 자세히 설명하도록 하겠다.

## 3.16 법선 벡터의 기하 변환

### 3.16.1 벡터 변환의 유도

지금까지 살펴본 기하 변환은 3차원 공간의 점에 대한 변환이었다. 점의 좌표는 물체의 기하 속성을 기술하는데 있어 공간에서의 위치 정보를 제공하는 가장 기본적인 요소이다. 컴퓨터 그래픽스 분야의 여러 작업에서 중요한 역할을 하는 또 다른 기하 요소로서 법선 벡터를 들 수가 있다. 물체 표면 상의 한 점에서의 법선 벡터는 바로 그 점에서 물체의 표면에 수직인 방향을 가리킨다. 다시 말해서 법선 벡터는 한 점에서 물체에 접한 법선 평면에 수직인 벡터인 것이다. 법선 벡터는 두 개의 방

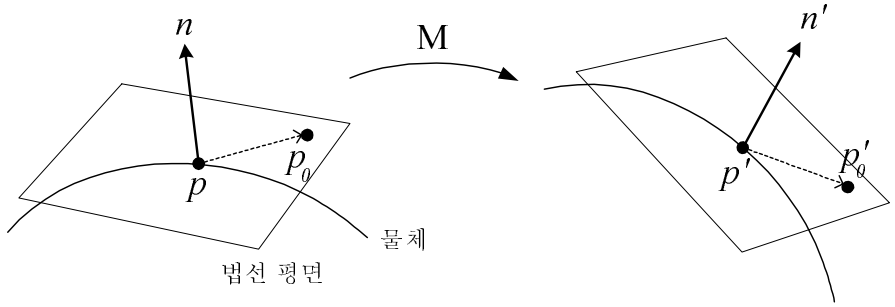



그림 2.40: 법선 벡터의 변환

향, 즉 물체의 안쪽 방향과 바깥쪽 방향, 어느 방향을 가리켜도 상관이 없으나, 일반적으로 바깥쪽 방향을 가리키도록 설정이 된다. 다면체 모델을 통하여 기하 물체를 표현할 때, 꼭지점 좌표와 함께 법선 벡터가 설정이 되면 그 물체가 기하 변환이 될 경우 점과 함께 법선 벡터도 적절히 변환이 되어야 한다. 넓은 의미에서 점과 벡터는 같은 개념으로 생각할 수가 있으나, 우리가 사용하는 유클리드 공간에서는 이 두 가지 기하 요소를 서로 다른 방식으로 처리를 해주어야 한다.

주어진 물체가 기하 변환 $M$에 의하여 변환이 될 때 법선 벡터는 어떻게 변환이 될까? 결론부터 말하면 벡터는 단순히 $M$에 좌표 값을 곱하는 점과는 달리, $M$의 역행렬의 전치 행렬 $(M^{-1})^t$에 곱해 변환을 한다. 그 이유는 아래와 같이 간단하게 증명할 수가 있다. 그림 2.40에서와 같이 주어진 점 $p = (x\ y\ z\ 1)^t$에서의 법선 벡터가 $n = (n_x\ n_y\ n_z\ 0)^t$이라고 하자[10]. 이 때 점 $p$에서의 법선 평면 상의 임의의 점 $p_0 = (x_0\ y_0\ z_0\ 1)^t$에 대하여 $n^t \cdot (p_0 - p) = 0$와 같은 관계가 성립한다. $p$, $p_0$, 그리고 $n$이 변환 행렬 $M$에 의해 각각 $p' = (x'\ y'\ z'\ 1)^t$, $p'_0 = (x'_0\ y'_0\ z'_0\ 1)^t$, $n' = (n'_x\ n'_y\ n'_z\ 0)^t$으로 변환이 된다고 하면, $p' = M \cdot p$와 $p'_0 = M \cdot p_0$로부터 $p$와

[10]앞에서 설명한 동차 좌표계의 개념을 잘 이해하였다면, 벡터의 경우 네 번째 좌표, 즉 $w$ 좌표가 자연스럽게 0이 됨을 알 수가 있을 것이다.

$p_0$를 구해 위의 관계식에 대입하여 다음과 같은 식을 얻게 된다.

$$\begin{aligned} n^t \cdot (M^{-1} \cdot p_0' - M^{-1} \cdot p') &= (n^t \cdot M^{-1}) \cdot (p_0' - p') \\ &= \{(M^{-1})^t \cdot n\}^t \cdot (p_0' - p') \\ &= 0 \end{aligned}$$

마지막 식을 잘 살펴보면 열 벡터 $(M^{-1})^t \cdot n$는 $M$에 의하여 변환된 점 $p'$과 $p_0'$에 의해 형성되는 새로운 법선 평면 상의 모든 벡터와 항상 수직이 되므로, 바로 이 벡터가 변환된 기하 물체의 새로운 법선 벡터가 됨을 알 수 있다. 따라서 $n' = (M^{-1})^t \cdot n$이 된다. □

만약 변환 $M$이 아핀 변환이라면 벡터의 변환을 좀 더 직관적으로 파악할 수가 있다. $M$의 왼쪽 위 부분의 3행 3열 부행렬을 $M_{33}$이라 하면 벡터의 변환은 다음과 같이 표현할 수 있음을 어렵지 않게 증명할 수가 있다.

$$\begin{pmatrix} n_x' \\ n_y' \\ n_z' \end{pmatrix} = (M_{33}^{-1})^t \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \tag{2.11}$$

즉 $M_{33}$의 역행렬의 전치 행렬에 대하여 유클리드 공간에서의 벡터 좌표를 곱하면 원하는 벡터를 얻게 된다. 만약 $M$이 강체 변환이라면, 즉 이동 변환과 회전 변환만으로 이루어진 변환이라면 어떠한 일이 발생할까? 이 경우에는 $M_{33}$가 직교 행렬이고, 직교 행렬의 역행렬의 전치 행렬은 자신이 되므로, $M_{33}$를 직접 곱하면 된다. 따라서 아주 쉽게 벡터가 변환이 됨을 알 수가 있다. 특히 이 경우 직교 행렬에 의한 선형 변환은 벡터의 크기를 보존하여 주므로, 만약 변환 전의 벡터의 길이가 1이라

면 변환 후에도 그 길이가 1이 됨을 알 수가 있다. 만약 $M$이 등방성 크기 변환이거나 강체 변환에 등방성 변환이 합성된 형태의 변환일 경우, 변환된 벡터는 $M$이 각 축 방향으로 $s$배 한 변환일 때 변환 전의 크기에 $\frac{1}{s}$배가 됨을 알 수 있다.

### 3.16.2 OpenGL과 법선 벡터

OpenGL 시스템에서와 같이 기하 물체가 다면체로 모델링이 되는 경우에 일반적으로 법선 벡터는 각 꼭지점에 대해서만 정의가 된다. 본 장에서와 같이 단순히 뷰잉 계산만 할 때에는 꼭지점 좌표만으로 충분하나, 보통의 경우와 같이 물체의 표면의 색깔도 계산을 하려면, 즉 조명 계산을 하려면 각 꼭지점에서의 법선 벡터도 설정을 하여야 한다. OpenGL에서는 꼭지점의 좌표를 설정할 때 그에 대응되는 법선 벡터도 같이 지정을 하게 된다. 법선 벡터는 void glNormal3f(GLfloat nx, GLfloat ny, GLfloat nz); 함수를 사용하여 기술할 수 있다. 아래의 코드는 전형적인 법선 벡터의 설정 예를 보여주고 있는데,

```
glBegin(GL_POLYGON);
  glNormal3d(-0.9927, 0.0000, -0.1201);
  glVertex3d(1.3813, 0.0000, 2.4546);
  glNormal3d(-0.9465, 0.0000, -0.3225);
  glVertex3d(1.4000, 0.0000, 2.4000);
  glNormal3d(-0.9140, 0.2452, -0.3229);
  glVertex3d(1.350741, -0.375926, 2.400000);
  glNormal3d(-0.9588, 0.2572, -0.1201);
  glVertex3d( 1.3327, -0.3709, 2.4546);
glEnd();
```

glNormal3*(*) 함수로 법선 벡터를 설정하면, 이 벡터가 OpenGL 상태 기계의 현재 법선 벡터(current normal vector)로 지정이 된다. 이후 glVertex3*(*) 함수로 꼭지점의 좌표를 기술하면 이 꼭지점에는 현재 법선 벡터로 지정된 벡터 값이 달라붙어 꼭지점과 함께 기하 파이프라인을 통해 흘러가게 된다. 따라서 위의 코드에서는 네

개의 꼭지점에 각각 서로 다른 법선 벡터가 붙게 되며, 만약 다음과 같은 코드를 수행하면,

```
glBegin(GL_POLYGON);
  glNormal3d(-0.9927, 0.0000, -0.1201);
  glVertex3d(1.3813, 0.0000, 2.4546);
  glVertex3d(1.4000, 0.0000, 2.4000);
  glVertex3d(1.350741, -0.375926, 2.400000);
  glVertex3d( 1.3327, -0.3709, 2.4546);
glEnd();
```

네 개의 꼭지점 모두에 같은 벡터가 연관되게 된다. 이는 해당 다각형 전체에 대하여 동일한 법선 벡터를 지정하는 예인데, 뒤에서 설명할 상수 쉐이딩에 적합한 설정이라 할 수가 있다.

법선 벡터는 3장에서 설명할 조명 계산을 수행할 때 매우 중요한 역할을 한다. 특히 라이팅 계산에 있어 길이가 1인 단위 법선 벡터(unit normal vector)가 사용되므로, 일반적으로 기하 물체를 모델링 하는 단계에서 정규화된 단위 벡터를 설정한다. 만약 주어진 기하 모델의 법선 벡터가 단위 벡터가 아닌 경우, 사용자가 전처리를 통하여 벡터들의 길이를 1로 만들어 사용하는 것이 좋다.

OpenGL 렌더링 파이프라인에서 조명 계산은 눈 좌표계에서 일어난다. glNormal3*(*) 함수를 사용하여 법선 벡터를 기술하면, 이는 물체 좌표계에서의 벡터로 취급이 되고, 따라서 그 순간의 현재 모델뷰 행렬 $M_{MV}$에 의해 눈 좌표계로 변환이 된다. 문제는 이 과정에서 벡터의 크기가 변할 수도 있다는 사실인데, 그러한 경우에는 눈 좌표계에서의 올바른 사용을 위해서 다시 정규화를 해주어야 한다. OpenGL에서는 디폴트로 벡터 변환 후 어떠한 처리도 해주지 않는다. 만약 정규화를 요청하려면 glEnable(GL_NORMALIZE);와 같은 문장을 수행시켜 명시적으로 벡터 변환 후 정규화 계산이 일어나도록 해주어야 한다. 이러한 상황에서는 변환된 각 벡터

$(n'_x\ n'_y\ n'_z)^t$에 대하여 길이 $d = \sqrt{n'^2_x + n'^2_y + n'^2_z}$를 계산하여 각 좌표 값을 $d$로 나누어 주어야 하기 때문에, 적지 않은 부동 소수점 연산을 동반하고, 따라서 꼭 필요한 경우가 아니라면 피해야 한다.

만약 기하 물체에 대하여 단위 법선 벡터를 사용하고, 또한 현재 사용하는 변환이 벡터의 길이를 항상 일정하게 변화시킨다면, 좀 더 효율적으로 벡터의 길이를 정규화 해줄 수가 있다. glEnable(GL_RESCALE_NORMAL);과 같은 문장을 수행하면, 현재 변환 행렬에 따른 크기 변화 값을 한 번 구해 놓고, 그 이후 변환된 벡터의 각 좌표를 그 값으로 나누어 주므로, 위의 경우와 같이 일반적인 방법으로 정규화 하는 것보다 효율적이다. 예를 들어, 이동 변환과 회전 변환과 함께 $S(\alpha, \alpha, \alpha)$ $(\alpha \neq 0)$과 같은 등방성 크기 변환만을 사용하는 경우에 이러한 방식으로 정규화를 할 수 있다. 이 경우 OpenGL에서는 변환 행렬의 부행렬 $M_{33}$의 세 번째 행의 원소 $m_{31}$, $m_{31}$, $m_{33}$에 대하여 $e = \sqrt{m^2_{31} + m^2_{32} + m^2_{33}}$를 계산하여 변환 벡터를 $e$로 나누어 준다. 이와 같은 방식을 사용하면 각 벡터에 대하여 $d$를 매번 계산할 필요가 없이, 변환 행렬에 대하여 $e$를 한 번만 계산하기 때문에 계산량이 줄어들어 프로그램의 효율을 높일 수가 있다. 앞에서도 언급한 바와 같이 $M_{33}$이 직교 행렬이면 벡터의 크기를 보존해주므로 단위 법선 벡터를 사용한다면 굳이 벡터의 크기를 다시 정규화 할 필요가 없다.

## 3.17 평면의 기하 변환

### 3.17.1 평면 변환의 유도

OpenGL에서는 위치를 나타내는 점과 방향을 가리키는 법선 벡터 외에도 기하 속성을 표현하기 위하여 3차원 공간에서의 평면을 사용한다. 그러한 대표적인 예로 사용자 설정 절단 평면(client-defined clipping plane)을 들 수가 있는데, OpenGL에

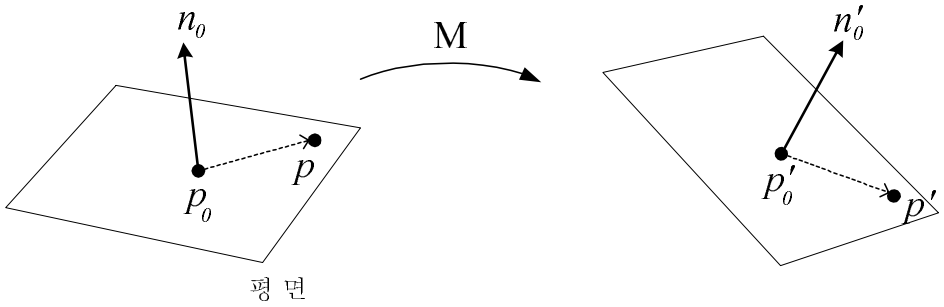



그림 2.41: 평면의 변환

서는 뷰잉 볼륨에 의하여 정의되는 여섯 개의 절단 평면 외에 사용자가 추가적으로 임의의 절단 평면을 정의함으로써 좀 더 다양한 절단 연산을 가능케 해준다. 또한 텍스춰 좌표의 자동 생성을 위해서 그에 기준이 되는 평면을 사용한다. 잘 알다시피 3차원 공간의 한 평면은 $a \cdot x + b \cdot y + c \cdot z + d = 0$과 같이 네 개의 상수 $(a\ b\ c\ d)$에 의하여 정의된다. 만약 주어진 기하 변환 $M$에 의해 좌표 변환이 이루어진다면 이러한 평면은 어떻게 변환이 될까? 답부터 이야기를 하면 변환 후의 새로운 평면에 대한 네 개의 상수 $(a'\ b'\ c'\ d')$는 다음과 같이 간단히 구할 수가 있다.

$$(a'\ b'\ c'\ d') = (a\ b\ c\ d) \cdot M^{-1} \tag{2.12}$$

언뜻 보면 그 이유가 잘 이해가 되지 않지만, 이 역시 조금만 생각을 해보면 어렵지 않게 그 사실을 증명할 수가 있다. 그림 2.41을 보자. 우리가 잘 알다시피 임의의 평면은 한 점 $p_0 = (x_0\ y_0\ z_0)^t$와 벡터 $n_0 = (n_{0x}\ n_{0y}\ n_{0z})^t$에 의하여 정의할 수가 있는데, 이러한 평면은 $n_0^t \cdot (p - p_0) = 0$의 조건을 만족시키는 모든 점 $p = (x\ y\ z)^t$로 구성이 된다[11]. 이 평면에 대한 식은 $n_0^t \cdot p + (-n_0^t \cdot p_0) = 0$과 같이 표현할 수 있는데, 이 때 $n_0^t$의 세 원소가 앞에서 정의한 평면의 첫 세 개의 계수 $a$, $b$, $c$가 되고,

[11] 3.16.1절에서 동차 좌표계를 사용한 것과는 달리 여기서는 편의상 유클리드 공간에서의 좌표를 사용하자.

$-n_0^t \cdot p_0$가 $d$가 됨을 알 수가 있다.

설명의 편의상 아핀 변환 $M$에 대해서만 고려를 하자. 또한 $M_{33}$을 앞에서와 같이 $M$의 왼쪽 위 부분의 3행 3열 부행렬이라 하고, $v_t$를 $M$에서의 이동 변환에 관한 마지막 열의 첫 세 원소로 구성된 열 벡터라 하자. $p_0'$, $n_0'$, $p'$을 각각 $p_0$, $n_0$, $p$가 $M$에 의해 변환된 값이라 하면, 앞의 두 값은 다음과 같이 표현할 수가 있다.

$$\begin{aligned} p_0' &= M_{33} \cdot p_0 + v_t \\ n_0' &= (M_{33}^{-1})^t \cdot n \end{aligned}$$

변환 후의 평면을 $a' \cdot x + b' \cdot y + c' \cdot z + d' = 0$이라 하면 앞에서와 같은 이유로 $(a'\ b'\ c') = n_0'^t$와 $d' = -n_0'^t \cdot p_0'$이 되고, 따라서 다음과 같은 관계를 얻게 된다.

$$\begin{aligned} (a'\ b'\ c' \quad d') &= (n_0'^t \quad -n_0'^t \cdot p_0') \\ &= (n_0^t \cdot M_{33}^{-1} \quad -n_0^t \cdot M_{33}^{-1} \cdot (M_{33} \cdot p_0 + v_t)) \\ &= (n_0^t \cdot M_{33}^{-1} \quad -n_0^t \cdot p_0 - n_0^t \cdot M_{33}^{-1} \cdot v_t) \\ &= (n_0^t \quad -n_0^t \cdot p_0) \cdot \begin{bmatrix} M_{33}^{-1} & -M_{33}^{-1} \cdot v_t \\ 0\ 0\ 0 & 1 \end{bmatrix} \\ &= (a\ b\ c \quad d) \cdot \begin{bmatrix} M_{33}^{-1} & -M_{33}^{-1} \cdot v_t \\ 0\ 0\ 0 & 1 \end{bmatrix} \end{aligned}$$

$M$이 아핀 변환일 때 오른쪽에 곱해지는 행렬은 바로 $M$의 역행렬이 됨을 쉽게 보일 수가 있고, 따라서 식 (2.12)의 변환이 성립하게 된다. □

### 3.17.2 사용자 설정 절단

앞에서 OpenGL에서는 뷰잉 볼륨 외에도 사용자가 임의로 절단 평면을 설정할 수 있다고 하였는데, 이에 대하여 간단히 살펴보자. $a \cdot x + b \cdot y + c \cdot z + d = 0$과 같이 표현된 한 평면은 3차원 공간을 두 개의 영역, 즉 안쪽과 바깥쪽으로 나눈다. 평면으로 절단을 한다는 것은 이 중 바깥쪽에 해당하는 공간을 제거한다는 것을 의미한다. 그러면 주어진 평면에 대하여 과연 어떤 공간이 바깥쪽으로 취급되어 잘려 나가게 될까? 우리가 잘 알다시피 벡터 $(a\ b\ c)^t$는 이 평면에 수직인 한 방향을 나타내는데, OpenGL에서는 바로 이 쪽 방향에 존재하는 공간이 안쪽으로 정의되어 살아남고 다른 쪽 공간은 잘려나가게 된다. 일반적으로 평면 상의 점도 안쪽으로 취급이 되므로, $a \cdot x + b \cdot y + c \cdot z + d < 0$인 점 $(x\ y\ z)^t$들로 이루어진 공간이 잘려나가게 된다.

절단 평면은 glClipPlane(GLenum plane, const GLdouble *equation); 함수에 의하여 설정된다. 여기서 plane은 현재 설정하려는 평면의 이름으로서 GL_CLIP_PLANEi와 같은 형태를 가지는 상수 값을 취한다[12]. equation은 네 개의 계수 $a$, $b$, $c$, $d$가 저장된 영역을 가리키는 포인터 변수이다. OpenGL에서 절단 평면을 설정하면 이는 다른 꼭지점과 법선 벡터와 같이 물체 좌표계에서의 평면으로 취급이 되어, 현재 모델뷰 행렬 $M_{MV}$에 의해 앞 절에서 설명한 방식으로 눈 좌표계로 변환이 된다. 뷰잉 볼륨에 대한 절단 계산이 절단 좌표계에서 수행이 되는 것과는 달리 사용자가 설정한 절단 평면에 대한 절단은 눈 좌표계에서 일어난다. 따라서 위의 함수를 사용하여 평면을 설정하면 $(a\ b\ c\ d) \cdot M_{MV}^{-1} \cdot (x_e\ y_e\ z_e\ w_e)^t \geq 0$을 만족시키는 눈 좌표계의 점 $(x_e\ y_e\ z_e\ w_e)^t$ 들이 이 절단 평면에 대하여 안쪽의 공간을 형성하게 된다.

[12] 이 때 i는 GL_CLIP_PLANE0, GL_CLIP_PLANE1, GL_CLIP_PLANE2, $\cdots$ 등과 같이 0부터 시작하는 정수 값을 가진다.

OpenGL 프로그램에서 사용자 정의 절단 평면을 사용하려면 원하는 평면에 대하여 glEnable(GL_CLIP_PLANEi);와 같은 문장을 수행시켜 명시적으로 그러한 기능을 요청해야 한다. 물론 원래의 상태로 돌아가려면 glDisable(GL_CLIP_PLANEi); 문장을 수행시키면 된다. OpenGL 규약에 의하면 최소한 여섯 개의 사용자 설정 절단 평면을 지원해야 하는데, 자세한 프로그래밍 기법은 생략하도록 하겠다.

## 3.18 절단 좌표계와 절단

### 3.18.1 절단 좌표계

이제 3.8절에서 설명한 투영 변환에 대하여 다시 생각을 해보자. OpenGL에서의 투영 변환은 직관적으로 말해서 카메라의 위치와 방향을 결정한 후, 세상을 얼마나 넓게 볼 것인가를 결정하는 것에 해당한다. 프로그래밍 관점에서 본다면 이는 투영 행렬 스택의 탑의 행렬을 원하는 투영 변환의 내용으로 지정하는 것이 된다. 또한 OpenGL의 기하 파이프라인 관점에서 보면 눈 좌표계로 변환된 기하 프리미티브의 꼭지점들이 투영 행렬에 곱해져 절단 좌표계로 간 후, 원근 나눗셈 연산을 통하여 정규 디바이스 좌표계로 옮겨가는 과정에 해당한다. 이러한 투영 계산 과정에서 각 프리미티브들이 OpenGL의 투영 함수를 통하여 설정된 뷰잉 볼륨에 대하여 절단 좌표계에서 절단이 일어난다고 하였는데, 여기서는 뷰잉 볼륨에 대한 절단 계산에 대하여 좀 더 자세히 살펴보기로 하자.

앞에서도 설명한 바와 같이 눈 좌표계에서 설정된 뷰잉 볼륨이 뷰 매핑 과정을 통하여 정규 디바이스 좌표계 공간의 각 변의 길이가 2인 정육면체로 매핑이 된다. 따라서 절단 연산 이후에도 제거되지 않고 살아남는 각 프리미티브들의 영역은 다

음과 같다.

$$-1 \le x_{nd} \le 1, \quad -1 \le y_{nd} \le 1, \quad -1 \le z_{nd} \le 1$$

절단 좌표계의 점 $(x_c\ y_c\ z_c\ w_c)^t$와 정규 디바이스 좌표계의 점 $(x_{nd}\ y_{nd}\ z_{nd}\ 1)^t$ 간에는 $x_{nd} = \frac{x_c}{w_c}$, $y_{nd} = \frac{y_c}{w_c}$, $z_{nd} = \frac{z_c}{w_c}$와 같은 관계가 있으므로, 정규 디바이스 좌표계의 정육면체 영역에 대응되는 절단 좌표계 공간의 영역은 다음과 같다.

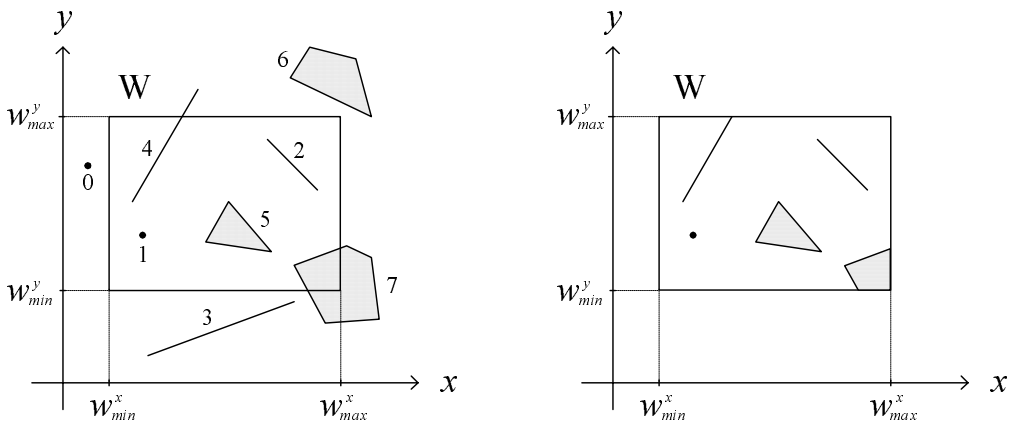
$$w_c > 0: \quad -w_c \le x_c \le w_c, \quad -w_c \le y_c \le w_c, \quad -w_c \le z_c \le w_c \tag{2.13}$$

$$w_c < 0: \quad w_c \le x_c \le -w_c, \quad w_c \le y_c \le -w_c, \quad w_c \le z_c \le -w_c \tag{2.14}$$

OpenGL 기하 파이프라인에서는 기하 프리미티브들을 동차 좌표를 사용하는 절단 좌표계로 변환하여 식 (2.13)과 (2.14)가 나타내는 절단 영역에 대하여 절단을 한 후 원근 나눗셈을 통하여 유클리드 공간인 정규 디바이스 좌표계로 보내준다. 절단 좌표계에서 절단 계산을 함으로써 결국 잘려 나갈 프리미티브들을 구성하는 각 꼭지점에 대한 원근 나눗셈, 즉 상대적으로 비용이 높은 부동 소수점 나눗셈을 피할 수가 있다. 문제는 기하 물체를 구성하는 프리미티브들을 각 타입에 따라 어떻게 절단을 할 것인가 하는 것인데, 이를 쉽게 이해하기 위하여 우선 2차원 공간에서의 고전적인 절단 방법에 대하여 살펴보자.

### 3.18.2 2차원 절단 알고리즘

2차원 절단(two-dimensional clipping)은 2차원 그래픽스 작업에 있어 가장 필수적인 연산중의 하나이다. 기하 프리미티브의 절단을 위해서는 점의 분류(point classification)나 두 직선의 교점의 계산 등의 기본 연산이 반복적으로 수행되어야하기




(a) 절단 전 (b) 절단 후

그림 2.42: 점, 선분, 다각형에 대한 절단 예

때문에, 특히 CPU의 속도가 상대적으로 느렸거나 그래픽스 가속기의 사용이 여의치 않았던 과거에는 절단 연산을 효율적으로 수행하는 것이 상당히 중요한 문제 중의 하나였다. 과거에 비하여 하드웨어의 성능이 월등한 요즘에도 처리하려는 데이터의 양이 워낙 많이 증가를 하였기 때문에, 효율적인 절단 연산의 구현은 지금도 상당히 중요한 문제라 할 수 있다. 2차원 평면에서 절단의 대상이 되는 영역을 절단 윈도우(clip window)라 하는데[13], 설명의 편의상 각 변이 좌표축에 평행한 직사각형 모양의 절단 윈도우에 대한 점, 선분, 그리고 다각형의 절단 방법에 대하여 생각을 하도록 하겠다.

그림 2.42는 네 개의 경계 값에 의해 정의되는 절단 윈도우 W = $(w^x_{min}, w^x_{max}, w^y_{min}, w^y_{max})$에 대한 절단의 예를 보여주고 있다. 2차원 평면 상의 한 직선은 그 평면을 두 개의 공간으로 나눈다. 윈도우 W에 대하여 각 변을 양 방향으로 쭉 뻗은 네 개의 직선 L : $x = w^x_{min}$, R : $x = w^x_{max}$, B : $y = w^y_{min}$, T : $y = w^y_{max}$을 생각할 수 있다. 각 직선에 대하여 $x \geq w^x_{min}$, $x \leq w^x_{max}$, $y \geq w^y_{min}$,

[13] 문맥에 따라 그냥 윈도우라 하기도 하겠다.

그리고 $y \leq w^y_{max}$에 해당하는 영역을 각각 해당 직선에 대한 안쪽 영역, 그리고 그 반대쪽을 바깥 영역으로 정의하면, W는 위의 네 개의 직선에 의해 정의되는 안쪽 영역의 교집합이라 생각할 수 있다.

**점에 대한 절단 알고리즘**

임의의 점 $\mathbf{p} : p_0 = (x_0, y_0)$에 대한 절단은 간단하다. 점은 윈도우를 구성하는 네 개의 직선 모두에 대하여 안쪽에 있을 경우에만 윈도우 W의 안쪽에 있게 된다. 따라서 아래의 조건을 모두 만족을 시킬 때만(그림 2.42의 1번 점) 윈도우의 안쪽에 위치하여 살아남게 되고, 그렇지 않은 경우에는(0번 점) 바깥쪽에 위치하기 때문에 잘려 나가게 된다.

$$x_0 \geq w^x_{min}, \qquad x_0 \leq w^x_{max}, \qquad y_0 \geq w^y_{min}, \qquad y_0 \leq w^y_{max}$$

**선분에 대한 절단 알고리즘**

임의의 두 점 $p_0$와 $p_1$에 의해 정의되는 선분 $\mathbf{l} : (p_0, p_1) = ((x_0, y_0), (x_1, y_1))$에 대한 절단은 약간 복잡해진다. 그림 2.42에서 볼 수 있는 바와 같이 선분의 절단은 크게 세 가지 경우로 나누어진다. 즉 선분 l의 전체가 절단 윈도우의 안쪽에 있어(그림 2.42의 2번 선분) 전부 살아 남거나, 선분 전체가 바깥쪽에 있어(3번 선분) 전부 잘려 나가거나, 부분적으로 윈도우에 걸쳐 있어(4번 선분) 선분 일부가 잘려 나가는 경우로 구별된다. 개념적으로는 단순하나 이를 실제로 구현할 때에는 가능한 한 필요한 연산의 양을 최소화하는 알고리즘을 사용하는 것이 중요하다. 선분 절단의 고전적인 방법으로 코헨-서덜랜드 알고리즘(Cohen-Sutherland algorithm)을 들 수가 있는데, 지금도 가장 보편적으로 쓰이고 있는 절단 방법의 하나로서 3차원 또는

4차원과 같은 고차원 공간으로의 확장이 용이하다.

이 방법에서는 2차원 평면을 절단 윈도우 W를 정의하는 네 개의 경계 직선 L, R, B, T에 대하여 아홉 개의 영역으로 나누어 각 지역에 대하여 아웃 코드(outcode)라 하는 값을 할당한다. 아웃 코드는 네 개의 비트로 구성되어 있는데, 첫 번째(가장 왼쪽) 비트는 직선 T에 대하여 바깥쪽이면 1, 그리고 안쪽이면 0 값을 가진다. 마찬가지로 나머지 세 비트는 각각 직선 B, R, L에 대한 지역 정보를 가진다. 그림 2.43에는 각 지역에 대한 아웃 코드 값이 도시되어 있는데, 각 경계 직선 상의 점은 내부로 간주를 하기 때문에, 윈도우의 경계 위의 점도 내부로 간주한다.

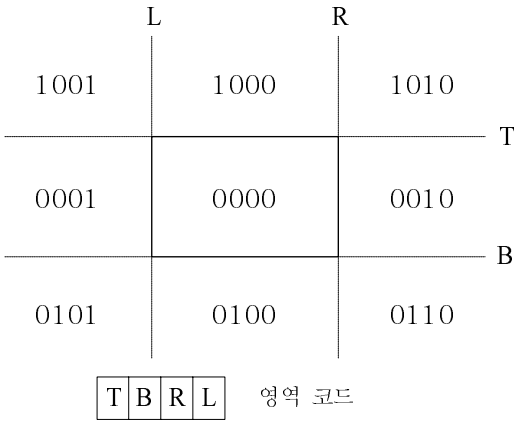



그림 2.43: 영역 코드

주어진 점 $(x, y)$에 대한 영역 코드는 다음과 같은 매크로를 사용하여 무부호 문자(unsigned char) 타입의 변수의 오른쪽 네 개의 비트에 저장을 할 수 있다.

```
#define outcode(x, y) ((x)<wxmin | ((x)>wxmax) << 1     \\
                      ((y) < wymin) << 2 | ((y) > wymax) << 3)
```

선분 l의 양 끝점 $p_0$와 $p_1$의 아웃 코드를 계산하여 각각 무부호 문자 타입의 변수 $c_0$와 $c_1$에 저장을 하였다면, 그 값들에 대하여 다음과 같이 세 가지 경우를 고려할 수 있다.

1. C/C++ 식 $(c_0 \mid c_1)$이 0인 경우, 즉 두 아웃 코드의 값이 모두 0인 경우에는 양 끝점이 모두 윈도우의 내부에 있다는 것을 의미하고, 따라서 선분 전체가 윈도우의 내부에 있게 된다(경우 1).

2. C/C++ 식 $(c_0 \ \& \ c_1)$이 0이 아닌 값을 가지는 경우, 즉 두 개의 아웃 코드가 최소한 한 개의 경계 직선에 대해 비트 필드가 모두 1 값을 가지는 경우에는 선분의 양 끝점이 해당 직선에 대하여 모두 바깥쪽에 있다는 것을 의미하고, 따라서 선분 전체가 윈도우의 바깥 쪽에 있게 된다(경우 2).

3. 만약 위의 두 경우에 해당하지 않고, $(c_0 \ \& \ c_1)$가 0 값을 가지는 경우 이는 선분의 두 끝점이 최소한 한 개의 경계 직선에 대하여 서로 다른 쪽에 놓여 있음을 의미한다(경우 3).

선분이 주어졌을 때 경우 1에 대해서는 단순히 선분 전체가 살아남고, 경우 2에 대해서는 선분 전체가 잘려나가기 때문에 간단하게 절단 계산을 끝낼 수가 있다.

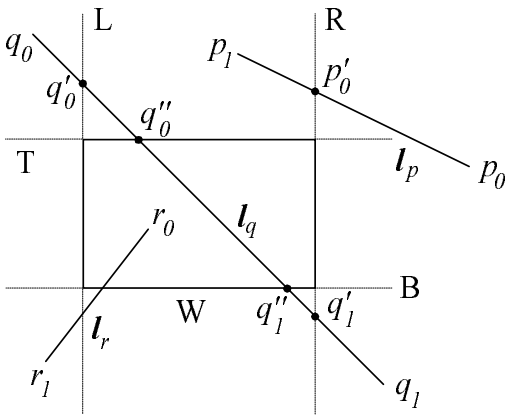



그림 2.44: 선분 절단의 예

문제는 이 두 경우에서와 같이 결정이 나지 않는 경우 3에 해당하는 상황이 발생하였을 때이다. 이는 두 가지 경우로 분류할 수가 있는데, 하나는 선분 전체가 윈도우 바깥에 있는 경우이고, 또 다른 경우는 선분의 일부가 윈도우 안에 걸쳐 있는 경우이다. 그림 2.44에 도시된 세 개의 직선 모두가 경우 3에 해당하는 예이다. 직관적으로 보면 선분 전체가 잘려나갈지, 아니면 일부가 살아 남을 지를 간단하게 판단할 수 있으나, 프로그래밍을 통하여 계산을 하려면 결국 네 개의 직선 L, R, B, T에 대하여 한 번씩 기계적으로 절단을 해주어야 한다. 이 알고리즘의 구현에 있어 중요한 사항은 이 네 개의 직선 각각에 대하여 연달아 절단을 하는 것이 아니라는 점이다. 한 직선에 대하여 일단 절단을 하

1. For W and l : $(p_0, p_1)$, compute outcodes $c_0$ and $c_1$.
2. *while*(1) {
   a. *if* (!($c_0$ | $c_1$)) accept l.
   b. *if* ($c_0$ & $c_1$) reject l.
   c. Make $p_0$ outside.
   d. *if* ($c_0$(L) == 1) clip against L
      *else if* ($c_0$(R) == 1) clip against R
        *else if* ($c_0$($B$) == 1) clip against B
          *else* clip against T.
   e. Update $p_0$ and $c_0$.
  }

그림 2.45: 코헨-서덜랜드 알고리즘

면, 그 결과로 생성되는 선분을 곧바로 다음 직선에 대하여 절단을 하는 것이 아니라, 새로운 선분의 양 끝점에 대하여 아웃 코드를 다시 구한 후, 혹시 이 선분 전체가 윈도우 안에 있는지(경우 1), 아니면 윈도우 밖에 있는지(경우 2)를 검사하여, 만약 해당 사항이 있으면 거기서 종료를 하고, 그렇지 않을 경우 계속하여 다음 경계 직선에 대한 절단 계산을 한다. 이렇게 조금이라도 빨리 종료를 하게 함으로써 교점 계산에 필요한 계산을 조금이라도 피할 수가 있다.

그림 2.45에는 지금까지 설명한 절단 방법이 요약되어 있다. 여기서 2.a와 2.b는 경우 1과 경우 2에 해당하고, 그 이후는 경우 3에 해당한다. 이 마지막 경우에서는 최소한 한 개의 끝점이 윈도우의 바깥쪽에 위치하게 된다. 2.c에서 만약 $p_0$가 안쪽에 있다면 두 점을 서로 바꾸어 $p_0$가 바깥쪽에 놓이게 해준다. 이 때 이 선분은 최소한 한 개의 경계 직선에 대하여 바깥쪽에 있으므로, 2.d에서 L $\rightarrow$ R $\rightarrow$ B $\rightarrow$ T 순서로 $p_0$가 어떤 직선에 대하여 바깥쪽에 있는지 검사하여 그에 대해 절단을 하게 된다[14]. 다시 한 번 강조하면, 중요한 사실은 이 단계에서 선분을 매 경계 직선에 대하여 절단을 하는 것이 아니라, 일단 한 직선에 대하여 절단을 하였으면 2.e에서 $p_0$와 $c_0$를 새로운 값으로 수정한 후 다시 2.a와 2.b로 가서 절단 연산을 종료할 수 있는

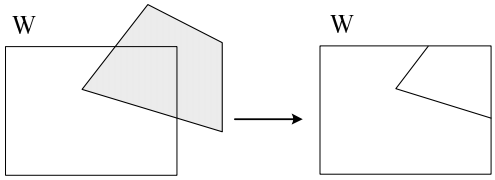
[14] 이 문장에서 $c_0(\cdot)$은 해당 경계 직선에 대응되는 아웃 코드의 비트 값을 나타낸다.

지 검사한다는 점이다.

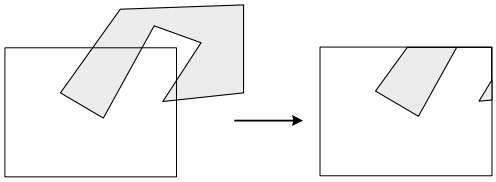
이제 간단한 예를 통하여 이 알고리즘을 이해하여 보자. 그림 2.44의 선분 $\mathbf{l_p}$ : $(p_0, p_1)$을 보면, 초기에 이 선분은 경우 1과 경우 2에 해당하지 않으므로, 알고리즘의 2.c에 이르게 된다. $p_0$가 윈도우의 바깥쪽에 있고, 특히 경계 직선 R의 바깥쪽에 있으므로 이 직선에 대하여 절단이 된 새로운 선분 $(p_0', p_1)$을 얻는다. 이 선분은 다음 회전 단계의 2.b에서 윈도우 바깥쪽에 있는 것으로 판별이 나서 선분 전체가 잘려나가게 된다. 두 번째 직선 $\mathbf{l_q}$ : $(q_0, q_1)$을 절단하기 위해서는 좀 더 많은 계산을 필요로 한다. 이 경우에는 코헨-서덜랜드 알고리즘의 회전문이 네 번을 돌면서 매번 경계 직선에 대하여 절단이 일어나게 된다. 이 알고리즘을 따라가면 매 단계의 절단 결과 $q_0'$, $q_0''$, $q_1'$, $q_1''$ 순서로 새로운 교차 점이 생성되는 것을 알 수 있는데, 자세한 것은 연습 문제로 풀어보기 바란다.

이 예에서 선분 $\mathbf{l_p}$는 한 번의 교점 계산으로 족했으나, $\mathbf{l_q}$는 네 번의 교점 계산 과정을 반복하여야만 절단을 할 수 있는데, 후자의 경우가 최악의 상황이라 할 수 있다. 일반적인 상황에서는 대부분의 경우 처음에 경우 1이나 경우 2에서 빠져나가거나 한두 번의 교점 계산, 즉 한두 직선에 대한 절단으로 선분을 절단할 수 있으므로 평균 수행 시간은 $\mathbf{l_q}$의 경우보다 작다고 할 수 있다.

요약을 하면 코헨-서덜랜드 방법은 결국 네 개의 경계 직선으로 차례대로 절단을 하면서 그 중간에 절단 계산을 종료할 수 있는지를 검사하여 가능한 한 빠르게 계산을 끝내는 방법이라 할 수 있다. 모든 절단 방법이 그러하듯이 한 점이 주어진 경계 직선에 대하여 안쪽에 있는지를 판단하는 점 분류 연산과 두 직선의 교점을 구하는 연산이 이 알고리즘에 있어 가장 기본이 되는 연산임을 알 수가 있다. 코헨-서덜랜드 알고리즘은 가장 최적의 선분 절단 방법은 아니나, 구현하기가 쉽고 또한 고차원으로 용이하게 확장이 되어 많이 쓰이고 있다.




(a) 볼록 다각형



(b) 오목 다각형

그림 2.46: 잘못된 다각형의 절단

**다각형에 대한 절단 알고리즘**

마지막으로 다각형에 대한 절단 알고리즘에 대하여 알아보자. $n(\geq 3)$개의 꼭지점으로 구성된 다각형 **pg** : $(p_0, p_1, p_2, \cdots, p_{n-1})$ $(p_i = (x_i, y_i), i = 0, 1, \cdots, n-1)$을 임의의 절단 윈도우 W $= (w^x_{min}, w^x_{max}, w^y_{min}, w^y_{max})$에 대하여 절단을 할 때도 그림 2.42에서와 같이 다각형 전체가 윈도우 안쪽에 있거나(5번 다각형), 바깥에 있거나(6번 다각형), 아니면 일부만이 윈도우에 걸쳐 있는 등(7번 다각형)의 세 가지 경우로 나누어진다. 다각형은 선분으로 이루어져 있기 때문에 각각의 선분을 앞에서 설명한 코헨-서덜랜드 알고리즘을 사용하여 절단을 한 후 그 결과로 생성된 선분들을 모아 하나의 다각형으로 구성하면 될 듯하나, 실제로 이러한 방법을 사용할 경우 종종 문제가 발생한다. 다각형은 단순히 선분들의 집합이 아니라 공간을 두 부분으로 나누는 닫힌 도형이기 때문에, 절단 후에도 그러한 성질을 보존해 주어야 한다. 그림 2.46은 절단 후에 닫힌 도형으로서의 다각형의 성질이 깨지는 두 가지 예를 보여주고 있다.

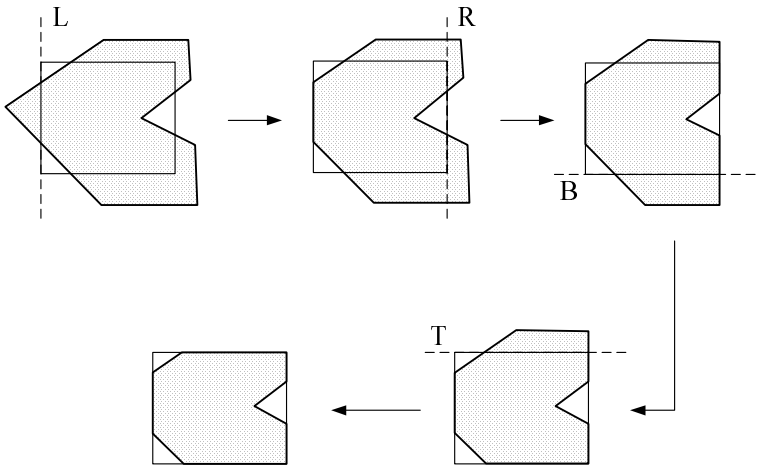
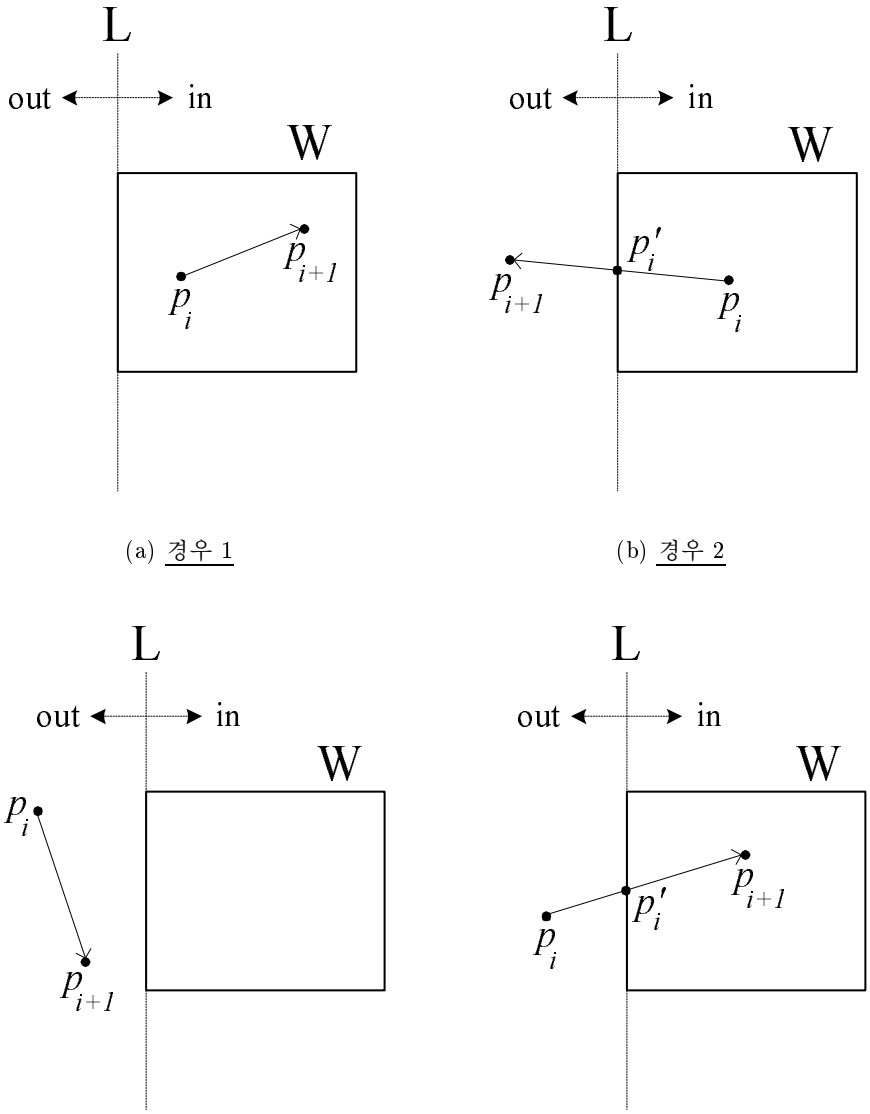



그림 2.47: 각 경계 직선에 대한 절단

다각형의 절단을 위한 고전적인 방법으로 서덜랜드-호지먼 알고리즘(Sutherland-Hodgeman algorithm)을 들 수가 있는데, 이 방법은 코헨-서덜랜드 알고리즘과 같이 절단 윈도우의 각 변을 확장한 네 개의 경계 직선에 대하여 순서대로 다각형을 절단한다. 즉 주어진 다각형을 L에 대하여 절단한 후, 그 결과로 생성된 다각형을 R에 대하여, 그리고 그 결과를 B에 대하여, 그리고 마지막으로 T에 대하여 절단을 함으로써 결과적으로 윈도우에 대하여 절단을 한 다각형을 구하게 된다(그림 2.47).

각 경계 직선에 대한 절단은 단지 고려하는 경계 직선에 따른 점의 분류 연산과 교점 계산의 방법이 다를 뿐 원칙적으로 같은 방법을 사용한다. 서덜랜드-호지먼 방법을 자세하게 이해를 하기 위하여 경계 직선 L을 예를 들어 절단 방법에 대하여 살펴보도록 하자. 기본적으로 한 경계 직선에 대하여 절단을 하려면, 다각형을 이루는 꼭지점을 순서대로 하나씩 처리해야 하는데, 그 결과 새로운 다각형을 생성하게 된다. 지금 꼭지점 $p_i$까지의 처리가 끝난 상태에서 $p_{i+1}$를 처리하려 한다고 가정하자. 이 상황에서 경계 직선에 대한 $p_i$와 $p_{i+1}$의 상대적인 위치에 따라 그림 2.48에서처럼 네 가지의 경우를 고려할 수가 있다.

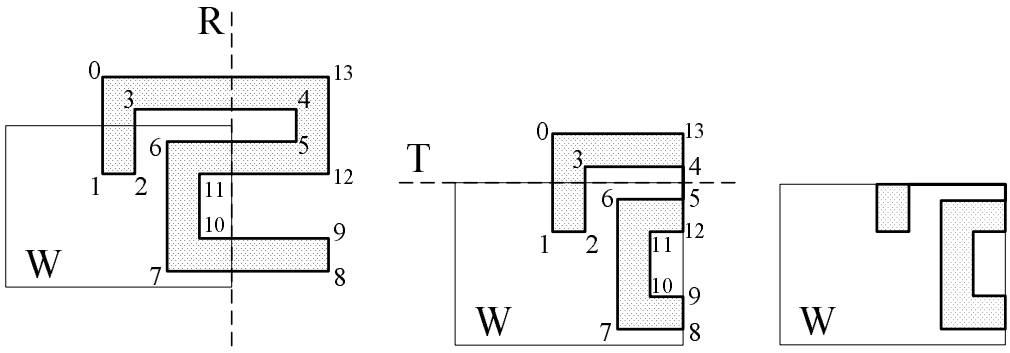
선분의 절단에서와 같이 경계 직선은 2차원 평면을 두 개의 공간, 즉 내부(in)와




(a) 경우 1 (b) 경우 2

(c) 경우 3 (d) 경우 4

그림 2.48: 꼭지점 처리에 대한 네 가지 경우




(a) R 절단 전 (b) R 절단 후 (c) 최종 결과

그림 2.49: 다각형 절단의 예

외부(out)로 구분을 한다. 경우 1은 두 점이 모두 L에 대하여 안쪽에 있는 상황이다. 이 경우에는 선분 전체를 보존해야 하므로 $p_{i+1}$을 새로운 다각형의 꼭지점을 순서대로 저장하는 출력 버퍼에 저장을 한다. 다음 경우 2는 두 점을 있는 선분이 윈도우의 내부에서 외부로 나가는 경우이다. 이 때에는 이 선분을 L로 절단한 점, 즉 이 선분과 L과의 교점 $p_i'$을 구해 이 점을 출력 버퍼에 저장을 한다. 이렇게 함으로써 절단을 하고 남은 선분을 보존할 수가 있다. 세 번째로 경우 3은 두 점이 모두 L의 외부에 있는 경우로서, 해당 선분 전체가 잘려 나가게 되므로, 이 경우는 그냥 넘어가면 된다. 마지막으로 경우 4는 이 선분이 바깥쪽에서 안쪽으로 들어오는 경우인데, 교점 $p_i'$와 $p_{i+1}$을 순서대로 출력 버퍼에 저장을 하면 원하는 부분을 보존할 수가 있다. 이렇게 각 꼭지점을 순서대로 처리를 하면 그 결과 생성된 출력 버퍼의 꼭지점들이 경계 직선 L에 대하여 절단된 새로운 다각형을 형성하게 될 것이다.

그림 2.49(a)은 주어진 다각형을 경계 직선 R에 대하여 절단을 하는 예를 보여주고 있다. 편의상 꼭지점에는 번호만 도시되어 있는데, 위에서 설명한 방식을 사용하여 R에 대하여 절단을 하면 그림 2.49(b)의 다각형을 얻게 된다(꼭지점의 순서를 확인할 것.). 마지막으로 이 다각형을 T에 대하여 절단을 하면 그림 2.49(c)의 다각

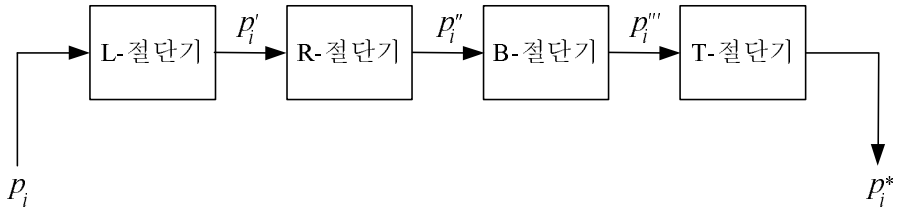



그림 2.50: 다각형 절단 파이프라인

형과 같이 되는데, 이 때 각 꼭지점들을 생성되는 순서대로 번호를 매겨보기 바란다.

선분 절단을 위한 코헨-서덜랜드 알고리즘과 다각형 절단을 위한 서덜랜드-호지먼 알고리즘은 모두 네 개의 경계 직선 각각에 대하여 연속적으로 절단을 하여 원하는 목적을 달성한다. 일견 비슷해 보이나 두 방법 사이에는 약간의 차이가 있다. 코헨-서덜랜드 방법에서는 매번 한 경계 직선에 대하여 절단을 하기 직전에 아웃 코드를 사용하여 종료를 할 수 있는지를 검사하는 반면, 서덜랜드-호지먼 알고리즘에서는 그러한 절차 없이 네 개의 경계 직선에 대하여 연속적으로 절단을 하게 된다. 따라서 서덜랜드-호지먼 방법은 어떻게 보면 불필요한 절단 계산을 수행하기도 하기 때문에 비효율적이라 할 수도 있다. 반면 어느 직선을 사용하는 가를 제외하고는 각 경계 직선에 대하여 절단을 수행하는 방식이 동일하여 코드를 구현하기가 용이하다. 또한 한 경계 직선에 대한 절단 결과로 생성되는 꼭지점들을 버퍼에 저장하지 않고 그대로 다음 경계 직선에 대한 절단 코드로 넘길 수가 있기 때문에, 자연스럽게 하나의 파이프라인으로 연결하여 구현할 수가 있다(그림 2.50). 따라서 서덜랜드-호지먼 알고리즘은 병렬 하드웨어로 용이하게 구현할 수 있다는 장점이 있다.

### 3.18.3 OpenGL 절단 좌표계에서의 절단

지금까지 2차원 유클리드 공간에서 직사각형 형태의 절단 영역에 대한 절단 방법에 대하여 살펴보았는데, 이제 OpenGL의 절단 좌표계에서의 절단에 대하여 알아보도록 하자. 앞에서 살펴본 방법들은 실제로 3차원 투영 공간인 OpenGL의 절단 좌표계 공간으로 자연스럽게 확장을 할 수가 있다. 설명을 짧게 하기 위하여 다각형의 절단에 대해서만 고려하겠다. 3.18.1절에 설명한 바와 같이 절단 좌표계의 점 $p = (x_c\ y_c\ z_c\ w_c)^t$가 사용자가 설정한 뷰잉 볼륨에 대응되는 절단 좌표계의 영역의 내부에 존재하려면 186쪽의 식 (2.13)과 (2.14)의 관계를 만족시켜야 한다. 사실 투영 변환의 과정를 살펴보면 식 (2.13)의 경우만 고려하면 됨을 알 수가 있다. 즉 직교 투영을 한다면 $w_c$는 항상 1이기 때문에 양수 값을 가지고, 원근 변환을 한다면 $w_c = -z_e$이고, 일반적으로 눈 좌표계의 음의 $z_e$축 방향으로 세상을 바라보게 되므로, 이 경우에도 $w_c$는 양수 값을 가진다. 원근 투영 변환을 할 경우 앞 절단 평면을 눈 좌표계의 원점의 뒤쪽에 놓을 수도 있기 때문에 수학적으로 말하면 $w_c < 0$인 경우도 발생하나, 이는 일반적인 상황이 아니므로 OpenGL 규약에서는 $w_c > 0$인 경우에 대해서만 절단을 하는 것을 허용하고 있다. $w_c > 0$일 때의 절단 좌표계 공간에서의 절단 영역을 다시 기술하면 다음과 같다.

$$-w_c \le x_c \le w_c, \quad -w_c \le y_c \le w_c, \quad -w_c \le z_c \le w_c$$

절단 좌표계에서의 절단은 $(x_c\ y_c\ z_c\ w_c)^t$ 형태의 좌표 값을 가지는 4차원 유클리드 공간에서의 절단으로 생각하면 편리하다. 즉 4차원 공간에서 6개의 조건 $x_c \ge -w_c$, $x_c \le w_c$, $y_c \ge -w_c$, $y_c \le w_c$, $z_c \ge -w_c$, $z_c \le w_c$에 의해 결정되는 영역에 대하여 절단을 한다고 생각을 하면 된다. 앞에서 다룬 2차원 절단은 네 개의 조건에 의해

결정되는 직사각형 영역에 대하여 절단을 한 것이었는데, 그 때 각 조건에 대응되는 방향성(안쪽과 바깥쪽)을 가지는 직선을 연속적으로 사용하여 기하 프리미티브들을 절단하였다. 절단 좌표계에서도 마찬가지로 위의 여섯 개의 조건에 의해 결정되는 방향성을 가지는 평면들을 사용하여 절단을 할 수가 있다. 2차원 서덜랜드-호지먼 알고리즘에서 네 개의 경계 직선으로 다각형을 차례대로 자르듯이, '4차원' 서덜랜드-호지먼 알고리즘에서는 여섯 개의 경계 평면을 연속적으로 사용하여 다각형을 절단하면 된다. 서덜랜드-호지먼 방법은 이렇게 고차원으로 자연스럽게 확장이 되며, 실제로 직사각형 영역뿐만 아니라 임의의 개수의 경계 직선(또는 경계 평면)에 의해 정의되는 볼록 영역(convex region)에 대한 절단에 대해서도 쉽게 확장이 된다는 장점이 있다[15].

편의상 위의 여섯 개의 조건에 대응되는 평면에 대하여 각각 `LEFT`, `RIGHT`, `BOTTOM`, `TOP`, `NEAR`, `FAR`와 같이 이름을 붙이자. 이러한 호칭은 자연스럽게 이해할 수 있을 것이다. '4차원' 서덜랜드-호지먼 방법의 기본적인 틀은 2차원의 경우와 동일하게 각 경계 평면으로 차례대로 절단을 하는 것이다. 앞에서도 언급한 바와 같이 절단 알고리즘의 구현에 있어 가장 중요한 기본 연산은 점의 분류와 선분과 경계와의 교점 계산이다. 절단 좌표계에서의 점의 분류, 즉 한 점이 주어진 경계 평면에 대하여 안쪽에 있는지, 아니면 바깥쪽에 있는지는 다음과 같은 코드에 의하여 쉽게 구현할 수가 있다.

```
#define X 0
#define Y 1
#define Z 2
#define W 3
```

[15] 지금 설명하는 절단 좌표계에서의 절단은 여섯 개의 경계 평면에 의해 결정되는 볼록 영역에 대한 것이다.

```
#define NEAR 0
#define FAR 1
#define LEFT 2
#define RIGHT 3
#define TOP 4
#define BOTTOM 5


int inside(float *p, int i, int bndrypl) {
 float w = p[i+W];
 switch(bndrypl) {
   case NEAR:
    return(p[i+Z] >= -w)?  1 :  0;
   case FAR:
    return(p[i+Z] <= w)?  1 :  0;
     ⋮
   case BOTTOM:
    return(p[i+Y] >= -w)?  1 :  0;
 }
 return 0;
}
```

이 예제 코드에서 다각형을 구성하는 꼭지점의 좌표들이 배열에 연달아 저장이 되어 있는데, 현재 처리하려는 꼭지점의 $x_c$, $y_c$, $z_c$, $w_c$ 값들이 `p[i]`부터 시작하여 연속하여 저장이 되어 있다. 내용을 살펴보면 위의 함수 `inside(*)`는 한 꼭지점이 특정 경계 평면 `bndrypl`에 대하여 어느 쪽에 위치하는 지를 결정해주는 코드임을 알 수가 있다.

다음 다각형의 변과 경계 평면의 교점의 계산인데, 두 점 $p_i = (x_c^i\ y_c^i\ z_c^i\ w_c^i)^t$와 $p_j = (x_c^j\ y_c^j\ z_c^j\ w_c^j)^t$에 의해 정의되는 선분과 경계 평면 $z_c = -w_c$(NEAR)와의 교점을 구해보자. 여기서 선분은 $L(t) = (x_c(t)\ y_c(t)\ z_c(t)\ w_c(t))^t = p_i + t \cdot (p_j - p_i)$ $(0 \le t \le 1)$과 같이 매개 변수 $t$를 사용하여 표현할 수가 있다. 선분 상의 점이 경계 평면 상에 있으려면 $z_c(t) = -w_c(t)$의 조건을 만족시켜야 하는데, 이는 $z_c^i + t \cdot (z_c^j - z_c^i) =$

$-(w_c^i + t \cdot (w_c^j - w_c^i))$를 의미하므로, 이 식을 $t$에 대하여 정리하면 교점에 대한 매개변수 값 $t_0$는 다음과 같게 된다.

$$t_0 = \frac{z_c^i + w_c^i}{z_c^i - z_c^j + w_c^i - w_c^j}$$

따라서 이 값을 $L(t)$에 대입하면 교점의 좌표를 구할 수가 있다. 아래에 주어진 코드의 함수 `intersect(*)`는 각각 배열 `p`의 `i`번째와 `j`번째 위치부터 저장이 되어 있는 두 개의 꼭지점에 의해 정의되는 선분과 경계 평면 `bndrypl`과의 교점을 계산하여 그 좌표 값들을 포인터 변수 `v`가 가리키는 장소에 저장을 해주는 함수로서, 쉽게 이해를 할 수가 있을 것이다.

```
void intersect(float *p, int i, int j, int bndrypl,
                  float *v) {
 float t0;


 switch(bndrypl) {
  case NEAR:
   t0 = (p[i+Z]+p[i+W])/(p[i+Z]-p[j+Z]+p[i+W]-p[j+W]);
   break;
  case FAR:
   t0 = (p[i+Z]-p[i+W])/(p[i+Z]-p[j+Z]-p[i+W]+p[j+W]);
   break;
    ⋮
  case BOTTOM:
   t0 = (p[i+Y]+p[i+W])/(p[i+Y]-p[j+Y]+p[i+W]-p[j+W]);
   break;
 }
 v[X] = p[i+X] + (p[j+X] - p[i+X])*t0;
 v[Y] = p[i+Y] + (p[j+Y] - p[i+Y])*t0;
 v[Z] = p[i+Z] + (p[j+Z] - p[f+Z])*t0;
 v[W] = p[i+W] + (p[j+W] - p[i+W])*t0;
}
```

이제 위의 두 함수를 사용하여 절단 좌표계에서의 서덜랜드-호지먼 방법을 아래의 프로그램 예 2.7에서와 같이 구현할 수 있다. 함수 clip_CC_polygon(*)은 꼭지점 버퍼 polyin에 저장된 npolyin개의 꼭지점으로 구성된 다각형을 뷰잉 볼륨에 대하여 절단을 한 후 그 결과 생성된 다각형의 꼭지점 좌표를 polyout에 저장을 하고, 그 다각형의 꼭지점 개수를 리턴을 해주는 일을 하는데, 자세한 설명은 생략을 하도록 하겠다.

**프로그램 예 2.7** 절단 좌표계에서의 다각형의 절단.

```
void copy_polygon(float *pa, float *pb, int nvert) {
 int i;

 for (i = 0; i < nvert*4; i++)
  pb[i] = pa[i];
}
void put_vert(float *v, int i, float *p, int j) {
 p[j*4+X] = v[i+X]; p[j*4+Y] = v[i+Y];
 p[j*4+Z] = v[i+Z]; p[j*4+W] = v[i+W];
}
int clip_CC_polygon(float *polyin, int npolyin,
             float *polyout) {
 int ifrom, ito, bndrypl;
 int npolyout, npolytmp;
 float *polytmp, vertex[4];

 polytmp = (float *) malloc(sizeof(float)*(npolyin+6)*4);

 npolyout = npolyin;
 copy_polygon(polyin, polyout, npolyout);

 for (bndrypl = NEAR; bndrypl <= BOTTOM; bndrypl++) {
  ifrom = (npolyout - 1)*4;
  npolytmp = 0;
```

```
    for (ito = 0; ito < npolyout*4; ito+=4) {
     if (inside(polyout, ito, bndrypl)) {
      if (inside(polyout, ifrom, bndrypl)) { // IN -> IN
       put_vert(polyout, ito, polytmp, npolytmp++);
      }
      else { // OUT -> IN
       intersect(polyout, ifrom, ito, bndrypl, vertex);
       put_vert(vertex, 0, polytmp, npolytmp++);
       put_vert(polyout, ito, polytmp, npolytmp++);
      }
     }
     else {
      if (inside(polyout, ifrom, bndrypl)) { // OUT -> IN
       intersect(polyout, ifrom, ito, bndrypl, vertex);
       put_vert(vertex, 0, polytmp, npolytmp++);
      }
     }
     ifrom = ito;
    }
    copy_polygon(polytmp, polyout, npolytmp);
    npolyout = npolytmp;
   }
   return npolyout;
}
```

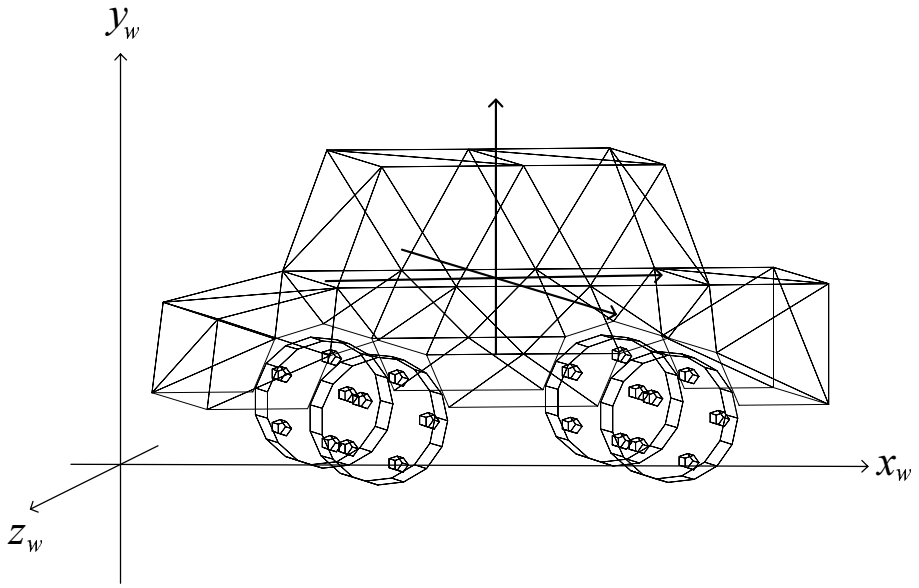



그림 2.51: 세상 좌표계에서의 자동차

# 제 4 절 OpenGL 뷰잉에 대한 프로그래밍 예

## 4.1 계층적 구조를 가지는 자동차

지금까지 3차원 뷰잉에 관련된 컴퓨터 그래픽스 이론과 OpenGL의 뷰잉 모델에 대하여 구체적으로 살펴보았는데, 이제부터 앞에서 배운 내용을 바탕으로 하여 OpenGL 프로그램을 작성하여 보자. 첫 번째 예제 프로그램은 3.15절에서 설명한 모델링 좌표계를 사용한 OpenGL 프로그래밍에 대한 이해를 심화시키기 위한 것으로서, 계층적 구조(hierarchical structure)를 가지는 기하 물체를 효과적으로 다루는 방법에 대하여 알아보도록 하겠다.

그림 2.51에는 세상 좌표계 공간에 자동차 한 대가 배치되어 있는 모습이 도시되어 있다. 이 자동차의 몸체에는 네 개의 동일한 형태의 바퀴가 달려 있고, 각 바퀴에는 다섯 개의 동일한 모양의 나사가 붙어 있다. 이 자동차를 기하학적으로 표현해 주는 방법의 하나는 자동차 전체를 하나의 다면체 모델로 구성하는 것이다. 이 경

우 각 몸체, 바퀴, 나사를 기술해 주는데 필요한 다각형의 개수를 각각 $n_{body}$, $n_{wheel}$, $n_{nut}$이라 하면, 자동차를 위한 모델링 좌표계에서 이 자동차 전체를 나타내기 위해서는 $n_{body} + 4 * n_{wheel} + 20 * n_{nut}$개의 다각형이 필요하다. 이러한 방법은 단순하기는 하나 일반적으로 상당히 비효율적인 방법이라 할 수 있다. 첫째로, 어떤 물체가 여러 개의 동일한 물체를 반복적으로 사용하여 구성이 되어 있을 때, 전체를 하나의 물체로 표현할 경우 메모리가 낭비될 가능성이 많다. 따라서 이러한 경우 3.15절에서도 설명한 바와 같이 서로 다른 형태의 물체들 각각에 대하여 모델을 하나씩 만들고, 이것들을 메모리에 올린 후 적절한 모델링 변환을 통하여 렌더링을 하는 것이 더 효율적인 방법이라 할 수 있다.

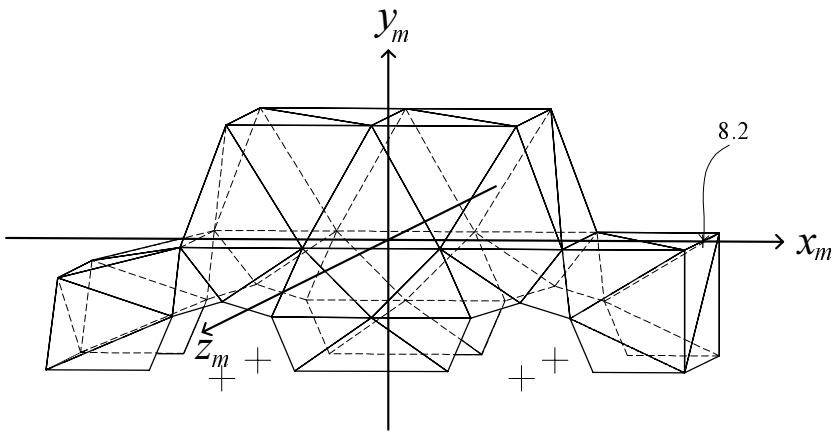
메모리의 비효율적인 사용 외에도 전체 물체를 하나의 좌표계를 사용하여 나타낼 때 발생할 수 있는 더 심각한 문제는 이 물체를 사용하여 애니메이션을 할 경우 매우 복잡한 기하 변환을 해야 될 경우가 발생한다는 점이다. 예를 들어 주어진 궤도를 따라 자동차가 움직이는 애니메이션을 제작한다고 생각해보자. 애니메이션이란 조금씩 변하는 이미지 프레임을 연속적으로 빠르게 그려주는 것이므로, 결국 각 프레임에 대하여 해당 기하 변환을 적용하여 자동차를 세상에 적절히 배치해 주어야 한다. 자동차가 움직임에 따라 자동차의 몸체가 이동을 할 뿐만 아니라, 자동차의 속도에 따라 알맞게 네 개의 바퀴가 바퀴 축을 중심으로 회전을 해야 한다. 또한 각 바퀴에 붙어 있는 20개의 나사도 적절하게 이동 및 회전을 해야 한다. 즉 자동차가 움직임에 따라 각 구성 물체에 대한, 모델링 좌표계로부터 세상 좌표계로의, 적절한 기하 변환을 매번 계산하여 적용을 해야 한다. 따라서 물체가 약간만 복잡한 구성을 가지더라도 이러한 방식의 애니메이션은 프로그래머의 입장에서 볼 때 매우 복잡하고 어려운 작업이 된다.

여기서 자동차를 자세히 관찰을 해보면 이 기하 물체 안에는 구성 요소간에 종속

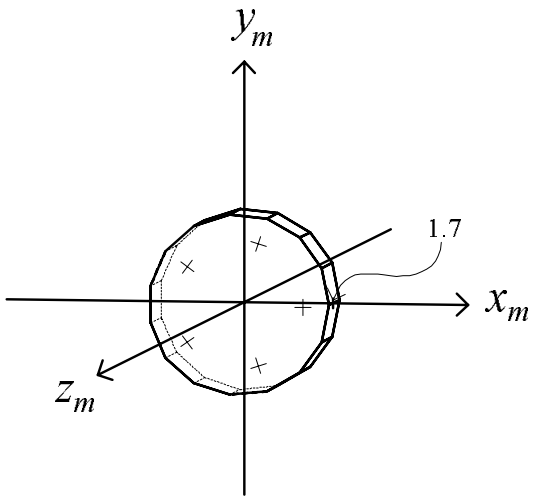
적인 관계, 즉 계층적인(hierarchical) 관계가 존재함을 쉽게 알 수 있다. 각 자동차 바퀴는 몸체에 대하여 상대적으로 그 위치가 결정이 되어 있고, 몸체가 움직일 때 따라 움직이게 된다. 즉 이 경우 자동차 바퀴는 몸체에 종속이 되어 있다고 볼 수 있다. 또한 나사를 생각해보면 나사는 바퀴에 대하여 상대적으로 고정이 되어 있고, 바퀴가 움직일 때마다 나사도 그에 따라 같이 움직이게 된다. 따라서 나사는 몸체에 종속이 되어 있는 바퀴에 종속이 되어 있는 것이다. 다시 말해서 이 자동차 경우 네 개의 바퀴는 각각 몸체에 종속이 되어 있고, 20개의 나사는 각각 다섯 개씩 해당 바퀴에 종속이 되어 있다. 이러한 계층적인 구조를 잘 이용하면 물체를 효과적으로 표현할 수 있을 뿐만 아니라, 위에서와 같은 애니메이션을 손쉽게 할 수가 있다.

이 자동차를 좀 더 조직적으로 표현하기 위하여 우선 세 개의 서로 다른 부속품들이 그림 2.52에서와 같이 자신의 모델링 좌표계를 기준으로 설계되어 있다고 가정하자. 지금 하고자 하는 것은 자동차를 세상에 배치할 때 각 부품에 대한, 경우에 따라 계산하기가 복잡해지곤 하는, 기하 변환을 계산하여 모델링 좌표계에서 직접 세상 좌표계로 보내는 것이 아니라, 위에서 관찰한 부품들간의 종속 관계를 이용하여 구조적으로 손쉽게 모델링 변환을 하려는 것이다. 우선 바퀴와 나사와의 관계를 생각하여 보자. 그림 2.53은 자동차 바퀴가 어떻게 조립이 되는지를 설명해주고 있다. 바퀴 안에는 72도 간격으로 0번에서 4번까지 다섯 개의 나사가 배치되어 있는데, $i$번 나사에 대하여 나사의 모델링 좌표계에서 바퀴의 모델링 좌표계로의 기하 변환은 $M_n^i = R(72 \cdot i, 0, 0, 1) \cdot T(1.2, 0, 1)$과 같이 표현할 수 있다. 즉 이동 변환을 통하여 나사를 자동차 바퀴의 표면으로 옮긴 후 $z_m$축을 둘레로 $72 \cdot i$도 만큼 회전을 하면 된다. 자동차를 세상에 배치할 때 각 나사에 대해서 필요한 변환은 이것이 전부인데, 이는 나사를 직접 세상 좌표계로 옮기는 것보다 훨씬 단순한 변환이다.

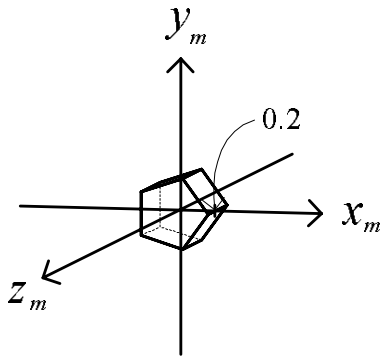
이렇게 나사가 붙은 바퀴를 구성한 후 마찬가지로 바퀴를 세상 좌표계 공간이 아




(a) 자동차 몸체




(b) 자동차 바퀴




(c) 자동차 나사

그림 2.52: 자동차 부품의 설계

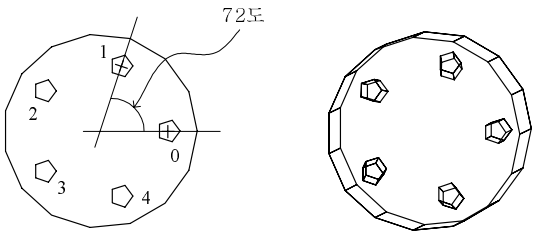



그림 2.53: 바퀴의 조립

니라 몸체의 모델링 좌표계로 보내 자동차를 조립하게 된다. 자동차에는 네 개의 바퀴가 있는데 몸체의 모델링 좌표계를 기준으로 하여 앞쪽에, 즉 $z_m$이 양수인 쪽에 있는 왼쪽과 오른쪽 바퀴를 각각 0번, 1번 바퀴라 하고, 반대편 쪽의 두개의 바퀴를 2번, 3번 바퀴라 하자. $i$번 바퀴를 바퀴의 모델링 좌표계에서 몸체의 모델링 좌표계로 옮겨주는 변환을 $M_w^i$이라 하면 이는 다음과 같이 표현할 수 있다.

$$\begin{aligned} M_w^0 &= T(-3.9, -3.5, 4.5), \\ M_w^1 &= T(3.9, -3.5, 4.5), \\ M_w^2 &= T(-3.9, -3.5, -4.5) \cdot S(1, 1, -1), \\ M_w^3 &= T(3.9, -3.5, -4.5) \cdot S(1, 1, -1) \end{aligned}$$

이러한 부품들간의 관계를 구체적으로 표현하면 그림 2.54에서와 같이 트리(tree) 형태로 나타낼 수 있다. 트리는 컴퓨터학에서 기본이 되는 자료 구조로서 데이터에 내재하는 종속적인, 다시 말해서 계층적인 관계를 표현하는데 널리 쓰인다. 이 트리를 아래에서 위로 가면서 살펴보면, 20개의 나사가 각각 다섯 개씩 해당 바퀴의 모델링 좌표계로 옮겨지고 있음을 알 수가 있다. 다음 바퀴는 자신의 좌표계로 옮겨진 나사들과 함께 다시 적절한 변환에 의하여 몸체의 모델링 좌표계로 옮겨진 후, 마지막으로 자동차를 세상으로 보내주는 변환 $M_b$에 의하여 세상으로 배치가 된다. 그 결과 여러 개의 부품으로 구성된 복합적인 물체인 자동차의 각 구성 요소들이 따로 움직이는 것이 아니라 유기적으로 연결이 되어 구조적으로 움직이게 된다.

이 트리에서 두 개의 노드를 연결하는 변(edge)에는 기하 변환 행렬이 속성으로 붙어 있는데, 이는 자식 노드(아래 쪽 노드)에 해당하는 물체가 어떻게 부모 노드(위쪽 노드)의 공간으로 변환이 되는지를 정의한다. 예를 들어 0번 바퀴의 0번 나사를

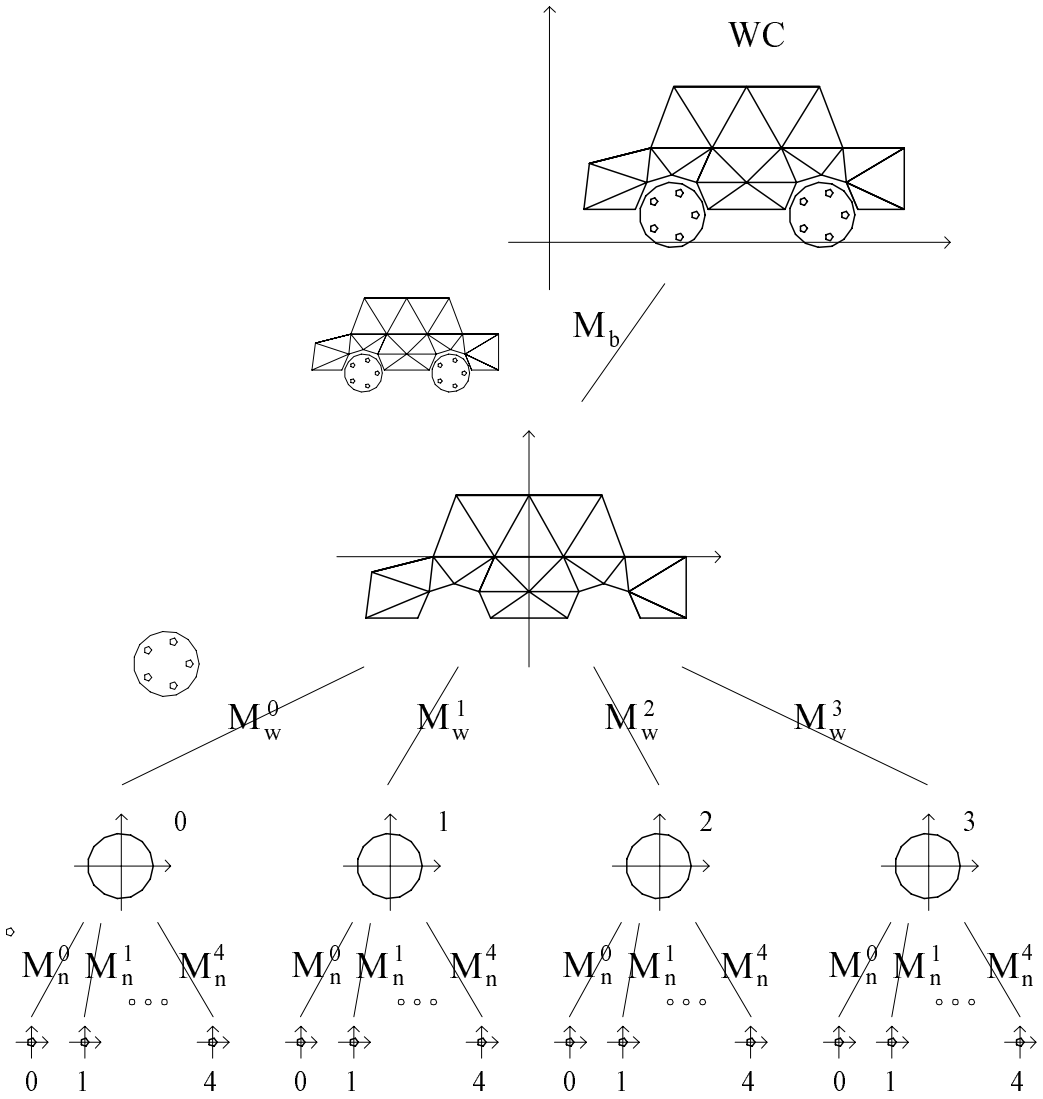



그림 2.54: 자동차의 계층적 구조

생각해보자. 세상에 배치된 자동차를 루트 노드라 할 때, 이 나사에 해당하는 리프에서 루트까지의 경로를 따라가면 $M_n^0$, $M_w^0$, $M_b$ 순서로 기하 변환 행렬과 만나게 된다. 바로 이 순서대로 기하 변환을 하면 나사의 모델링 좌표계에서 기술된 꼭지점들이 몇 단계의 변환을 거쳐 세상으로 배치가 된다. 이렇게 하면 직접 각 물체 자신의 공간에서 세상으로 직접 옮겨가는 변환을 매번 구할 필요가 없이, 자신의 부모 노드에 해당하는 물체의 공간으로만 변환을 하여 자동차를 구성한 후, 비교적 계산하기 수월한 변환 행렬 $M_b$만 매번 구하면 움직이는 자동차를 세상으로 보내는 작업이 훨씬 수월해진다[16].

한편 이 트리의 노드에는 기하 물체에 대한 다면체 모델이 노드의 속성으로 붙어 있다. 물론 구현을 할 때에는 서로 다른 물체는 메모리에 한 번씩만 올라오기 때문에, 다면체 데이터에 대한 포인터가 노드에 붙어 있다고 생각하면 된다[17]. 이렇게 트리의 노드와 변에는 각각의 속성이 붙어 있는데, 한 노드에 연결된 다면체 모델의 꼭지점들에 대하여 그 노드로부터 루트 노드까지 올라가며 만나는 변환 행렬들을 순서대로 곱하면 세상으로 배치가 된다.

전체 자동차를 세상으로 배치하기 위해서는 결국 이 트리를 탐색(search)하면서 각 노드에 대하여 지금 설명한 바와 같은 연산을 반복하면 된다. 트리를 탐색하는 방법으로 크게 깊이 우선 탐색(depth first search, DFS)와 넓이 우선 탐색(breadth first search, BFS) 두 가지를 들 수 있는데, 여기서 계층적인 구조를 효과적으로 이용하기 위해서는 깊이 우선 탐색이 더 적절하다고 할 수 있다. 이 트리를 루트 노드

[16] 설명의 편의상 여기서는 자동차가 움직일 때 바퀴가 회전하지 않는다고 가정한다. 실제로는 매번 자동차가 움직일 때마다 그 속도에 따라 적절히 바퀴가 회전을 하여야 한다. 이러한 효과는 행렬 $M_w^i$를 적절히 조절하면 되는데, 지금 설명하고 있는 물체간의 계층적인 관계와 그에 대한 OpenGL 프로그래밍 방법을 잘 이해한다면 어려움 없이 그러한 것을 구현할 수 있을 것이다.

[17] 나중에 좀 더 복잡한 물체를 구성할 때에는 종종 노드는 기하 물체가 붙어 있지 않는, 가상 노드일 수도 있을 것이다. 이 예에서는 바로 루트 노드가 기하 물체 속성이 붙어 있지 않는 가상 노드이고 나머지 노드들에는 기하 모델이 붙어 있다.

에서 출발하여 깊이 우선 순서로 탐색을 하면서 매 순간 루트 노드에서 현재 방문하고 있는 노드까지의 경로 상에 있는 변환들을 순서대로 저장을 해 놓고, 현재 노드에 붙어 있는 기하 물체에 대하여 적절한 순서로 변환을 하여 배치하면 된다. 이때 깊이 우선 탐색을 하면서 항상 가장 마지막에 내려간 변에 붙은 행렬을 가장 먼저 곱해주어야 한다. 즉 탐색 순서와 행렬의 곱셈의 순서는 반대이므로 여기서 자연스럽게 스택이라는 자료 구조가 사용이 된다. 아직 약간은 추상적으로 이야기하면, 여기서의 탐색은 1. 아래로 내려 갈 때 변에 있는 행렬을 푸쉬하고, 2. 다시 위로 돌아올 때 그 행렬을 팝하며, 또한 3. 현재 노드에 기하 물체가 붙어 있으면 스택에 있는 행렬들을 순서대로 곱하여 배치를 하는 방식으로 자동차에 대한 모델링 변환을 구현할 수 있다.

바로 지금까지 설명한 바와 같은 계층적인 구조를 효율적으로 구현하기 위하여 OpenGL과 같은 그래픽스 라이브러리에서는 스택과 그에 대한 연산을 위한 함수들을 제공한다. 110쪽의 3.3절에 설명한 OpenGL 함수들이 직간접적으로 스택을 다루는데 쓰인다. 예제 프로그램 2.C에는 이 자동차를 렌더링 해주는 프로그램이 주어져 있는데, 프로그램 예 2.8에는 모델링 변환과 관련된 주요 함수들이 설명되어 있다.

**프로그램 예 2.8** 자동차에 대한 계층적인 모델링 변환.

```
void draw_wheel_and_nut(void) {
 int i;

 draw_wheel(); // draw wheel object
 for (i = 0; i < 5; i++) { // for nut i
  glPushMatrix();
   glRotatef(72.0*i, 0.0, 0.0, 1.0);
   glTranslatef(rad-0.5, 0, ww); // rad = 1.7, ww = 1.0
   draw_nut(); // Line (d):  draw nut object
```

```
    glPopMatrix();
  }
}

void draw_car(void) {
  draw_body(); // draw body object
  glPushMatrix(); // for wheel 0
    glTranslatef(-3.9, -3.5, 4.5);
    draw_wheel_and_nut(); // Line (c)
  glPopMatrix();
  glPushMatrix(); // for wheel 1
    glTranslatef(3.9, -3.5, 4.5);
    draw_wheel_and_nut();
  glPopMatrix();
  glPushMatrix(); // for wheel 2
    glTranslatef(-3.9, -3.5, -4.5);
    glScalef(1.0, 1.0, -1.0);
    draw_wheel_and_nut();
  glPopMatrix();
  glPushMatrix(); // for wheel 3
    glTranslatef(3.9, -3.5, -4.5);
    glScalef(1.0, 1.0, -1.0);
    draw_wheel_and_nut();
  glPopMatrix();
}

void draw_world(void) {
  draw_axes();

  glPushMatrix(); // Line (a)
    // from body to world
    glTranslatef(10.0, 5.0, 0.0);
    glRotatef(15.0, 0.0, 1.0, 0.0);
    draw_car(); // Line (b)
  glPopMatrix();
}
```

프로그램을 살펴보면 알 수 있듯이 각 물체들은 한 번씩 메모리에 저장이 되어 있고, 필요할 때마다 `draw_body()`, `draw_wheel()`, `draw_nut()` 함수를 불러 해당하는 기하 물체를 그리고 있다. 이 예제 프로그램에서는 뷰폿 변환과 투영 변환을 하고 모델뷰 행렬 스택에 뷰잉 변환 행렬 $M_V$를 올린 후 `draw_world()` 함수를 부르고 있다. 따라서 이 함수의 수행이 막 시작되었을 때의 모델뷰 행렬 스택의 내용은 그림 2.55(a)와 같다. 이 함수는 그림 2.54에 도시된 트리를 앞에서 설명한 방식으로 탐색을 하면서 자동차에 대하여 적절한 모델링 변환을 통하여 렌더링을 해주는 역할을 한다. 이제 이 함수가 어떻게 트리를 깊이 우선 순서로 쫓아가는지 따라가 보자. 출발은 세상에 배치된 자동차에 해당하는 루트 노드에서부터인데 이 노드에는 기하 물체가 붙어 있지 않으므로 그냥 아래로 내려간다. 위에서 설명한 바와 같이 항상 아래로 내려갈 때에는 변에 붙어 있는 변환 행렬을 스택에 푸쉬를 하여야 한다. 이를 위하여 우선 `draw_world()` 함수의 Line (a)에서 glPushMatrix() 함수를 사용하여 푸쉬를 하는데, 이 함수는 스택의 깊이를 하나 증가시키면서 이전의 탑의 내용을 그대로 복사를 한다. 다음 $M_b$에 해당하는 내용을 스택에 올리게 되는데, OpenGL 기하 변환 함수의 성질로 인하여 이 행렬이 스택의 탑의 오른쪽에 곱해진다. 이러한 방식으로 푸쉬를 하면 항상 모델뷰 행렬 스택의 탑에는 현재 탐색중인 노드에서 루트 노드까지의 경로 상에 있는 행렬들이 우리가 원하는 순서대로 곱해져 저장되어 있게 된다. 또한 glPushMatrix() 함수를 사용하여 푸쉬를 함으로써 이전의 스택의 탑의 내용을 보존 할 수 있고, 추후에 그 내용을 다시 사용할 수 있다. 이 예에서는 특별히 그럴 필요는 없으나 추후 움직이는 자동차를 그리기 위해서는 $M_V$를 보존하는 것이 매 프레임마다 뷰잉 변환을 해주는 OpenGL 함수를 다시 부르는 것보다 더 효율적이다.

`draw_world()` 함수에서는 푸쉬를 한 후 이동 및 회전 변환을 통하여 $M_b$에 해당


(a) Line (a) 수행 직전 (b) Line (b) 수행 직전

(c) Line (c) 수행 직전 (d) Line (d) 수행 직전

그림 2.55: 모델뷰 행렬 스택의 내용

하는 내용을 스택에 올리고 있다. 따라서 `draw_car()` 함수를 호출하기 직전의 모델뷰 행렬 스택의 내용은 그림 2.55(b)와 같고, 자동차를 다 그린 후 `draw_world()` 함수의 마지막 문장 `glPopMatrix();`을 통하여 팝을 하면[18], 모델뷰 행렬 스택에는 이 자동차를 그리기 직전의 내용이 복원되어 있게 된다. 이 예에서는 $M_V$가 다시 스택의 탑에 있게 되는데, 만약 다른 물체를 더 그릴 경우 이 내용이 필요할 것이다.

이제 `draw_car()` 함수를 통하여 트리의 아래 부분을 탐색하는 과정을 간단히 살펴보면, 우선 현재 몸체에 해당하는 노드를 방문하고 있으므로 `draw_body()` 함수를 호출하여 그에 대한 기하 물체를 그리고, 다시 재귀적으로 이 노드의 아래 부분을 탐색하게 된다. 몸체를 그릴 때에는 모델뷰 행렬 스택의 탑에는 $M_V \cdot M_b$가 올라와 있으므로, 자신을 세상으로 배치해주는 모델링 변환 $M_b$가 정확하게 스택에 올라와 있게 된다. 이제 0번 바퀴로 내려가기 위하여 다시 푸쉬를 한 후 $M_w^0$을 스택의 탑에 올리게 된다(`glTranslatef(-3.9, -3.5, 4.5);` 문장). 이 때의 스택의 내용은 그림 2.55(c)와 같고, 이제 0번 바퀴를 그릴 준비가 되었으므로 `draw_wheel_and_nut()`

[18] 이는 트리에 대한 탐색을 마친 후 마지막으로 루트 노드로 돌아오는 것에 해당한다.

함수를 부르게 된다.

이 함수는 우선 `draw_wheel()` 함수를 통하여 바퀴에 해당하는 기하 물체를 그린 후, `for` 루프를 통하여 0번 바퀴의 다섯 개의 자식 노드들을 반복적으로 방문하면서 나사들을 그리게 된다. 0번 바퀴의 $i$번째 나사 노드를 방문하기 위하여 `glRotatef(72.0*i, 0.0, 0.0, 1.0);`과 `glTranslatef(rad-0.5, 0, ww);` 문장을 통하여 $M_n^i$를 스택에 올리고 `draw_nut()` 함수를 불러 나사를 그려주고 있다. 따라서 0번 바퀴에 대한 `for` 루프에서 `draw_nut()` 함수를 부르기 직전에 스택의 탑에는 그림 2.55(d)와 같은 형태의 행렬이 올라가 있을 것이다. 일단 나사를 다 그리면 그러한 행렬이 더 이상 필요가 없으므로 팝을 해주는데 이는 해당 노드(만약 리프 노드가 아니라면 해당 노드 및 그 아래의 내용)에 대한 탐색이 다 끝난 후 다시 부모 노드로 돌아가는 것에 해당한다. 이러한 방식으로 이 함수들은 자동차에 대한 트리를 탐색하면서 적절하게 모델링 변환을 해주는데, 이 함수들을 잘 쫓아가면서 모델뷰 행렬 스택의 내용이 동적으로 어떻게 변하는지 살펴보기 바란다.

이제 위의 내용을 이해했다면 왜 일면 더 복잡해 보이는 이러한 방식으로 프로그래밍을 하는지 알아보자. 0번 바퀴의 0번 나사의 경우 이 나사 입장에서는 단순히 바퀴의 모델링 좌표계의 해당 위치로만 가면 된다. 또한 0번 바퀴의 경우에도 그냥 몸체 좌표계의 해당 위치로 가는 변환만 생각하면 된다. 그러면 각각의 구성 요소들은 어떻게 세상의 원하는 위치로 가게 되는가? 몸체는 자동차를 세상으로 배치하는 변환 $M_b$에 의하여 세상으로 옮겨가고 0번 바퀴는 일단 $M_w^0$에 의해 몸체 좌표계까지만 가면, 몸체가 $M_b$에 의해 세상으로 갈 때 같이 붙어 가게 된다. 마찬가지로 0번 나사는 일단 $M_n^0$에 의해 바퀴 좌표계로 가면 바퀴가 바퀴 좌표계에서 몸체 좌표계를 거쳐 세상 좌표계로 갈 때 같이 쫓아가게 되는 것이다. 따라서 자동차가 움직일 경우 $M_b$만 다시 계산을 하면 되므로 훨씬 프로그래밍이 단순해진다. 또한 만

약 자동차가 움직이면서 바퀴가 회전을 할 경우에도 $M_w^i$ 내용만 적절히 바꾸어 주면 나사 입장에서는 아무 것도 바뀔 것이 없다.

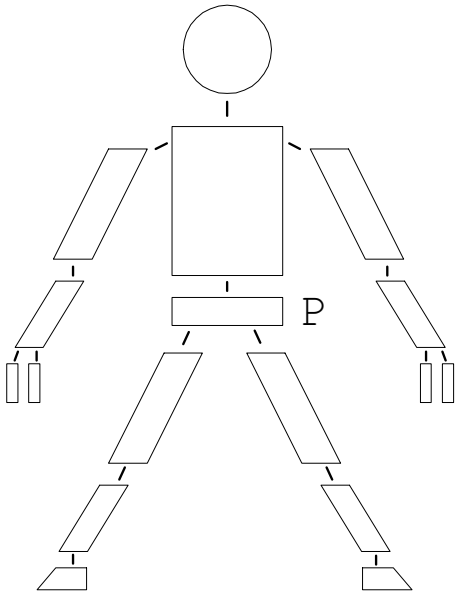



그림 2.56: 계층적 구조를 가지는 로보트

실제로 복잡한 물체들을 모델링할 때 물체를 구성하는 요소들간의 계층 구조를 이용함으로써 렌더링과 애니메이션 작업을 단순화시킬 수가 있다. 그림 2.56과 같은 로보트를 생각해보자. 자동차에서와 같이 이 로보트를 구성하는 요소 물체간에는 서로간의 종속적인 관계가 존재한다. 예를 들어 몸통이 움직이면 거기에 붙어 있는 머리, 팔, 다리 등이 움직이고, 이들에 종속된 손가락, 발 또한 같이 움직이게 된다. 로보트와 같은 물체는 보통 허리나 골반이 구성하는 트리의 루트 노드에 해당하는데, 주의하여야 할 것은 이와 같은 물체에는 트리의 노드간에 종속성뿐만 아니라, 구성요소들간의 제한도 같이 존재한다는 점이다. 예를 들면 팔에 붙은 손가락이 회전할 수 있는 각도에는 제한이 있다. 따라서 자동차보다는 약간은 더 복잡해지나 이러한 정보들을 위에서와 같은 트리에 적절해 저장을 해주면, 주어진 제한 조건을 만족시키면서 로보트가 움직이게 할 수 있다.

다시 한 번 강조하면 데이터에 내재하는 계층성을 표현하기 위하여 컴퓨터학에서의 대표적인 자료 구조인 트리를 사용을 한다. 그렇게 표현된 내용을 탐색하는 방법으로 깊이 우선 탐색과 넓이 우선 탐색이 있다고 하였는데, 전자는 스택이라는 자료 구조를 사용하여 자연스럽게 구현을 할 수 있고, 후자는 큐(queue)라는 자

료 구조와 자연스럽게 매칭이 된다. 스택의 기본 연산은 푸쉬와 팝인데 스택은 어떤 정보를 저장하기 위하여, 그리고 팝은 그 정보가 더 이상 필요가 없어 제거를 하려할 때 사용이 된다. 스택을 사용한 모델링 변환과 개념적으로 밀접한 관계가 있는 것으로 역시 스택이라는 자료 구조를 사용해야 하는 프로그래밍 언어의 블록 구조(block structure)를 들 수가 있다. 우리가 프로그래밍을 할 때 '{'로 새로운 블록을 열면, 이는 하나의 새로운 환경을 설정하는 것을 의미한다. 그러나 이 때 기존 환경의 내용을 제거하는 것이 아니라, 필요할 때마다 다시 사용을 할 수 있도록 보존을 해야 한다. 따라서 '{'는 스택의 푸쉬 연산에 대응이 된다고 할 수 있다. 그리고 새로운 환경에서의 작업이 모두 끝나면 '}'를 사용하여 그 블록을 닫는데, 더 이상 그 환경의 내용을 저장할 필요가 없으므로 제거를 하면 된다. 따라서 '}'는 스택의 팝 연산과 자연스럽게 대응이 된다. 또한 항상 가장 나중에 '{'로 연 블록이 가장 먼저 나오는 '}'에 의하여 끝나게 되므로 구조상 스택과 자연스럽게 연관이 되고, 따라서 이러한 블록 구조를 수행할 때 동적으로 변하는 스택이 중요하게 쓰이는 것이다. 그래픽스 렌더링을 위한 기하 물체의 계층적인 구조, 그에 대한 계층적인 모델링 변환, 트리라는 자료 구조와 그에 대한 깊이 우선 탐색, 스택과 푸쉬 및 팝 연산, OpenGL에서의 glPushMatrix()와 glPopMatrix() 함수, LIFO 구조, 그리고 블록 구조 프로그램의 수행 등이 개념적으로 어떻게 연결이 되는지 잠시 생각해보기 바란다.

마지막으로 다음 예제로 넘어가기 전에 다시 강조를 하고자 하는 것은 OpenGL 프로그래밍을 할 때에는 항상 프로그래머 입장이 아니라 OpenGL 시스템 관점에서 해야 한다는 점이다. `draw_wheel()` 함수에서 회전 및 이동 변환에 대한 문장을 다시 한번 살펴보자. 이 두 문장이 의도하는 바는 $i$번째 나사를 바퀴 안에서의 적절한 위치로 옮기는 것이다. 직관적으로 생각하면 나사를 우선 이동을 한 후, 다음 회전

을 하여야 하기 때문에 `glTranslatef(*);` 문장이 수행된 다음에 `glRotatef(*);` 문장이 수행되어야 할 것처럼 생각할 수 있다. 그러나 여기서 중요한 것은 OpenGL 시스템에서 사용하는 스택의 탑에 적절한 변환 행렬을 설정하는 것이므로, 스택과 OpenGL의 기하 관련 함수의 성질에 따라 나중에 적용되는 변환에 해당하는 OpenGL 문장이 먼저 나와야 하는 것이다. 만약 두 문장의 순서를 바꾼후 렌더링을 하였을 때 어떠한 결과가 나올지 생각하여 보기 바란다.

## 4.2 회전하는 어미소와 송아지

이제 두 번째 예제 프로그램으로서 앞 절에서 익힌 계층적 모델링 기법을 적용하여 간단한 애니메이션을 위한 OpenGL 프로그램을 작성해보자. 그림 2.57에는 예제 프로그램 2.D에 주어진 프로그램을 수행시킬 때의 한 장면이 도시되어 있다. 여기서 어미소(A)가 $y_w$축(수직인 축)을 둘레로 반지름이 5인 원을 따라 회전을 하고 있다. 어미소는, 사실은 동일한 모양의, 송아지(B)를 목에 태우고 회전을 하고 있다. 한편 어미소가 회전을 할 때 그 가슴둘레로 정육면체 하나(C)가 어미소가 회전을 하는 것보다 10배 빠른 각속도로 회전을 하고, 또한 이 정육면체 둘레로, 역시 동일한 모양의, 크기가 약간 작은 정육면체(D)가 어미소보다 40배 빠른 각속도로 회전을 하고 있다.

이러한 물체들이 자연스럽게 움직이는 것처럼 보이게 해주기 위해서는 각 프레임마다 움직이고 있는 물체들의 세상 좌표계에서의 위치와 방향을 적절히 결정해 주어야 한다. 언뜻 보기에 약간은 복잡하게 보이는 이러한 애니메이션은 물체들간의 관계를 분석해보면 별로 어렵지 않게 프로그래밍을 할 수 있다. 우선 어미 소와 송아지와의 관계를 생각을 해보자. 어미소가 움직임에 따라 송아지는 어미소에 대하여 고정이 된 채로 따라 움직이고 있으므로, 송아지는 어미소에 종속이 되어 있

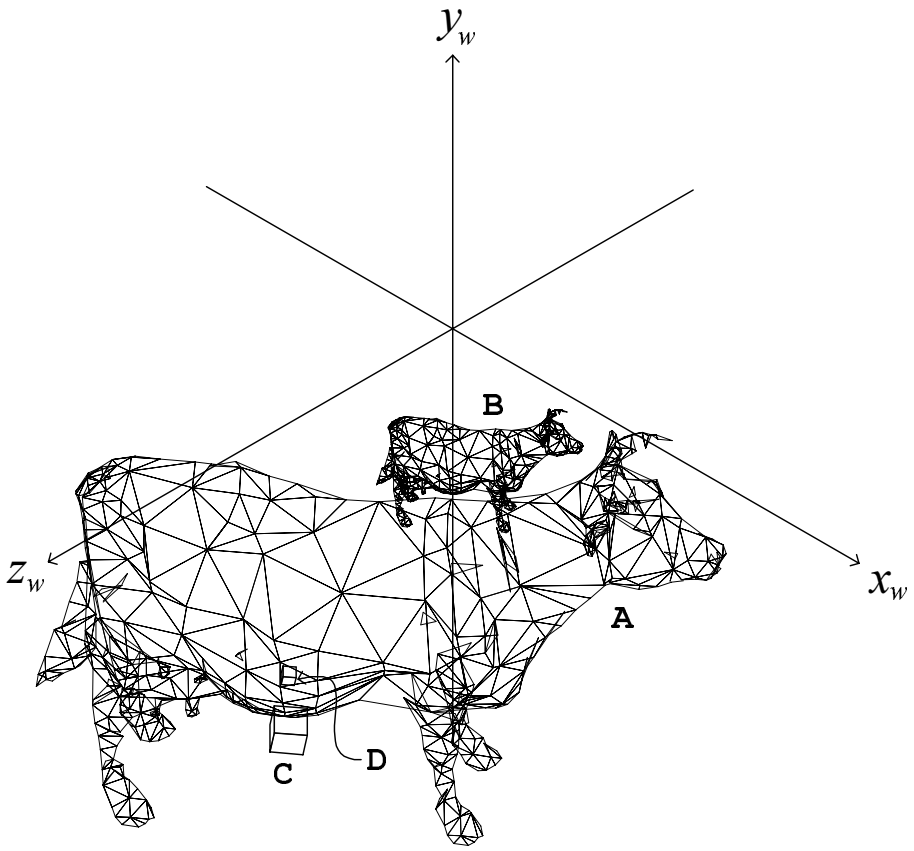



그림 2.57: 세상 좌표계에서의 한 장면

다고 볼 수가 있다. 따라서 송아지에 대한 모델링 좌표계에서 기하 물체를 직접 세상 좌표계로 보내주는 것보다 어미소의 모델링 좌표계로 보내주는 것이 더 간단한 방법이 된다.

다음 큰 정육면체의 경우, 이 물체는 어미소의 주변으로 회전을 하는 동시에 $y_w$축을 따라 회전을 하고 있지만, 후자는 어미소의 회전에 기인을 한다. 따라서 이 정육면체를 어미소에 종속을 시키면 전자의 회전만 고려를 하면 되고, 따라서 큰 정육면체의 회전에 대한 모델링 계산을 단순화시킬 수가 있다. 마지막으로 작은 정육면체는 큰 정육면체의 둘레로 돌고 있기 때문에, 마찬가지로 큰 정육면체에 종속을 시키면 된다. 이렇게 함으로써 큰 정육면체가 회전을 할 때 어미소가 회전을 하는 것에 무관하게 어미소 둘레로 회전을 하면 되고, 작은 정육면체는 어미소가 회전하는 것이나 큰 정육면체가 회전하는 것에 상관없이 큰 정육면체 둘레로만 회전을 하면 되므로, 필요한 모델링 변환이 단순해진다. 그림 2.58은 지금 분석을 한 물체들 간의 종속 관계를 보여주고 있다.

아래의 프로그램 예 2.9에는 지금까지 기술한 애니메이션을 해주는 프로그램 중 모델링 변환에 관련된 함수들이 주어져 있다. 우선 애니메이션을 구동하는 과정을 생각해보자. 애니메이션을 하기 위해 조금씩 변하는 영상을 반복적으로 그려주려면, 누군가가 조금씩 변하는 렌더링 인자들에 대한 이미지 프레임을 그려주는 함수(여기서는 `render()` 함수)를 지속적으로 불러주어야 한다. 우선 `render()` 함수는 이 프로그램에서 GLUT의 디스플레이 컬백 함수로 등록된 `display()` 함수에서 호출을 하고 있다[19]. 따라서 움직이는 장면을 도시하기 위해서는 디스플레이 컬백 함수를 주기적으로 호출해주는 수단이 필요하다.

[19] 실제로 이렇게 간접적으로 `render()` 함수를 호출할 할 필요는 없다. 디스플레이 컬백 함수로 `render()` 함수를 직접 등록해도 무방하나, 여기서는 나중의 확장성을 생각해서 이런 방식으로 프로그래밍을 하였다.

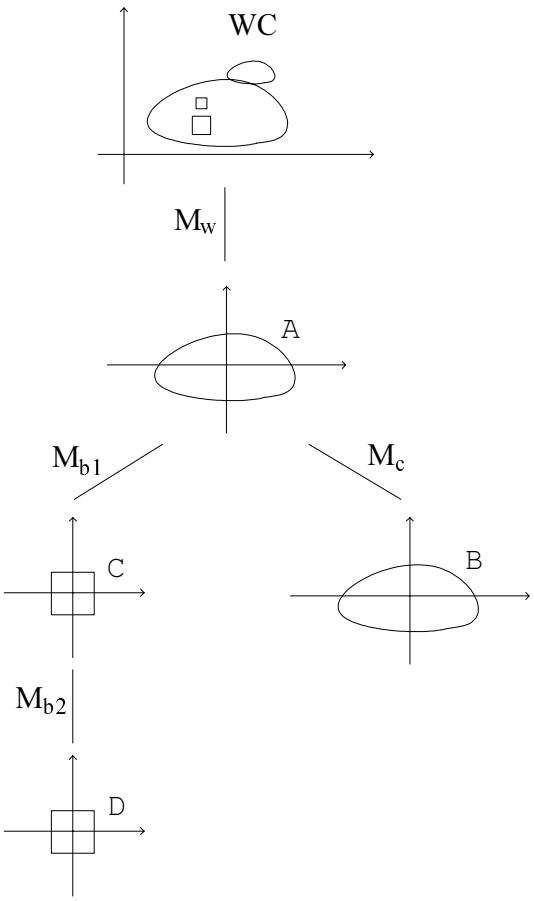



그림 2.58: 물체간의 계층 구조

윈도우 프로그래밍에서 사용하는 전형적인 방법은 응용 프로그램이 윈도우 시스템을 총괄적으로 관리하는 윈도우 서버에게, 윈도우 시스템이 이 응용 프로그램에 대한 이벤트가 발생하지 않고 있는 휴면 상태(idle state)에 들어갈 때마다, 특정 함수를 주기적으로 불러 달라고 요청을 하는 것이다. 일반적으로 윈도우 시스템에서 제공하는 윈도우 함수를 사용하면 함수를 호출해주는 시간 간격을 설정할 수 있으나(예를 들어, 100ms와 같이), 여기서 사용하는 GLUT 라이브러리에서는 간격을 명시적으로 지정할 수는 없고, 단지 고정된 시간 간격을 사용한다. `register_callback()` 함수를 보면 디스플레이 컬백 함수와 키보드 컬백 함수를 등록을 한 후, 마지막 문장 `glutIdleFunc(next_frame);`에서 반복적으로 호출이 될 함수로 `next_frame`을 등록하고 있다. GLUT 함수 void glutIdleFunc(void (*func)(void));는 휴면 상태에서 주기적으로 호출이 될 아이들 컬백 함수(idle callback function)를 등록하는데 사용이 되는데, 아이들 컬백 기능을 정지하고 싶으면 이 함수의 인자 func에 NULL 값을 넣어 다시 한 번 등록을 하면 된다.

**프로그램 예 2.9** 어미소의 애니메이션을 위한 기하 변환.

```
void register_callback (void) {
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  glutIdleFunc(next_frame);
}
      ⋮
void display (void) {
  render();
}
float angle = 0.0;

void next_frame(void) {
  angle = ((int) (angle + 1.0)) % 360;
  glutPostRedisplay();
```

```
}
      ⋮
void render(void) {
     ⋮
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
 gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
     ⋮
draw_axis();

 glPushMatrix();
  glRotatef(angle, 0.0, 1.0, 0.0); // Line (a)
  glTranslatef(0.0, 0.0, 5.0); // Line (b)
  glScalef(6.5, 6.5, 6.5); // Line (c)
  draw_cow(1.0, 0.0, 0.0); // draw object A
  glPushMatrix();
   glRotatef(10.0*angle, 1.0, 0.0, 0.0); // Line (d)
   glTranslatef(-0.1, 0.0, 0.3); // Line (e)
   glScalef(0.025, 0.025, 0.025); // Line (f)
   draw_box(); // draw object C
   glRotatef(40.0*angle, 1.0, 0.0, 0.0); // Line (g)
   glTranslatef(0.0, 4.0, 0.0); // Line (h)
   glScalef(0.4, 0.4, 0.4); // Line (i)
   draw_box(); // draw object D
  glPopMatrix();
  glTranslatef(0.15, 0.3, 0.0); // Line (j)
  glScalef(0.3, 0.3, 0.3); // Line (k)
  draw_cow(0.0, 0.0, 1.0); // draw object B
 glPopMatrix();
 glutSwapBuffers();
}
```

아이들 컬백 함수로 등록된 next_frame() 함수를 보면, 이 함수는 매번 호출이 될 때마다 전역 변수인 angle 값을 0과 359사이에서 순환적으로 1만큼씩 증가를 시킨다. 이 값은 현재 그리려는 프레임에서 어미소가 $y_w$축 둘레로 몇 도 회전한 곳에

위치하는지를 정해준다. 이렇게 새롭게 프레임을 그리는데 필요한 렌더링 인자를 설정을 한 후 `glutPostRedisplay();` 문장을 수행시켜 디스플레이 컬백 함수가 호출이 되도록 하고 있다. 따라서 `next_frame()` 함수를 아이들 컬백 함수를 등록을 함으로써 결과적으로 변수 `angle` 값이 1씩 증가되면서 `render()` 함수가 반복적으로 호출이 되게 된다.

이제 `render()` 함수를 살펴보자. 여기서는 뷰폿, 투영, 그리고 뷰잉 변환을 마친 후, 앞 절에서 자동차를 그린 방식과 마찬가지로 이 장면에 내재하는 계층 구조에 대한 트리(그림 2.58)를 깊이 우선 탐색을 하며 렌더링을 하고 있다. 우선 루트 노드인 세상 공간에서 아래로 내려갈 때, 스택을 푸쉬하여 현재의 모델뷰 행렬 스택의 탑에 있는 뷰잉 행렬을 보존한다. 다음 어미소 A에 대한 노드에서 기하 물체를 그려 주어야 하는데, 그 전에 변에 붙은 변환 행렬 $M_w$를 스택에 올려주어야 한다. 이 함수에서는 `draw_cow(*)` 함수를 호출하기 직전의 세 개의 OpenGL 함수(Line (a), (b), (c))가 어미소를 세상 좌표계로 보내주는 변환 $M_w = R(angle, 0, 1, 0) \cdot T(0, 0, 5) \cdot S(6.5, 6.5, 6.5)$을 스택에 푸쉬하는 역할을 한다. 그 결과 모델링 좌표계의 소를 크게 확대한 후, $z_m$축(화면 왼쪽 아래쪽으로 향하는 축)으로 옮겨 각도 `angle` 만큼 $y_m$축 둘레로 회전을 하여 세상에 배치하게 된다[20]. 각 프레임에 대하여 `angle` 값이 1씩 증가하므로, 이러한 방식으로 어미 소가 1도씩 회전하는 효과를 낼 수 있다.

다음 트리에서 물체 C에 대한 노드로 내려갈 때, 큰 정육면체를 어미소의 모델링 좌표계 공간에서 몸통 둘레로 회전을 시키기 위하여(기하 변환 $M_{b1} = R(10 \cdot angle, 1, 0, 0) \cdot T(-0.1, 0, 0.3) \cdot S(0.025, 0.025, 0.025)$), 마찬가지로 크기-이동-회전 변환을 위한 OpenGL 함수를 호출하고 있다(Line (d), (e), (f)). 여기서는 원래 자신의 모델링 좌표계에서 각 변의 길이가 2인 정육면체의 크기를 축소한 다음 $x_m$축으

[20] 이 예에서는 세상 좌표계와 소의 모델링 좌표계의 좌표축의 방향이 서로 일치한다.

로 -0.1과 $z_m$축으로 0.3만큼 이동시킨 후, $x_m$축 둘레로 반지름이 0.3인 원을 따라 `10*angle` 각도만큼 회전을 시키고 있다. 소의 모델링 좌표계에서 머리에서 꼬리까지가 $x_m$축을 따라 놓여져 있으므로, 이러한 회전의 결과 정육면체 C가 어미소의 몸통을 둘레로 돌게 된다. 한 가지 짚고 넘어가야 할 점은 어미소는 비교적 크게 보이는데 과연 반지름이 0.3인 원을 따라 회전하면 어미소의 몸통 둘레로 회전을 할 수 있는가 하는 사실이다. 여기서 반드시 이해를 해야 할 것은 큰 정육면체를 세상 좌표계로 직접 보내는 것이 아니라, Line (c)에서 6.5배 확대되기 전의 비교적 작게 모델링이 된 원래의 어미소의 좌표계로 보내는 것이므로 문제가 없고, 어미소의 크기가 확대될 때 회전 반지름도 같은 비율로 커진다는 점이다.

다음 Line (g), (h), (i)에서 적절한 변환을 하여 작은 정육면체가 큰 정육면체보다 네 배 빠른 속도로 회전을 할 수 있도록 큰 정육면체의 좌표계로 보내주고 있다. 이 OpenGL 문장들은 그림 2.58에서 물체 C의 노드에서 물체 D의 노드로 내려 갈 때의 변환 $M_{b2} = R(40 \cdot angle, 1, 0, 0) \cdot T(0, 4, 0) \cdot S(0.4, 0.4, 0.4)$를 합성하고 있는데, 앞에서 설명한 대로라면 트리의 아래로 내려갈 때 푸쉬를 하고(Line (g) 직전에서), 올라올 때 팝을 해야(`draw_box();` 문장 직후에서) 한다. 그러나 여기서는 물체 D를 그린 후 Line (g) 직전의 스택의 탑의 내용을 보존할 필요가 없기 때문에 생략을 하고, 물체 D를 그린 후의 탑의 행렬 $M_V \cdot R(angle, 0, 1, 0) \cdot T(0, 0, 5) \cdot S(6.5, 6.5, 6.5) \cdot R(10 \cdot angle, 1, 0, 0) \cdot T(-0.1, 0, 0.3) \cdot S(0.025, 0.025, 0.025) \cdot R(40 \cdot angle, 1, 0, 0) \cdot T(0, 4, 0) \cdot S(0.4, 0.4, 0.4)$를 제거한다. 이 때의 `glPopMatrix();` 문장(Line (i)의 다음 다음 문장)의 팝 연산은 개념적으로 탐색을 마친 후 트리의 물체 C에 대한 노드에서 물체 A에 대한 노드로 올라오는 것과 대응이 된다는 점을 명심하기 바란다. 마지막으로 송아지 B를 그리기 위해 Line (j)와 (k)에서 송아지를 어미소의 등으로 보내는 변환 $M_c = T(0.15, 0.3, 0) \cdot S(0.3, 0.3, 0.3)$을 수행한 후 송아지를 그리게 된다. 여기서도

위에서와 같은 이유로 푸쉬와 팝을 위한 OpenGL 함수를 호출하지 않았다.

## 4.3 물체의 대화식 조작

지금까지 살펴 본 OpenGL 뷰잉의 예에서는 모두 기하 물체를 구성하는 기하 프리미티브들이 기하 파이프라인의 물체 좌표계에서 출발하여 단계적인 변환을 통하여 윈도우 좌표계를 향하여 흘러갔다. 응용 소프트웨어를 개발할 때에는 종종 개념적으로 이와는 반대의 방향으로 데이터가 흘러가야 하는 경우가 발생한다. 예를 들어 회로 설계 소프트웨어나 3차원 기하 모델링 소프트웨어를 개발할 때 반드시 필요한 기능 중의 하나는 사용자가 원하는 물체를 집어내어(picking)[21], 그 물체에 대하여 대화식으로 위치를 옮겨주거나, 변경을 가하거나 또는 제거하는 등의 조작을 하는 것이다. 보통 사용자가 마우스를 화면상의 물체가 있는 위치로 옮긴 후, 마우스 버튼을 눌러 물체를 집어내는데, 과연 어떠한 방식으로 이러한 집어내기 계산이 수행이 될까?

집어내기 기능을 사용하려면 사용자가 원하는 물체를 선택할 수 있도록 우선 기하 물체들을 렌더링 하여 윈도우에 그림을 그려주어야 한다. 이후 그 그림을 보고 원하는 물체를 가리키기 위하여 마우스로 클릭을 하면, 윈도우 시스템에서 클릭을 한 지점의 좌표 $(x, y)$를 알려주게 되는데, 이 때 윈도우 안에서의 이 지점에 투영된 기하 물체를 찾아내야 한다. OpenGL 기하 파이프라인 관점에서 생각해보면, 개념적으로 윈도우 좌표계가 주어졌을 때 그림을 그리기 위하여 사용한 기하 변환의 역변환을 수행하여 물체 좌표계로 가서 그에 해당하는 기하 물체를 찾아주어야 한다. 하지만 약간 생각을 해보면 이러한 방식으로 집어내기 기능을 구현한다는 것이 그

[21] 컴퓨터 그래픽스 분야에서는 이러한 행위를 보통 picking이라 한다. 이 단어에 적절한 우리 용어로 '선택'이라는 단어를 들 수가 있으나, 여기서는 어색하기는 하나 '집어내기'라는 용어를 사용하겠다. 대신 '선택'은 이 장에서 설명할 OpenGL의 selection 기능을 나타낼 때 사용하기로 하겠다.

리 수월한 작업이 아님을 알 수 있다.

무엇보다도 OpenGL과 같은 API를 사용하여 렌더링을 하면, 전체 렌더링 파이프라인 중 윈도우 좌표계까지의 기하 변환을 마친 후, 뒤에서 설명할 래스터화와 프래그먼트별 연산 과정이 진행이 된다. 이 두 과정에서 기하 물체에 대한 기하학적인 정보를 다 잃어버리게 되고, 궁극적으로 단지 윈도우의 어느 화소에 어떤 색깔을 칠할 것인지에 대한 래스터 이미지 정보만 남게 된다. 따라서 마우스 클릭의 결과 2차원 평면인 윈도우 좌표계의 한 점이 주어졌을 때, 이를 굳이 3차원 공간인 물체 좌표계로 매핑을 할 경우, 직선, 즉 그 점으로 투영이 되는 투영선이 계산이 될 것이다. 물론 물체 좌표계에서 이 직선과 모든 기하 물체간의 교점을 계산하여, 어떤 물체들이 클릭을 한 위치로 투영이 되는지를 결정할 수는 있겠지만, 이를 위해서는 장면이 약간만 복잡해도 실시간 렌더링에서는 허용되지 않을 정도의 많은 계산량이 요구된다. 그러면 응용 소프트웨어를 개발하는 데 있어 필수적인 집어내기 기능을 어떻게 구현을 할 수 있을까? OpenGL에서는 두 가지 방법을 생각할 수 있는데, 여기서는 OpenGL에서 제공하는 선택(selection) 기능을 사용하여 구현하는 예에 대하여 살펴보도록 하겠다.

243쪽의 프로그램 예 2.11에는 앞 절에서 설명한 프로그램에 간단한 집어내기 기능을 추가하여 발전시킨 프로그램의 일부가 주어져 있다(예제 프로그램 2.E참조). 이 프로그램을 수행시키면, 앞에서와 같이 어미소, 송아지, 그리고 두 개의 정육면체가 회전을 한다. 이 때 ‘s’(stop) 키를 누르면 회전을 멈추는데, 다시 회전하게 하려면 ‘c’(continue) 키를 누르면 된다. 앞에서 설명한 바와 같이 이러한 기능을 구현하기 위해서는 키보드 컬백 함수에서 상황에 따라 아이들 컬백 함수를 적절히 등록을 해주면 된다. 다음 정지된 상태에서 어미소나 송아지가 있는 곳으로 마우스를 움직여 왼쪽 마우스를 클릭을 하면, 해당 동물을 집어낼 수가 있다. 이 때 그러한

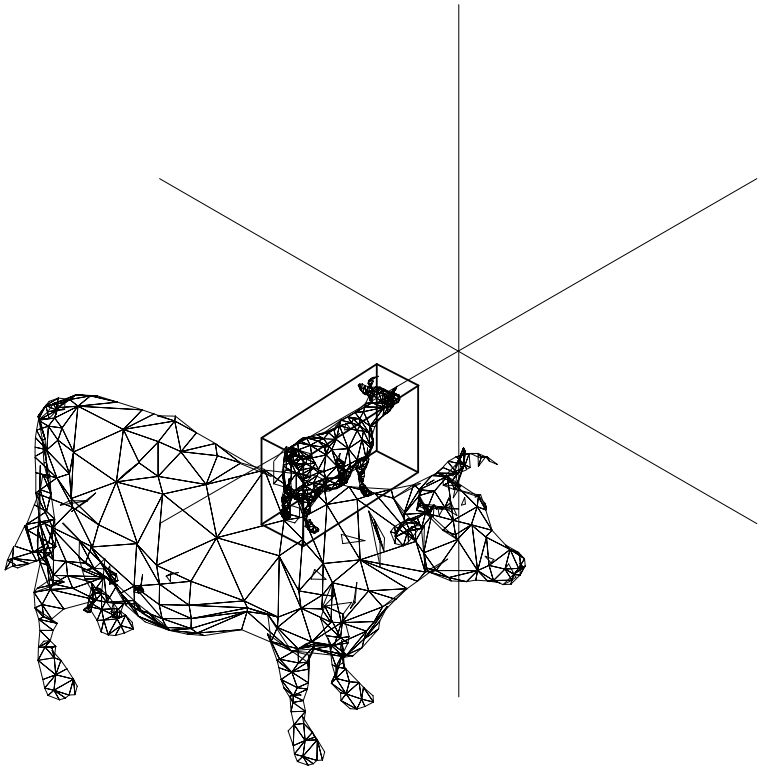


그림 2.59: 송아지의 선택 장면

사실을 알려주기 위하여 같은 색깔의 바운딩 박스가 그려지는데, 그림 2.59는 송아지를 집어냈을 때의 모습을 보여주고 있다. 한 물체를 집어낸 후, 그 물체에 대하여 다시 왼쪽 마우스를 클릭을 하면 바운딩 박스가 없어지면서 집어내기가 취소가 된다. 이제 송아지를 집어내어보자. 그 상태에서 오른쪽 마우스를 매번 클릭을 할 때마다 송아지가 5도씩 회전을 하는데, 원하는 만큼 회전을 시킨 후 'c' 키를 누르면 회전된 상태에서 다시 회전을 시작하게 된다. 지금까지 설명한 새로운 기능을 추가하기 위해서 무엇보다도 어미소나 송아지를 집어낼 수 있어야 하는데, 이 프로그램에서 사용한 OpenGL의 선택 기능에 대하여 알아보자.

지금까지는 이에 대하여 설명을 하지 않았었는데, OpenGL에서는 기하 물체를 렌더링 하라는 명령을 내렸을 때, 이를 처리하는 방식으로 렌더링 모드(rendering mode), 선택 모드(selection mode), 그리고 피드백 모드(feedback mode) 등 세 가지의 그림을 그리는 모드를 제공한다. 이러한 모드들은 GLint glRenderMode(GLenum mode); 함수의 인자 mode에 GL_RENDER, GL_SELECT, 그리고 GL_FEEDBACK 중 적절한 OpenGL 상수를 사용하여 설정할 수가 있다. 렌더링 모드는 그림을 그리라는 명령을 내리면 그에 대한 전 계산 과정이 수행이 되고, 그에 따라 프레임 버퍼의 내용이 변경되어 화면에 그림이 나타나게 되는 정상적인 상태의 모드로서, 디폴트 모드로 지정이 되어 있다.

반면에 선택 모드와 피드백 모드에서는 렌더링 계산이, 부분적으로, 수행이 되나 그 결과가 프레임 버퍼에 그려지는 것이 아니라, 각 모드에 해당하는 렌더링 정보를 선택 버퍼(selection buffer)와 피드백 버퍼(feedback buffer)를 통하여 응용 프로그램에게 돌려준다. 따라서 이 두 모드에서 렌더링을 하면 화면의 내용에는 변함이 없다. 선택 모드에서는 만약 렌더링 모드를 사용하여 렌더링을 하였을 경우에 화면에 그려졌을 기하 프리미티브들에 대한 정보가 리턴이 되고, 피드백 모드에서는 화

면에 그려졌을 기하 프리미티브들의 좌표 값들이 리턴이 되는데, 이 절에서는 선택 모드에 대하여 구체적으로 살펴보겠다.

선택 모드에서는 선택 버퍼를 통하여 화면에 그려졌을 기하 프리미티브에 대한 정보를 얻게 된다고 하였는데, 이것이 의미하는 바를 정확하게 알아보자. 과연 가상의 세상에 존재하는 기하 물체를 구성하는 프리미티브들 중 어떤 것들이 화면에 그려질까? 바로 뷰잉 볼륨과 조금이라도 교차하는 프리미티브들이 그려지게 된다. 앞에서 설명한 바와 같이 뷰잉 볼륨은 무엇보다도 뷰잉 변환과 투영 변환에 의하여 설정이 된다고 하였다. 이러한 뷰잉 볼륨은 세상 좌표계 공간에서 여섯 개의 절단 평면에 의하여 결정이 된다고 볼 수 있는데, 좀 더 정확히 말하면 여기서 고려하는 뷰잉 볼륨은 이러한 여섯 개의 절단 평면 외에 사용자가 세상 공간에서 glClipPlane(*) 함수를 사용하여 설정한 절단 평면들이 함께 정의하는 공간을 의미한다.

선택 모드에서 렌더링을 하면 세상의 물체들을 그리는 과정에서 이러한 뷰잉 볼륨과 조금이라도 교차하는, 즉 전체 또는 일부가 뷰잉 볼륨 안에 있는, 프리미티브들에 대한 정보가 리턴이 된다. 한 기하 프리미티브가 뷰잉 볼륨과 교차를 할 때, 선택 히트(selection hit)가 일어났다고 하는데, 그에 대한 정보가 히트 레코드(hit record) 형태로 선택 버퍼에 기록이 된다.

OpenGL에서의 선택 기능을 간단한 예제 프로그램을 통하여 이해해보자. 그림 2.60은 예제 프로그램 2.F의 프로그램을 수행시킬 때의 한 장면을 보여주고 있고, 아래의 프로그램 예 2.10에는 선택 기능과 관련된 주요 함수들이 주어져 있다. 이 프로그램을 수행시키면, 처음에 빨간색, 초록색, 파란색 등의 세 가지 색깔의 정사각형이 그려진다. 이 상황에서 마우스를 윈도우 안으로 옮겨 'n'(next) 키를 누를 때마다 이 사각형들의 위치가 무작위로 결정이 되어 그려지게 되는데, 이와 함께 윈도우 안에 조금이라도 그려지는 사각형에 대한 정보가 출력이 된다. 예를 들어 그

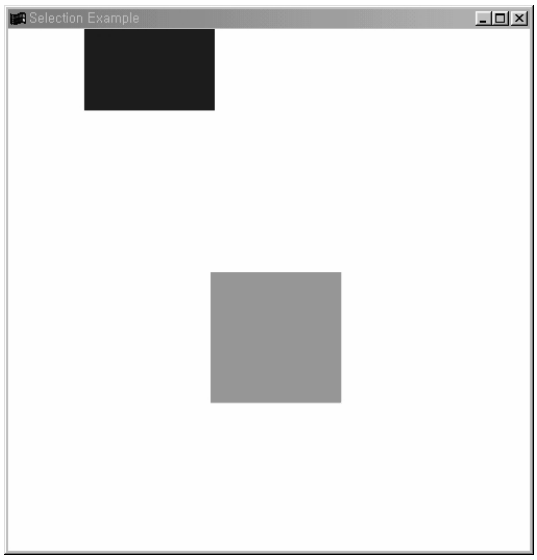



그림 2.60: 프로그램 예 2.10 수행 예

림 2.60과 같은 경우 다음과 같은 문장이 출력이 된다.

```
$$$ 2 square(s) drawn:  GREEN BLUE
```

사각형을 그릴 때 조금이라도 화면에 나타난다는 것은 곧 그 사각형이 뷰잉 볼륨과 조금이라도 교차를 한다는 사실을 의미한다. 따라서 이러한 프로그램은 OpenGL의 선택 기능을 사용하면 쉽게 구현을 할 수가 있다. OpenGL에서 선택을 하는 방법은 다음과 같다. 우선 렌더링을 할 때, 선택을 하려하는 기하 프리미티브 또는 여러 개의 프리미티브들로 구성된 기하 물체에 대하여 이름을 붙여준다. 다음 선택 모드로 들어가 원하는 렌더링을 한 후, 다시 렌더링 모드로 돌아오게 되는데, 이 때 OpenGL 시스템에서 선택된, 즉 뷰잉 볼륨과 일부라도 교차하는 프리미티브나 물체에 대하여 이름을 비롯한 관련 정보를 선택 버퍼를 통하여 넘겨받게 된다. 다음 선택 버퍼에 들어있는 히트 레코드들의 내용을 분석하여 어떤 프리미티브나 물체가 선택이 되었는지 파악을 하게 된다.

이제 프로그램 예 2.10을 살펴보자. 디스플레이 컬백 함수인 `display()` 함수를 보면 적절한 기하 변환을 한 뒤, `draw_squares(*)` 함수를 호출하여 렌더링 모드에서

세 개의 사각형을 그리게 된다(Line (k)). 이 함수는 인자 mode가 GL_RENDER일 때는 단순히 서로 다른 색깔과 위치를 가지는 사각형 세 개를 그려 준다. 실제로 사각형을 그리는 draw_square(*) 함수의 첫 세 인자는 그리고자 하는 사각형의 색깔을, 그리고 나머지 두 개의 인자는 사각형 중심의 위치를 나타낸다. 다시 display() 함수로 돌아와 Line (l)에서 변수 first_drawing의 값이 참이면 함수 select_squares()를 호출을 한다. 이 변수 값은 사용자가 'n' 키를 누르면 참 값으로 설정이 되고, 선택 모드에서 그림을 한 번 그린 후 다시 거짓 값으로 설정이 되는데, 이 변수를 사용하는 이유는 'n' 키를 눌렀을 때 처음 한 번에 대해서만 선택을 하기 위해서이다. 따라서 어떤 이유에서이건 윈도우의 내용을 다시 그려야 할 필요가 생겼을 때에는 선택 기능을 사용하지 않고 사각형들만 다시 그려주게 된다.

키보드 컬백 함수인 keyboard(*) 함수를 보면 사용자가 'n' 키를 눌렀을 때, 변수 first_drawing을 참 값으로 지정을 하고(Line (m)), set_new_centers() 함수에서 난수를 발생시켜 사각형들의 위치를 새롭게 변경을 한 후(Line (n))[22], 윈도우 서버에게 다시 그림을 그리게 해달라고 요청을 하고 있다(Line (o)). 그러면 다시 display() 함수가 수행이 되는데, 이번에는 Line (l)에서 first_drawing이 참이므로 select_squares() 함수가 호출이 된다.

**프로그램 예 2.10** 간단한 선택 모드 사용 예.

```
#define SQ_HSIZE 0.15
#define RED 0
#define GREEN 1
#define BLUE 2

int first_drawing;
float rx=-0.4, gx=0.0, bx=0.4, ry=0.0, gy=0.0, by=0.0;
```

[22] 참고로 전역 변수 (rx, ry), (gx, gy), (bx, by)는 각각 빨간색, 초록색, 그리고 파란색 사각형의 현재 중심 좌표 값을 나타낸다.

```
double w, h;
      ⋮
void draw_square(float r, float g, float b, float cx, float
cy) {
 glPushMatrix();
  glColor3f(r, g, b);
  glTranslatef(cx, cy, 0.0);
  glRectf(-SQ_HSIZE, -SQ_HSIZE, SQ_HSIZE, SQ_HSIZE);
 glPopMatrix();
}

void draw_squares(GLenum mode) {
 if (mode == GL_SELECT) glInitNames(); // Line (a)

 if (mode == GL_SELECT) glPushName(RED); // Line (b)
 draw_square(1.0, 0.0, 0.0, rx, ry);

 if (mode == GL_SELECT) glLoadName(GREEN); // Line (c)
 draw_square(0.0, 1.0, 0.0, gx, gy);

 if (mode == GL_SELECT) glLoadName(BLUE); // Line (d)
 draw_square(0.0, 0.0, 1.0, bx, by);

 if (mode == GL_SELECT) glPopName(); // Line (e)
}

void print_names(GLint hits, GLuint *buf) {
 int i, j;
 GLuint n, *ptr;

 printf("\n$$$ %d square(s) were drawn:", hits);

 ptr = (GLuint *) buf;
 for (i = 0; i < (int) hits; i++) {
  n = *ptr; ptr++; ptr++; ptr++; // Line (f)
  for (j = 0; j < (int) n; j++) {
   switch (*ptr) {
```

```
      case RED: printf(" RED"); break;
      case GREEN: printf(" GREEN"); break;
      case BLUE: printf(" BLUE"); break;
     }
     ptr++;
    }
  }
  printf("\n");
}


#define BUF_SIZE 256
void select_squares(void) {
  GLuint selbuf[BUF_SIZE];
  GLint hits;


  glSelectBuffer(BUF_SIZE, selbuf); // Line (g)
  glRenderMode(GL_SELECT); // Line (h)
  draw_squares(GL_SELECT);
  hits = glRenderMode(GL_RENDER); // Line (i)
  print_names(hits, selbuf); // Line (j)
  first_drawing = 0;
}


void display(void) {
  glClearColor(0.259, 0.259, 0.435, 1.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-0.6, 0.6, -0.6, 0.6, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0, 1.0, 0.0);
  draw_squares(GL_RENDER); // Line (k)
  if (first_drawing) select_squares(); // Line (l)
  glFlush();
}
```

```
void keyboard(unsigned char key, int x, int y) {
  switch (key) {
    case 'n':
      first_drawing = 1; // Line (m)
      set_new_centers(); // Line (n)
      glutPostRedisplay(); // Line (o)
      break;
        ⋮
}
```

select_squares() 함수는 바로 선택 기능을 사용하여 어떤 사각형이 화면에 나타나는지를 알아내는 역할을 한다. 이 함수를 보면 Line (g)에서 glSelectBuffer(*) 함수를 호출하고 있다. 앞에서 교차 기록, 즉 히트 레코드는 선택 버퍼에 담겨져 온다고 하였는데, 우선 그러한 정보가 담겨질 선택 버퍼를 지정을 하여야 한다. 함수 void glSelectBuffer(GLsizei size, GLuint *buffer);는 무부호 정수 타입의 원소를 가지는 배열을 사용하여 선택 버퍼를 지정하는데 사용된다. 여기서 buffer는 배열에 대한 포인터로서 size개의 원소를 가지는 배열이 지정이 된다. 다음 그리는 모드를 선택 모드로 바꾼 후(Line (h)), draw_squares(GL_SELECT); 문장을 통하여 선택 모드에서 사각형들을 다시 한 번 그려주게 된다.

draw_squares(*) 함수를 다시 보면 이번에는 렌더링 모드에서와는 달리 약간의 작업을 더 해주게 된다. 우선 앞에서 기하 프리미티브나 여러 개의 기하 프리미티브들로 구성된 물체를 지칭하기 위하여 이름을 붙여주어야 한다고 했다. 이를 위하여 OpenGL에서는 이름 스택(name stack)이라 하는, 무부호 정수 타입의 원소를 가지는 또 하나의 스택을 사용한다. 우선 Line (a)에서 이름 스택을 초기화 해준다. void glInitNames(void); 함수의 목적이 이름 스택을 초기화하는 것인데 주의해야 할 것은 초기 상태에는 스택의 내용이 비어 있게 된다는 점이다. 다음 이 스택에 적절

한 값을 넣어주어 기하 프리미티브나 물체에 이름을 붙여 주게 된다.

Line (b)에서 첫 번째 그릴 사각형의 이름을 RED로 붙여주고 있다. 함수 void glPushName(GLuint name);는 그 이름이 나타내듯이 이름 스택에 푸쉬를 한 후 그 탑에 무부호 정수 타입의 이름 name을 올려준다. 물론 이 함수의 마지막 문장에서처럼 팝은 void glPopName(void); 함수에 의해 수행이 된다. 푸쉬를 한 후 다음 문장에서 첫 번째 사각형, 즉 빨간 사각형을 그려준다. 다음 Line (c)에서 이번에는 glLoadName(*) 함수를 통하여 다음 프리미티브의 이름 GREEN을 올린 후 초록색의 사각형을 그리는데, 이처럼 항상 그리려는 프리미티브나 물체의 이름을 스택에 적절히 올린 후 렌더링을 하는 방식을 취한다. 참고로 void glLoadName(GLuint name); 함수는 현재의 이름 스택의 탑의 내용을 name으로 바꾸어 주는 함수이다.

선택 모드에서 이름이 붙여진 물체를 그릴 때 뷰잉 볼륨과 교차하는 물체들에 대한 정보가 어떻게 선택 버퍼에 저장이 될까? 이 예에서는 각 물체가 한 개의 프리미티브로 구성이 되어 있지만, 많은 경우 어미소나 송아지처럼 물체는 다수의 프리미티브들로 구성이 될 것이다. 만약 그렇다면 한 물체가 뷰잉 볼륨 안에 들어온다고 할 때 그것을 구성하는 모든 다각형에 대하여 히트 기록을 선택 버퍼에 저장을 하는 것은 상당한 낭비일 것이다. 따라서 OpenGL에서는 교차하는 프리미티브들에 대한 정보를 저장하고 있다가, 1. 이름 스택에 관련된 OpenGL 함수가 수행이 되거나, 2. glRenderMode(*) 함수가 호출이 될 때마다, 바로 이전의 히트 레코드를 생성한 직후부터 현재까지 저장하고 있던 정보를 요약하여 한 개의 히트 레코드를 작성하여 선택 버퍼에 저장을(물론 교차가 발생할 경우에만) 한다. 즉 동일한 이름을 가지는 한 물체에 대하여 여러 개의 다각형이 교차를 하더라도 이에 대해서는 여러 개의 히트 레코드가 생성이 되지는 않는다.

각 히트 레코드에는 다음과 같은 네 가지 부류의 정보가 담겨진다. 첫째, 어떤 물

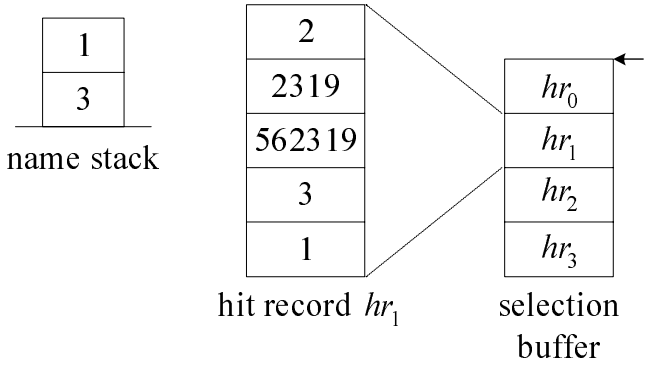



그림 2.61: 히트 레코드의 내용 예

체가 교차를 하였을 때 당시의 이름 스택에 올려져 있는 이름의 개수, 즉 스택의 깊이가 저장된다. 둘째, 보통 히트 레코드를 생성할 때 한 개 이상의 프리미티브들이 교차를 하는데, 뷰잉 볼륨과 교차하는 모든 프리미티브들의 윈도우 좌표계에서의 $z_{wd}$ 값들의 최소 값과 최대 값이 저장이 된다. 윈도우 좌표계에서는 이 값이 [0, 1] 사이의 범위에 들어오는데, 여기에 $2^{32} - 1$을 곱해 무부호 정수로 바꾸어, 최소 값, 그리고 다음에 최대 값을 저장한다. 이 값들은 추후 집어내기를 할 때, 여러 물체들이 동일한 선택 지점으로 떨어질 경우 물체간의 깊이 정보, 즉 전후 관계에 대한 정보를 제공해주기 때문에, 상황에 맞는 적절한 물체를 집어 낼 수 있도록 해준다. 마지막으로 현재 히트 레코드를 생성할 때의 이름 스택의 내용, 즉 스택에 올려진 이름들을 탑 쪽의 이름이 가장 마지막에 나오는 순서로 저장한다. 따라서 만약에 그림 2.61에서처럼, 히트 레코드 생성시의 이름 스택의 내용이 왼쪽과 같다면 오른쪽과 같은 히트 레코드가 선택 버퍼에 저장이 된다.

이 예에서는 빨간색 사각형이 교차하였다면 그에 대한 히트 레코드는 Line (c)가 수행이 될 때, 그리고 초록색과 파란색 사각형에 대한 히트 레코드는 각각 Line (d)와 Line (e)가 수행이 될 때 선택 버퍼에 저장이 될 것이다. 앞에서 설명한 바와 같이 선택 모드에서는 실제로 프레임 버퍼에 그림이 그려지는 것이 아니라, 렌더링 과

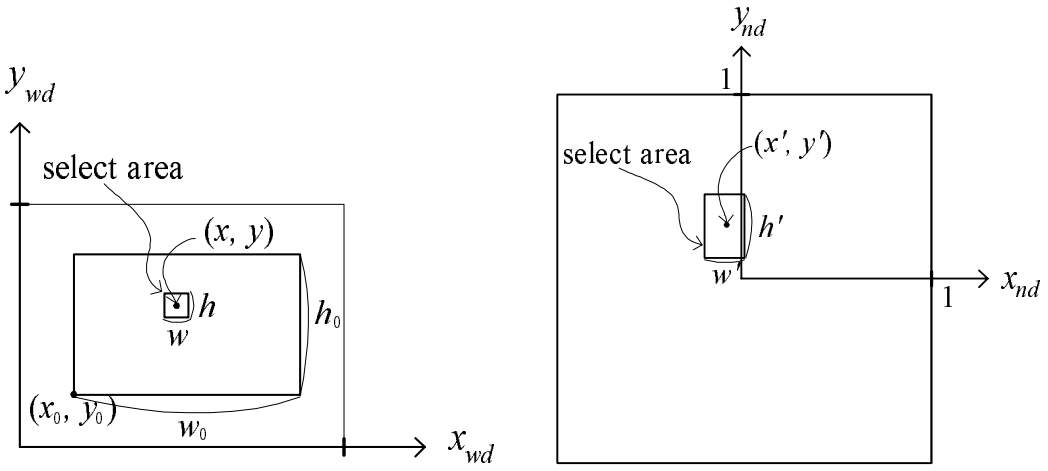
정 중의 교차 정보를 파악하여 그러한 결과를 선택 버퍼에 저장을 하게 된다. 이렇게 `select_squares()` 함수의 수행을 마친 후 Line (i)에서 다시 그림을 그리는 모드를 정상 모드인 렌더링 모드로 바꾸어 주게 된다. 함수 정의를 보면 알 수 있듯이 glRenderMode(*) 함수는 정수 타입의 값을 돌려주는데, 이 값은 이 함수를 수행시킬 당시의 그림 그리기 모드가 GL_SELECT이거나 GL_FEEDBACK일 때만 의미를 가진다. 특히 전자의 경우에는 선택 버퍼에 담겨오는 히트 레코드의 개수가 되는데, 만약 이 값이 음수라면 이는 버퍼에 오버플로우가 생긴 것을 의미한다.

Line (i)의 수행 결과 변수 `hits`는 이제 윈도우 안에 그려지는 사각형의 개수 값을 가지게 되는데, Line (j)에서 `print_names(*)` 함수를 호출하여 선택 버퍼의 내용을 해석한 후 그것을 출력하게 된다. 앞에서 설명한 히트 레코드의 형식을 이해하였다면 이 함수의 내용을 어렵지 않게 이해할 수 있을 것이다. 참고로 Line (f)에서 변수 `n`은 이 예에서는 항상 1 값을 가질 것이고, 또한 세 개의 `ptr++;` 문장 중 뒤의 두 개는 깊이 정보의 최소 값과 최대 값을 건너뛰기 위한 것임을 명심하기 바란다.

지금까지는 OpenGL에서 제공하는 기본적인 선택 기능에 대하여 살펴 보았는데, 어떻게 보면 이러한 기능과 이 절의 주제인 집어내기와는 별 상관이 없어 보인다. 그러나 약간의 트릭을 사용하면 집어내기는 선택 기능으로 쉽게 구현할 수가 있다. 일반적으로 집어내기를 할 때 마우스 커서로, 예를 들어 왼쪽 마우스를 클릭을 하여, 윈도우 상의 한 지점을 지정하면 그 점으로 투영되는 물체가 결정이 된다. 실제로는 이러한 것을 구현하기 위해서 사용자가 한 점을 선택하였을 때, 정확하게 그 점만이 아니라 그 점을 중심으로 하는 일정 영역 안으로 투영이 되는 물체들을 집어내는 방식을 취한다[23].

그림 2.62(a)는 $(x_0, y_0, w_0, h_0)$에 의해 결정되는 뷰폿 안에서의 선택 영역의 설정

[23] 편의상 이 영역을 선택 영역(select area)이라 하겠다.




(a) 윈도우 좌표계에서의 선택 영역

(b) 정규 디바이스 좌표계에서의 선택 영역

그림 2.62: 두 좌표계에서의 선택 영역

예를 보여주고 있다. 사용자가 지정한 점 $(x, y)$을 중심으로 가로, 세로 각 $w$, $h$ 화소인 사각형이 바로 선택 영역인데, $w$와 $h$ 값을 얼마나 크게 하는가에 따라 집어내기의 민감도(sensitivity)가 결정이 된다. 이제 이 영역 안으로 들어오는 기하 물체를 집어내려 한다. OpenGL의 투영 변환 과정에서 설정한 뷰잉 볼륨 안에서 선택 영역에 해당하는 공간에 대하여 생각하여 보자. 그림 2.63은 원근 변환을 사용할 때의 예를 보여주고 있는데, 눈 좌표계(EC) 공간에서 설정한 뷰잉 볼륨에 대하여 투영 변환($M_P$)을 수행하면, 정규 디바이스 좌표계(NDC 1)의 $[-1, 1] \times [-1, 1] \times [-1, 1]$ 영역으로 뷰 매핑이 된다고 하였다. 이 그림에서 빗금으로 친 영역은 선택 영역에 해당하는 공간인데 뷰 매핑의 결과 정규 디바이스 좌표계의 가느다란 직육면체 영역으로 매핑이 된다. 평행 투영을 사용하더라도 마찬가지로 눈 좌표계에서의 선택 영역에 해당하는 공간은 이와 같은 직육면체 영역으로 매핑이 될 것이다. 우리가 집어내기를 하기 위해서 필요한 것은 NDC 1에서 빗금으로 친 영역과 교차하는 기하 물체를 선택하는 것인데, 만약 그냥 앞에서와 같이 OpenGL의 선택 기능을 사용한

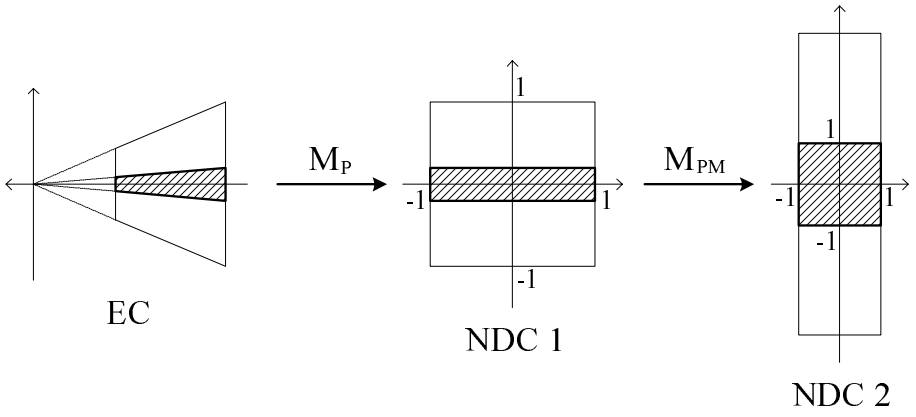



그림 2.63: 뷰잉 볼륨의 변환

다면 $[-1,1] \times [-1,1] \times [-1,1]$ 영역과 교차하는 모든 물체들이 선택이 될 것이다.

이러한 문제를 해결하는 한 가지 방법은 선택 영역에 해당하는 직육면체 영역이 뷰 매핑 후 $[-1,1] \times [-1,1] \times [-1,1]$ 영역으로 매핑이 되도록 해 주는 것이다. 그림 2.63에서 NDC 1으로 변환이 된 후, NDC 1의 빗금친 직육면체 영역을 $[-1,1] \times [-1,1] \times [-1,1]$ 영역으로 바꾸어 주는 간단한 변환 $M_{PM}$을 계산하여, 정상적인 투영 변환 $M_P$ 대신에 $M_P^{'} = M_{PM} \cdot M_P$를 수행할 경우, 바로 선택 영역에 해당하는 공간이 새로운 뷰잉 볼륨으로 대치되는 효과를 낳게 될 것이다. 이렇게 함으로써 OpenGL에서의 선택 기능을 사용하여 선택 영역에 그려지는 기하 물체를 집어낼 수가 있는 것이다.

이 때 $M_P$ 대신에 $M_P^{'}$를 사용하면 개념적으로는 선택 영역에 해당하는 공간과 교차하는 물체들만 화면에 그려지겠지만, 어차피 선택 모드에서는 화면에 그림이 그려지는 것이 아니기 때문에 아무런 상관이 없다. 집어내기를 위한 투영 변환의 변경을 쉽게 하기 위하여 OpenGL에서는 void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4]);라는 함수를 제공한다. 사용자가 이 함수의 처음 네 개의 인자 x, y, width($w$), height($h$)로 선택 영역을 설정하면, 그에 해당하는 변환 행렬 $M_{PM}$이 계산되어 현재 행렬 스택의 탑의 오른쪽에 곱해

진다. 이 행렬 변환을 계산하려면 뷰폿의 위치 $(x_0, y_0)$와 크기$(w_0, h_0)$에 대한 정보가 필요한데, 마지막 인자인 배열 viewport에 그러한 정보가 담겨가게 된다.

변환 행렬 $M_{MP}$는 비교적 쉽게 구할 수가 있다. 그림 2.63에서 NDC 1의 빗금친 직육면체의 단면을 정규 윈도우 상에서 다시 보면 그림 2.62(b)와 같은데, 바로 이와 같은 정규 윈도우와 그 안의 NDC 1에서의 선택 영역이 그림 2.62(a)와 같이 윈도우 좌표계의 뷰폿과 선택 영역으로 매핑이 된다. 이 경우 $x'$, $y'$, $w'$, $h'$은 윈도우 좌표계에서의 선택 영역을 NDC 1으로 역매핑을 함으로써 다음과 같이 구할 수가 있다.

$$x' = -1 + 2 \cdot \frac{x - x_0}{w_0}, \qquad y' = -1 + 2 \cdot \frac{y - y_0}{h_0}, \qquad w' = 2 \cdot \frac{w}{w_0}, \qquad h' = 2 \cdot \frac{h}{h_0}$$

$M_{PM}$는 단면이 $(x', y', w', h')$에 의해 정의되는 직육면체 영역을 $[-1, 1] \times [-1, 1] \times [-1, 1]$으로 매핑 해주는 변환이므로, 다음과 같이 이동 변환과 크기 변환의 곱으로 표현할 수가 있다.

$$M_{PM} = S(\frac{2}{w'}, \frac{2}{h'}, 1) \cdot T(-x', -y', 0) = \begin{bmatrix} \frac{w_0}{w} & 0 & 0 & \frac{w_0 + 2(x_0 - x)}{w} \\ 0 & \frac{h_0}{h} & 0 & \frac{h_0 + 2(y_0 - y)}{h} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \square$$

한 가지 주의해야 할 것은 렌더링 모드에서 투영 변환 $M_P$를 사용하여 정상적으로 렌더링을 한 후, 선택 모드에서 $M_P^{'}$를 투영 행렬 스택에 올릴 때, 일반적으로 $M_P$를 다시 계산하지 않기 위하여 그 내용을 보존시켜 주어야 한다는 점이다. 따라서 선택 모드에서 렌더링을 하려면 일반적으로 아래와 같은 형태의 코드를 수행시

켜야 한다.

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPickMatrix(···);
gluPerspective(···);
     ⋮    // draw objects in selection mode
glMatrixMode(GL_PROJECTION);
glPopMatrix();
```

이제 프로그램 예 2.11에 주어진 주요 함수들의 내용을 이해하여보자. 마지막에 있는 함수 `init_OpenGL()`에서는 이 프로그램의 수행에 필요한 초기화를 해주고 있는데, 무엇보다도 OpenGL 함수의 호출을 통하여 투영 행렬 스택과 모델뷰 행렬 스택의 내용을 적절히 설정하고 있다. 다음 제일 처음에 주어진 마우스 컬백 함수 `mouse(*)`를 보면, 정지 상태에서 왼쪽 마우스 버튼을 누르면, 해당 지점에 대하여 집어내기 기능을 수행해주는 함수 `pick_cow(*)`를 호출하고 있다(Line (a)). 이 함수의 내용을 보면 우선 Line (b)에서 현재 사용하고 있는 뷰폿에 대한 인자를 문의하여 배열 `viewport[4]`에 저장을 한다. 함수 void glGetIntegerv(GLenum pname, GLint *params);는 void glGetFloatv(*); 함수, void glGetDoublev(*); 함수, void glGetBooleanv(*); 함수 등과 함께 현재 OpenGL 시스템에 설정되어 있는 특정 상태의 값을 문의하는데 사용이 된다. 각 함수의 두 번째 인자 params는 해당 타입의 배열에 대한 포인터 값을 가지는데, 첫 번째 인자 pname을 통하여 원하는 상태를 기술하면 두 번째 인자가 가리키는 곳으로 해당하는 설정 값(들)을 돌려준다. 예를 들어 이 프로그램에서와 같이 GL_VIEWPORT를 첫 번째 인자로 사용하면, 현재 뷰폿의 위치 $x$, $y$와 크기 $w$, $h$ 값 네 개가 두 번째 인자가 가리키는 위치에 순서대로 저장이 되어 돌아온다.

**프로그램 예 2.11** 어미소와 송아지의 대화식 회전.

```
void mouse(int button, int state, int x, int y) {
  if (mode.window == AnimateMode) return;
  if ((button == GLUT_LEFT_BUTTON) && (state == GLUT_DOWN)
    pick_cow(x, y); // Line (a)
  if ((button == GLUT_RIGHT_BUTTON) &&
                  (state == GLUT_DOWN)) {
    if (cow.selected == SmallCow) {
      angle_small_cow = ((int) (angle_small_cow + 5.0)) % 360;
      glutPostRedisplay();
    }
  }
}

void pick_cow(int x, int y) {
  GLint viewport[4];
  GLint hits;

  glGetIntegerv(GL_VIEWPORT, viewport); // Line (b)

  glSelectBuffer(256, selectbuffer); // Line (c)
  glRenderMode(GL_SELECT);
  mode.render = GL_SELECT
  glInitNames(); // Line (d)
  glPushName(NoCow);

  glMatrixMode(GL_PROJECTION); // Line (e)
  glPushMatrix(); // Line (f)
  glLoadIdentity();
  gluPickMatrix(x, viewport[3]-y, 5.0, 5.0, viewport);
  glOrtho(-5.5, 5.5, -5.5, 5.5, 5.0, 100.0);

  glMatrixMode(GL_MODELVIEW);
  draw_world(); // Line (g)
  glMatrixMode(GL_PROJECTION);
  glPopMatrix(); // Line (h)
  hits = glRenderMode(GL_RENDER); // Line (i)
```

```
  mode.render = GL_RENDER;
  process_hits(hits); // Line (j)
}

void process_hits(GLint hits) {
  GLuint names, *ptr, cow_i[2], min_i[2], selected_cow;
  int i, j;

  if (hits > 0) {
    printf("**********\nNO. OF HITS = %d \n", hits);
    ptr = selectbuffer;
    for (i =0; i < hits; i++) {
      printf("*** Hit(%d) ***\n", i);
      printf("*** Number of names = %d\n", (names =
*(ptr++)));
      printf("*** Minimum depth = %u\n", (min_i[i] =
*(ptr++)));
      printf("*** Maximum depth = %u\n", *(ptr++))
      // always names = 1 in this example
      cow_i[i] = *(ptr++);
       printf("*** hit cow = ");
       if (cow_i[i] == BigCow) printf("BigCow\n\n");
       else printf("SmallCow\n\n");
      }
      selected_cow = cow_i[0];
      if ((hits == 2) && (min_i[0] > min_i[1]))
      // assume only maximum two hits
       selected_cow = cow_i[1];
      printf("*** Selected object = ");
      if (selected_cow == BigCow) printf("BigCow\n**********\n");
      else printf("SmallCow\n**********\n"); ptr++;
      if (cow.selected == NoCow) {
       cow.selected = selected_cow;
       glutPostRedisplay();
      }
      else if (cow.selected == (int) selected_cow) {
       cow.selected = NoCow;
       glutPostRedisplay();
      }
```

```
  }
}

void init_OpenGL(void) {
      ⋮

  // use the default viewport trans.

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-5.5, 5.5, -5.5, 5.5, 5.0, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

다음 앞의 예제 프로그램에서도 설명한 바와 같이 Line (c)에서 선택 버퍼를 설정한 후, 그 다음 문장에서 선택 모드로 들어간다. Line (d)에서 적절히 이름 스택을 초기화 한 후, Line (e)와 이후 네 개의 문장에서 집어내기를 사용하는데 필요한 행렬 $M_{PM} \cdot M_P$를 투영 행렬 스택의 탑에 올려주고 있다. 이 때 정상적인 렌더링 모드에서 사용하는 변환 $M_P$를 스택에 보존을 해주고(Line (f)), $M_{PM} \cdot M_P$를 탑에 올려 선택 모드에서 렌더링을 한 후, 이를 팝 해주면(Line (h)), OpenGL 함수를 호출하여 매번 $M_P$를 계산할 필요가 없게 된다. 여기서 주의해야 할 점은 `gluPickMatrix(x, viewport[3]-y, 5.0, 5.0, viewport);` 문장에서 사용자가 선택한 지점의 $y$ 좌표를 넘길 때, 윈도우 시스템에서 넘겨준 `y` 좌표 값이 아니라 `viewport[3]-y`, 즉 $h_0 - y$를 넘겨야 한다는 사실이다. 이는 일반적인 윈도우 시스템에서의 2차원 좌표계에서의 $y$축 방향이 OpenGL 윈도우 좌표계에서의 $y$축 방향과 반대이기 때문이다.

이제 Line (g)에서 물체들을 그려주는데, 물론 이 부분에서 각 물체를 그릴 때 적절히 이름 스택의 내용을 조절해주어야 한다. 특히 이 예에서는 `draw_world()` 함수에서 호출하는 `draw_cow(*)`라는 함수에서 어미소와 송아지를 그릴 때 각각 `BigCow`와 `SmallCow` 값을 이름 스택의 탑에 올리게 된다. 다음 Line (i)에서 다시 정상적인 렌더링 모드로 돌아와 Line (j)에서 `process_hits(*)` 함수를 호출하여 선택 버퍼의 내용을 해석한 후 어떤 소가 선택이 되었는지를 인지하게 되는데, 이 함수에 대한 설명은 생략하기로 하겠다.

이제 지금까지의 내용을 요약하여 보자. 집어내기는 그래픽스 응용 프로그램을 제작하는데 있어 중요한 기능 중의 하나로서, 이러한 기능을 구현하기 위해서 개념적으로는 기하 파이프라인에서의 변환을 거슬러 올라가야 한다. 그러나 OpenGL에서는 윈도우 상에서 커서로 집어내기 지점과 그 부근의 영역을 설정하면, 선택 모드에서 물체를 다시 그리는 과정을 반복하여 어떤 물체가 윈도우상의 선택 영역으로 떨어지는지를 결정하는 방식을 취한다. 따라서 OpenGL의 선택 기능은 집어내기 기능을 쉽게 구현할 수 있도록 해주나 상황에 따라 주의를 해야 한다. 문제는 집어내기를 위하여 세상의 물체를 다시 한번 그려야 한다는 점이다. 세상이 복잡하게 모델링이 되어 있다면 이를 위하여 상당한 계산을 하여야 하기 때문에 프로그램이 비효율적이 될 수가 있다. 따라서 이 기능을 사용하기 전에 물체를 집어낼 수 있는 다른 방법이 있는지를 살펴봐야 할 것이다. 예를 들어 VLSI 회로 설계 프로그램 같으면 바운딩 박스 등을 사용하여 단순히 물체를 다시 그리는 것보다 더 효율적으로 집어내기 기능을 구현할 수 있을 것이다.

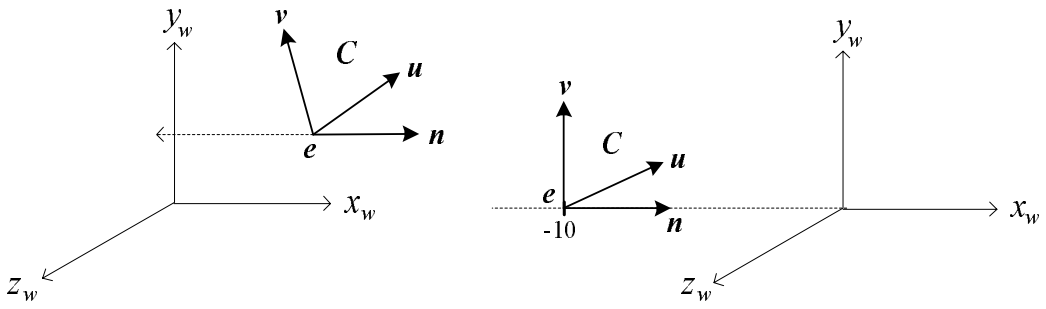
OpenGL의 선택 기능에 대하여 한 가지 더 이야기하면 여기서도 이름 스택을 제공하는데, 이는 계층적 구조를 가지는 물체를 구성하는 부품을 집어내는데 유용하게 이용을 할 수 있다. 물체를 그리기 위하여 모델뷰 행렬 스택을 조작할 때 그에

대응되는 방식으로 이름 스택에 적절한 정보를 올리면 효과적인 집어내기를 구현할 수 있을 것이다. 연습 문제로 앞에서 설명한 계층적 구조를 가지는 자동차 바퀴의 나사를 선택하였을 때, 어떤 바퀴의 몇 번째 나사가 선택되는지를 출력하는 프로그램을 작성하여 보기 바란다. OpenGL에서 교차 정보를 알려줄 때 물체가 선택될 당시의 이름 스택의 전체 내용을 알려주므로, 이름 스택에는 물체의 계층 구조 트리의 루트에서 현재 그리는 부품까지의 경로에 관한 정보를 올려주면 된다. 이러한 기능은 3차원 물체 설계 소프트웨어의 가장 기본적인 기능 중의 하나라 할 수 있다.

## 4.4 카메라의 움직임

애니메이션이나 대화식 소프트웨어를 제작하는데 있어 중요한 요소 중의 하나는 물체뿐만 아니라, 카메라를 어떻게 하면 자연스럽게, 그리고 의도하는 대로 조절을 할 수 있는가 하는 것이다. 드라마나 영화를 보면 카메라의 움직임이 영상의 표현에 있어 얼마나 큰 영향을 미치는지를 알 수가 있다. 기하 파이프라인에서 카메라의 움직임과 직접적으로 관련된 변환은 뷰잉 변환이다. 앞에서는 설명의 편의상 gluLookAt(*) 함수를 사용하여 뷰잉 변환을 하였는데, 일반적으로 뷰잉 변환은 강체 변환이므로 이동 변환과 회전 변환의 곱 $M_V = R \cdot T$의 형태로 표현할 수 있다. 오일러 각도(Euler angle)나 쿼터니온(quaternion)과 같은 개념을 사용하면 카메라의 움직임을 효과적으로 표현할 수가 있지만, 이들에 대한 내용은 이 책의 범위를 넘어서므로 여기서는 직관적으로 가장 단순한(사실은 오일러 각도와 밀접한 관련이 있는) 방법을 사용하여 카메라의 조작에 대하여 설명을 하겠다.

3차원 공간인 세상 좌표계에서 카메라의 위치와 방향은 한 점 $\mathbf{e} = (e_x, e_y, e_z)$와 서로 수직인 세 개의 단위 벡터 $\mathbf{u} = (u_x, u_y, u_z)$, $\mathbf{v} = (v_x, v_y, v_z)$, 그리고 $\mathbf{n} =$




(a) 카메라의 설정 (b) 카메라의 변환

그림 2.64: 카메라의 설정 및 변환

$(n_x, n_y, n_z)$를 사용하여 표현할 수 있다. 즉 카메라를 $C = (\mathbf{e}, \mathbf{u}, \mathbf{v}, \mathbf{n})$과 같이 나타낼 수 있는데(그림 2.64(a)), 여기서는 편의상 3.5절에서 gluLookAt(*) 함수에 대하여 설명할 때 사용했던 것과 같은 기호를 사용하겠다. 따라서 **e**는 카메라의 위치를 나타내고, **u**는 카메라에서 세상을 바라볼 때 오른쪽 방향, **v**는 위쪽 방향, 그리고 **n**은 시선의 반대 방향을 가리킨다. 행렬 $R$을 3.7.2절에서 정의한 바와 같이 왼쪽 위의 3행 3열 부행렬의 각 행이 **u**, **v**, **n**의 $x_w$, $y_w$, $z_w$ 좌표 값을 가지는 행렬이라 하면, OpenGL 함수를 사용한 일반적인 뷰잉 관련 코드는 다음과 같다.

```
/* Assign values to GLfloat R[16].  */
glMultMatrixf(R);
glTranslatef(-ex, -ey, -ez);
```

카메라도 3차원 공간에서 움직이는 하나의 물체이므로, 다른 기하 물체들과 같이 카메라 $C = (\mathbf{e}, \mathbf{u}, \mathbf{v}, \mathbf{n})$에 대한 프레임에 대해서도 기하 변환을 수행할 수 있다. 편의상 초기 상태의 카메라의 설정은 $C_0 = ((0\ 0\ 0), (1\ 0\ 0), (0\ 1\ 0), (0\ 0\ 1))$과 같다고 가정하자. 즉 초기에는 카메라의 중심이 세상 좌표계의 원점에 위치하고 있고, **u**, **v**, **n** 벡터는 각각 $x_w$, $y_w$, $z_w$축과 일치되어 있다. 이 때 그림 2.64(b)에서와 같이 이 카메라를 $y_w$축 둘레로 90도만큼 회전한 후, $x_w$축 방향으로 -10만큼 움직여 $C_1 = ((-10\ 0\ 0), (0\ 0\ -1), (0\ 1\ 0), (1\ 0\ 0))$ 상태로 옮겼다고 가정하자. 이는 카메라

라는 물체에 대하여 $R(90,0,1,0)$에 해당하는 변환을 한 후, $T(-10,0,0)$에 해당하는 변환을 한 셈이다. 이러한 방식의 카메라에 대한 변환과 뷰잉 변환, 그리고 이를 구현하는 OpenGL 프로그램과는 어떠한 관계가 있을까? 잠시 생각을 해보면 $C_1$에 대한 뷰잉 변환은 $M_V = R(-90,0,1,0)\cdot T(10,0,0) = R^{-1}(90,0,1,0)\cdot T^{-1}(-10,0,0)$과 같고, 이를 구현하기 위한 OpenGL 코드는 다음과 같다.

```
glRotatef(-90.0, 0.0, 1.0, 0.0);
glTranlatef(10.0, 0.0, 0.0);
```

여기서 카메라에 대한 변환과 뷰잉 변환에 대한 OpenGL 코드를 비교해보면 상당히 혼란스러울 수도 있다. OpenGL 프로그램에서 함수를 부르는 순서(회전 $\rightarrow$ 이동)와 카메라에 대한 변환 순서는 일치하나 대응되는 각 변환은 서로 역변환으로 되어 있다. 앞에서도 설명을 한 바와 같이 뷰잉 변환과 관련한 OpenGL 프로그래밍의 목적은 뷰잉 변환 행렬을 계산하여 모델뷰 행렬 스택의 탑에 올려놓는 것이다. 뷰잉 변환은 세상 좌표계를 기준으로 하여 표현된 기하 물체의 좌표를 눈 좌표계, 즉 카메라의 프레임 $C$를 기준으로 하여 표현해주기 위한 변환인데, 이는 임의의 $C$를 초기 상태의 $C_0$과 일치하게 해주는 변환에 해당한다. 카메라에 대한 변환을 세상 좌표계의 물체 입장에서 보면 뷰잉 변환을 하기 위하여 그의 역변환을 수행하여야 한다. 카메라에 대하여 $M_1$이라는 변환을 하였다면, 뷰잉 변환을 위해서 상대적으로 물체에 대하여 $M_1^{-1}$에 해당하는 변환을 해주어야 한다. 만약 카메라에 대하여 $M_1$이라는 변환을 한 후 다시 $M_2$라는 변환을 하였다면, 뷰잉 변환은 이를 거꾸로 돌리는 것이므로, 기하 물체에 대하여 $M_2^{-1}$에 해당하는 변환을 한 후 $M_1^{-1}$에 대한 변환을 해주면 된다.

카메라를 $y_w$축으로 90도 만큼 회전한 후 $x_w$축으로 -10만큼 옮겼다면, 뷰잉 변환을 위해서는 물체를 $x_w$축으로 10만큼 이동한 후 $y_w$축 둘레로 -90 회전하면 된다. 따라서 결론은 1. $C_0$ 상태에서 시작하여 카메라에 대하여 $M_1 \rightarrow M_2 \rightarrow \cdots \rightarrow M_n$ 순

서로 변환을 하였다면, 2. 뷰잉 변환은 $M_V = M_1^{-1} \cdot M_2^{-1} \cdot \cdots \cdot M_n^{-1}$이 되고, 3. 이를 구현하는 OpenGL 프로그램은 다음과 같은 형태를 갖게 된다.

```
glMultMatrixf(M1^-1);
glMultMatrixf(M2^-1);

glMultMatrixf(Mn^-1);
```

따라서 OpenGL에서는 카메라에 대한 변환을 같은 순서로 각 변환의 역변환에 해당하는 함수를 부르면 되는데, 단순히 이러한 사실을 외우기보다는 그 원리를 정확하게 이해하고 적용하기 바란다.

카메라는 우리가 세상을 바라보는 창문이라 생각하면 된다. 마우스나 조이 스틱과 같은 입력 장치를 사용하는 3차원 비행 시뮬레이션 소프트웨어에서 비행기를 조종한다고 할 때, 비행기, 즉 카메라의 움직임은 크게 두 가지로 나누어진다. 하나는 카메라의 이동으로서, 이는 3차원 공간에서의 **e**값의 이동 변환에 해당한다. 다음은 카메라의 회전으로서 임의의 회전은 세 개의 기본 축을 기준으로 하여 분해를 할 수 있다. 참고로 **u**, **v**, **n**축 방향으로의 이동을 돌리(dolly), 붐(boom), 트럭(truck)이라 하고, 각 축을 둘레로 하는 회전을 틸트(tile) 또는 피치(pitch), 팬(pan) 또는 요(yaw), 롤(roll)이라 한다. 문제는 사용하는 입력 장치로부터의 신호를 잘 해석하여 이동의 양과 각 회전축에 대한 회전 각도를 효과적으로 계산을 한 후, 그에 맞게 카메라의 상태를 수정해주어야 한다.

아래의 프로그램 예 2.12에서는 문제를 간단하게 하기 위하여 $x_w$와 $z_w$축 방향으로의 이동, 즉 수평 방향으로 이동과, $y_w$축 둘레로의 회전, 즉 **v**축 둘레로의 회전만 가능한 카메라의 움직임에 대한 프로그램을 이해하여보자(예제 프로그램 2.G참조). 그림 2.65에는 탱크 게임 부류의 프로그램의 한 장면이 도시되어 있다. 초기에는 가운데 아래쪽의 뷰폿에 회전하는 소와 육면체를 옆에서 바라본 모습이 나타난

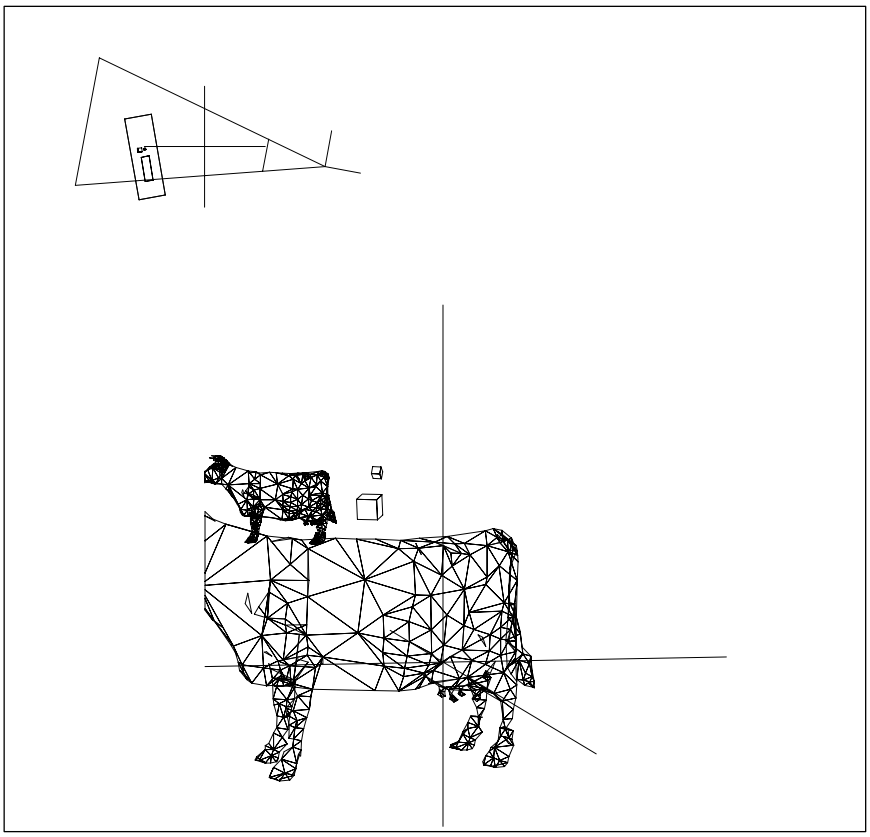


그림 2.65: 카메라의 움직임에 대한 한 장면

다. 다음 왼쪽 위의 또 하나의 뷰폿에는 세상을 위에서 바라본 모습이 나타나 있는데, 편의상 이 뷰폿을 레이더라고 부르자. 이제 왼쪽 버튼을 누른 상태에서 마우스를 움직여 보자. 우선 마우스를 앞으로 움직이면 카메라가 화면 안쪽으로 들어가는 효과가 있고, 뒤로 움직이면 그 반대 방향으로 움직이게 될 것이다. 반면 마우스를 좌우로 움직이면 카메라가 해당 방향으로 $y_w$축 둘레로 회전을 하게 될 것이다. 레이더에는 이러한 카메라의 움직임을 카메라의 뷰잉 볼륨을 통하여 표시하게 된다.

이 프로그램에서 카메라는 `Cam`이라는 스트럭춰 타입으로 선언이 되어 있는데, 여기에는 이 프로그램에서 필요한 카메라의 모든 성질에 대한 변수들을 모아 놓았다. 각 변수의 의미는 분명한데 여기서 배열 `mat`는 앞에서 설명한 $R$ 행렬의 값을 저장하게 된다. `draw_scene()` 함수에서 보듯이 항상 뷰잉 변환은 glMultMatrix(*) 함수

와 glTranslatef(*) 함수에 의하여 수행이 된다(Line (a)와 그 다음 문장). 이 프로그램의 핵심은 2차원 평면 상에서 이동하는 마우스의 움직임으로부터, 그에 해당하는 카메라의 이동 변환과 회전 변환에 대한 인자 값을 찾아내는 것이다. 즉 마우스의 움직임에 대한 이벤트 메시지가 들어 왔을 때, 마우스의 움직임을 각각 $x_w$와 $y_w$축 방향의 움직임으로 분해를 한 후, 그에 맞게 카메라의 프레임에 대한 인자들을 적절히 수정을 해주어야 한다. 이 프로그램에서 움직임 컬백 함수로 등록된 `motion(*)` 함수는 왼쪽 버튼을 누른 상태에서 마우스를 움직일 때마다, 각각 $x_w$와 $y_w$축 방향으로 움직인 양 `delx`와 `dely`를 계산하여, 전자는 프레임의 방향을 수정하는데 사용하고(Line (c)), 후자는 카메라의 위치를 바꾸어 주는데 사용을 하게 된다(Line (b)).

`renew_cam_pos(*)` 함수에서는 마우스를 위 아래로 움직인 양에 비례하여 카메라의 위치를 **n** 방향을 따라 움직여 준다. 반면 `renew_cam_ori_y(*)` 함수는 우선 마우스를 좌우로 움직인 양에 비례하는 각도만큼 **v**축으로 회전을 시켜주는 행렬 `m`을 계산을 한다(Line (e)). 다음 이 회전 행렬을 사용하여 두 배열 `cam.uaxis`(**u**축 방향), `cam.naxis`(**n**축 방향)의 값을 수정한 후, Line (f)에서 `set_rotate_mat(*)` 함수를 수행 시켜 `cam.mat`의 내용을 변경해 준다. 이렇게 새로운 카메라의 성질을 설정한 후 다시 그림을 그려주면(Line (d)), 마우스의 움직임을 반영한 새로운 카메라의 상태에서 바라본 세상이 렌더링이 되는 것이다. 이 예제의 함수 `get_rotation_mat(*)`는 임의의 방향 $(x\ y\ z)^t$을 가지는 직선 둘레로 주어진 각도만큼 회전을 해주는 행렬을 계산하는데, 89쪽의 2.5.2절에서 유도한 방식으로 계산을 하였다. 참고로 이 프로그램에서는 임의의 방향이 아니라 항상 $(0\ 1\ 0)^t$ 방향 둘레로만 회전을 하므로, 이보다 훨씬 간단하게 이 함수의 코드를 수정할 수가 있다.

**프로그램 예 2.12** 카메라의 움직임.

```
typedef struct _cam {
```

```
  float pos[3]; // e
  float uaxis[3], vaxis[3], naxis[3]; // u, v, n
  GLfloat mat[16]; // R matrix
  int move; // moving state
  GLdouble fovy, aspect, near_c, far_c; // camera properties
} Cam;
Cam cam;
      ⋮
void draw_scene(void) {
      ⋮
  gluPerspective(cam.fovy, cam.aspect, cam.near_c,
cam.far_c);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glMultMatrixf(cam.mat); // Line (a)
  glTranslatef(-cam.pos[X], -cam.pos[Y], -cam.pos[Z]);
      ⋮
}
      ⋮
void motion(int x, int y) {
  int delx, dely;
  if (!cam.move) return;
  delx = prevx - x; dely = prevy - y;
  prevy = y; prevx = x;

  renew_cam_pos(dely); // Line (b)
  renew_cam_ori_y(delx); // Line (c)
  glutPostRedisplay(); // Line (d)
}
      ⋮
#define CAM_TSPEED 0.05
void renew_cam_pos(int del) {
  cam.pos[X] += CAM_TSPEED*del*(-cam.naxis[X]);
  cam.pos[Y] += CAM_TSPEED*del*(-cam.naxis[Y]);
  cam.pos[Z] += CAM_TSPEED*del*(-cam.naxis[Z]);
}
```

```
#define CAM_RSPEED 0.1
void renew_cam_ori_y(int angle) {
  float m[3][3], tmpX, tmpY, tmpZ;

  get_rotation_mat(cam.vaxis[X], cam.vaxis[Y], cam.vaxis[Z],
                     CAM_RSPEED*angle, m); // Line (e)

  cam.uaxis[X] = m[0][0]*(tmpX=cam.uaxis[X]) +
     m[0][1]*(tmpY=cam.uaxis[Y]) + m[0][2]*(tmpZ=cam.uaxis[Z]);
  cam.uaxis[Y] = m[1][0]*tmpX + m[1][1]*tmpY + m[1][2]*tmpZ;
       ⋮       // update u and n

  set_rotate_mat(cam.mat); // Line (f)
}
```

## 4.5 비아핀 모델링 변환

지금까지 살펴본 기하 변환의 예에서는 모두 이동 변환, 크기 변환, 회전 변환 등 OpenGL에서 기본적으로 제공하는 함수들을 사용하여 모델링 변환을 하였는데, 이러한 기본 변환 외에도 glMultMatrix(*) 함수를 사용하여 다른 형태의 변환을 수행할 수가 있다. 예를 들어 쉬어링 변환을 하고자 하면 그에 해당 행렬을 계산하여 이 함수를 통하여 스택에 올릴 수가 있다. 또한 수행하려는 변환이 여러 개의 기본 변환으로 복잡하게 구성이 되어 있다면, 매 장면마다 각 기본 변환에 대한 OpenGL 함수를 부르는 것보다 미리 그 변환을 합성하여 놓고 glMultMatrix(*) 함수로 그 행렬을 올리는 것이 더 효율적일 것이다. 또한 일반적으로 모델링 변환과 뷰잉 변환에 대해서는 아핀 변환을 사용하지만 필요에 따라서는 비아핀 변환을 사용할 수도 있다. 이 절에서는 간단하게 비아핀 모델링 변환을 사용하는 예에 대하여 알아보자(예제 프로그램 2.H참조).

146쪽의 그림 2.31을 다시 보자. 이 그림은 3절에서 OpenGL 기하 파이프라인의 과정을 설명하기 위하여 예로 든 168쪽의 프로그램 예 2.4를 수행할 때 거쳐가는 정규 디바이스 좌표계의 모습을 보여주고 있다. 물론 그 프로그램이 수행이 될 때는 이러한 장면이 도시되지는 않는다. 그러면 이 그림은 어떻게 렌더링이 되었을까? 예를 들어 그림 2.31(b)는 원근 변환을 사용하는 경우인데, 이 그림을 그리기 위해서는 앞에서 살펴본 다른 예들과는 다른 형태의 모델링 변환이 요구된다. 우선 중요한 사실은 프로그램 예 2.4에서의 정규 디바이스 좌표계가 현재 작성하려하는 예제 프로그램에서의 세상 좌표계가 된다는 것이다. 따라서 이 절에서 작성하려는 프로그램에서의 모델링 변환은 프로그램 예 2.4에서의 모델링 변환, 뷰잉 변환, 투영 변환을 모두 합성한 변환에 해당이 된다.

여기서 한 가지 주의를 기울여야 할 것은 앞의 프로그램에서 기하 물체를 그리는 과정 중 절단 좌표계에서 뷰잉 볼륨에 대하여 절단이 일어나는데, 만약 현재 프로그램에서 위의 세 가지 변환을 합성하여 모델링 변환으로 사용한다면 그러한 절단 과정이 생략이 될 것이다. 따라서 이에 대한 적절한 처리를 해주지 않으면 그림 2.31(b)에서 정규 디바이스 좌표계의 정육면체에 대한 절단이 되지 않고 모든 물체가 그려지게 될 것이다. 그러면 이러한 문제는 어떻게 해결할 수 있을까? 절단은 뷰잉 볼륨이 주어졌을 때, OpenGL에서는 절단 좌표계에서 절단이 일어난다고 했다. 이는 OpenGL 시스템을 설계할 때 렌더링 계산의 효율을 극대화하기 위하여 그렇게 결정을 한 것인데, 개념적으로는 눈 좌표계에서의 뷰잉 볼륨, 절단 좌표계에서의 대응되는 뷰잉 볼륨, 그리고 정규 디바이스 좌표계에서의 정육면체 영역 등 어떤 것에 대하여 절단을 하건 그 결과는 같게 될 것이다. 따라서 이 절의 예제 프로그램의 세상 좌표계에서의 $[-1,1]\times[-1,1]\times[-1,1]$ 영역에 대하여 사용자 설정 절단 평면으로 절단을 해주면(3.17.2절 참조), 이는 프로그램 예 2.4에서의 절단 좌표

계에서의 절단 효과를 내줄 것이다.

아래의 프로그램 예 2.13에 주어진 함수를 잠깐 살펴보자. 여기서 `render()` 함수에서 이 프로그램에 필요한 뷰폿 변환(Line (a)), 투영 변환(Line (b)), 뷰잉 변환(Line (c))을 설정하기 위하여 OpenGL 함수를 호출하고 있다. 이제 Line (c) 이후부터의 좌표는 세상 좌표계에서의 의미를 갖으므로, 세상 좌표계 공간에서의 좌표축(Line (d))과 프로그램 예 2.4의 정규 디바이스 좌표계에서의 정육면체 영역에 해당하는 육면체를 세상에 그려주고 있다(Line (e)). 다음 기하 물체들을 그리기 전에 절단 효과를 내기 위하여 Line (f)의 `cutting_plane()` 함수를 호출하여 세상 좌표계를 기준으로 한 여섯 개의 절단 평면을 설정하고 있다.

**프로그램 예 2.13** 비아핀 모델링 변환의 예.

```
void render(void) {
  glClear(GL_COLOR_BUFFER_BIT);
  glViewport(0.0, 0.0, 600.0, 600.0); // Line (a)
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1.5, 1.5, -1.5, 1.5, 1.0, 100.0); // Line (b)
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  // Line (c)
  gluLookAt(2.0, 4.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  draw_axes(); // Line (d)
  draw_box(); // Line (e)
  cutting_plane(); // Line (f)
  draw_world(); // Line (g)
  glFlush();
}
      ⋮
void draw_world(void) {
  gluPerspective(28.0, 800.0/600.0, 5.0, 19.0); // Line (h)
  // Line (i)
  gluLookAt(2.0, 8.0, 8.0, 0.0, -4.0, 0.0, 0.0, 0.0, 2.0);
```

```
  glPushMatrix();
    glTranslatef(0.0, 0.0, 1.5);
    glColor3f(1.0, 1.0, 0.0);
    draw_object(teapot, npolytp); // draw teapot
  glPopMatrix();
  glPushMatrix();
    glTranslatef(-6.0, -5.0, 1.0);
    glScalef(2.0, 2.0, 2.0);
    glColor3d(1.0, 0.0, 1.0);
    draw_object(donut, npolydn); // draw bdonut
  glPopMatrix();
  glPushMatrix();
    glTranslatef(0.0, -6.0, 0.0);
    glRotatef(-45.0, 0.0, 1.0, 0.0);
    glScalef(1.0, 1.0, -1.0);
    glRotatef(45.0, 0.0, 1.0, 0.0);
    glColor3d(0.0, 1.0, 1.0);
    draw_object(donut, npolydn); // draw sdonut
  glPopMatrix();
}
```

이제 Line (g)에서 `draw_world()` 함수를 불러 적절한 모델링 변환을 한 후 세 개의 물체를 그려주어야 한다. 앞에서 설명한 바와 같이 이 함수에서의 모델링 변환을 설정하기 위하여, 프로그램 예 2.4에서 사용한 투영 변환(Line (h)), 뷰잉 변환(Line (i)), 모델링 변환(Line (i) 이후의 기하 변환)을 적절한 순서대로 모델뷰 행렬 스택에 올려주고 있음을 알 수가 있다. 여기서 한 가지 언급할 것은 이 프로그램에서 사용한 모델링 변환은 더 이상 아핀 변환이 아니라는 사실이다. Line (h)의 `gluPerspective()` 함수를 통하여 모델뷰 행렬 스택의 탑에 곱해지는 행렬은 아핀 변환이 아닌 원근 투영 변환이므로 전체 모델링 변환은 비아핀 변환이 되게 된다. 따라서 이 모델링 및 뷰잉 변환에 의하여 각 기하 물체들이 눈 좌표계로 변환이 되었을 때, 좌표 값들의 $w$ 좌표가 1이 아닐 수도 있게 된다. 그러나 계속되는 투

영 변환의 결과 눈 좌표계에서 정규 디바이스 좌표계로 가는 과정에서 원근 나눗셈을 통하여 $w$ 값이 다시 1로 돌아오게 되므로, 이 경우에서처럼 모델링 변환이 비아핀 변환이어도 아무런 문제가 없다. 이 예에서는 비아핀 형태의 변환을 통한 모델링 변환에 대하여 알아보았는데, 그 외에도 뷰잉 변환이나 투영 변환의 경우에서도 OpenGL에서 제안하는 일반적인 방법 외의 다른 형태의 변환을 사용함으로써 특수한 효과를 낳을 수가 있을 것이다.

# 제 3 장

# OpenGL의 조명 모델

## 제 1 절 라이팅과 쉐이딩

2장에서는 전체 OpenGL 렌더링 과정에서 뷰잉, 즉 가상의 세상에 존재하는 기하 물체들이 카메라에 대한 성질이 결정이 되었을 때, 화면의 어느 지점에 나타날 것인가를 계산하는 과정에 대하여 알아보았다. 아직까지는 다면체 모델로 표현된 물체들의 골격만 화면에 나타나는 상황이기 때문에, 그러한 이미지는 매우 엉성하게 보일 것이다. 이제부터 해야 할 일은 골격만 보이는 기하 물체에 '그럴듯하게 껍질을 입혀' 사실적인 이미지를 생성하도록 하는 것이다. 그러기 위해서는 지금까지의 뷰잉 계산과는 전혀 다른 부류의 계산을 수행하여야 한다. 껍질을 입힌다는 것은 기하 물체가 자연스럽게 보이도록 표면의 색깔을 계산하는 것을 의미한다. 그러면 과연 OpenGL에서는, 아니 3차원 컴퓨터 그래픽스 분야에서는 어떠한 방식으로 그러한 색깔을 계산을 할까?

이 장의 목표중의 하나가 그러한 계산 과정을 이해하는 것인데, 우선 그림 3.1을 보도록 하자. 뷰잉 과정을 통하여 물체 상의 한 점 P가 화면상의 점 $P'$으로 투영이

된다는 사실을 알아내었다면, 그 다음 물체 표면의 색깔을 구하기 위해서 무엇보다도 다음과 같은 사항을 고려해야 한다. 첫째, 과연 물체의 점 P가 P′ 지점의 화소라는 조그마한 창문을 통하여 보일 것인가 하는 것이다. 일반적으로 가상의 세상에는 여러 개의 물체가 존재하기 때문에 한 개, 또는 두 개 이상의 물체 상에 존재하는 다수의 점들이 화면의 같은 점으로 투영이 될 수가 있다. 즉 동일한 투영선에 존재하는 여러 개의 점들이 같은 지점으로 투영이 될 텐데, 이러한 경우 과연 실제로 눈에 보이는 점이 어떤 점인지를 결정해야 한다. 예를 들어 여러 개의 불투명한 물체들이 같은 시선 상에 존재한다면, 화면에서 가장 가까운 점만 보이고, 나머지 점들은 무시되어야 할 것이다. 이렇게 어떤 물체가 보이는지, 아니면 다른 물체에 가려 안 보이는 지를 검사하는 계산 과정을 가시성 검사(visibility test)라 하는데, 그 결과 안 보이는 물체가 제거되기 때문에 은면 제거(hidden surface elimination)라 하기도 한다. 이는 렌더링 과정에서 매우 중요한 부분으로서, 오래 전부터 이에 대하여 다양한 알고리즘이 개발이 되어 왔다. OpenGL에는 은면 제거를 위하여 깊이 버퍼에 기반을 둔 깊이 버퍼링 방법을 사용하는데 이에 대해선 뒤에 다시 설명하도록 하겠다.

다음 둘째로 고려해야할 사항은 만약 점 P가 P′ 지점을 통하여 보이는 점이라면 과연 P′ 지점에 해당하는 화소를 어떤 색깔로 칠할 것인가 하는 것이다. 이를 결정하기 위해서는 가상의 세상에서 빛을 발하는 물체, 즉 광원(light source)에서 출발한 빛이 복잡한 경로를 통하여 물체의 P 지점에 다다른 후, 투영 방향, 즉 시선의 방향으로 어떻게 반사가 되는 지를 계산하여야 한다. 사실 이러한 과정은 물리학적인 문제로서 상당히 복잡한 계산을 수행해야 할 것이다. 대부분의 그래픽스 렌더링 알고리즘들은 색깔 계산을 하는데 있어 그 효율을 위하여 실제의 물리학적인 모델을 단순화시킨 실험적인 모델을 사용한다. 이러한 모델을 그래픽스 분야에서는 조명

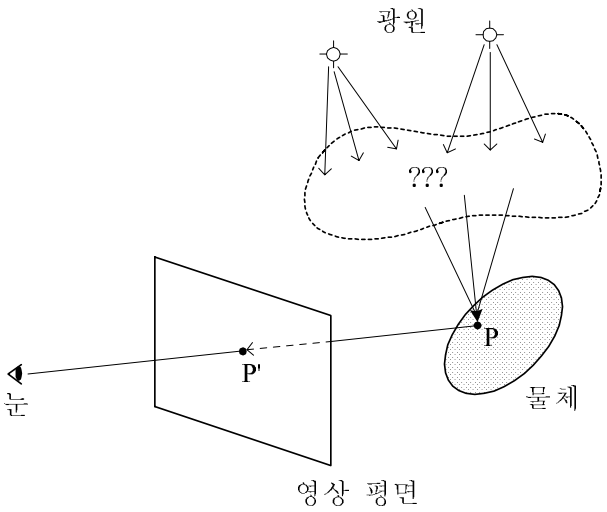



그림 3.1: 조명 모델

모델(illumination model) 또는 라이팅 모델(lighting model)이라 하는데, 그 단순화의 정도에 따라 계산 시간과 생성되는 이미지의 사실성이 달라지게 된다.

그림 3.1에서 물음표가 있는 부분을 고려해보자. 광원에서 출발한 빛이 물체의 P 지점으로 들어오는 경로는 매우 복잡할 것이다. 광원에서 직접 들어 올 수도 있고, 다른 물체에 여러 번 반사되어 들어 올 수도 있을 것이다. 따라서 P 지점으로 들어오는 빛은 광원과 물체의 직접적인 관계뿐만 아니라 주변의 물체나 기타 여러 환경적인 요인에 영향을 받을 것이다. 우리가 사진을 찍을 때는 그러한 상황이 모두 고려되기 때문에 사실적인 이미지를 얻을 수가 있다. 그러나 렌더링을 할 때에는 그러한 상황을 모두 고려하여 계산을 한다는 것은 거의 불가능할 것이다. 그래픽스 분야에서 사용하는 조명 모델 중 가장 단순한 모델은 주변 환경은 무시하고 오로지 광원, 물체, 그리고 시점간의 직접적인 관계만 고려하는데, 이러한 모델을 지역 조명 모델(local illumination model)이라 한다. 이 모델에서는 광원에서 물체로 직접 들어오는 빛만 고려하기 때문에 간접적인 경로를 통하여 들어오는 빛은 무시가 된다. 따라서 이 모델은 원래의 물리학에 기반을 둔 정확한 모델을 매우 단순화 시킨

것으로서, 그 결과 모델의 부정확성에 기인한 오차가 발생하게 된다. 즉 주변 환경에서의 빛의 상호 작용을 전혀 고려하지 않은 이 모델에서의 오차는 시각적인 오차로 나타나고, 이는 지역 조명 모델을 사용할 경우 종종 사실적인 이미지와는 동떨어진 엉성한 이미지를 생성하게 되는 결과를 초래한다.

반면에 렌더링을 하고자 하는 물체 주변의 상호 작용까지 고려하여 색깔을 계산하려하는 모델을 전역 조명 모델(global illumination model)이라 한다. 렌더링 알고리즘 중 광선 추적법(ray tracing)과 래디오시티(radiosity) 방법이 바로 여기에 속한다. 이러한 방법들도 물체의 표면으로 들어오는 빛을 모두 고려하는 것이 아니라 색깔 계산에 있어 영향을 많이 미치는 몇 가지 종류의 입사 광선만 고려를 한다. 따라서 원래의 정확한 모델에 비해서는 단순화되기는 하였으나, 주변의 상황을 어느 정도 고려를 하기 때문에 지역 조명 모델보다 훨씬 사실적인 이미지를 생성한다.

지역 조명 모델의 문제점에도 불구하고 그것이 사용되는 이유는 계산량이 적다는 점이다. 여러 번 강조하였듯이 OpenGL과 같은 실시간 렌더링 시스템을 개발하는데 있어 가장 중요한 척도중의 하나는 이미지를 생성할 때 요구되는 계산량을 최적화하여 제한된 시간 안에 원하는 이미지를 만들 수 있도록 하는 것이다. 특히 지금보다 상대적으로 하드웨어의 성능이 훨씬 뒤떨어졌던 과거에는 설사 사실적인 이미지를 만들지 못하더라도 단순한 지역 조명 모델을 사용하였던 것이다. 따라서 OpenGL 시스템에서도 이러한 모델을 사용하는데, 최근의 CPU의 속도와 그래픽스 가속기의 성능 향상에 힘입어 전역 조명 모델의 효과를 내주는 고급 기법들이 속속 개발되고 있다.

실시간 렌더링을 위한 API인 OpenGL 렌더링 파이프라인에서는 다면체 모델로 표현된 물체 표면의 모든 점에 대하여 동일한 방법을 사용하여 색깔을 계산하지를 않는다. 즉 물체 표면의 특정 지점에 대해서만 비교적 정확한, 따라서 계산량이 많

은, 방법을 사용하여 색깔을 계산하고, 나머지 부분에 대해서는 그러한 결과를 사용하여 빠르게 계산을 하게 된다. OpenGL에서의 물체 표면의 색깔을 계산하는 과정을 이해하기 위해서는 무엇보다도 다음과 같은 세 가지 사항을 이해하여야 한다. 첫째, 물체의 한 지점의 표면 색깔을 현실감 있게 계산하기 위하여 어떠한 방식을 사용할 것인가 하는 것이다. 이는 위에서 언급한 조명 모델과 밀접한 연관이 있다. 다음 둘째는, 주어진 기하 물체의 표면 상의 점들 중 어떤 점들에 대해서만 그러한 방법을 적용할 것인가 이고, 마지막으로 나머지 부분에 대해서는 물체의 표면 색깔을 어떠한 방식으로 계산을 할 것인가 이다.

우선 OpenGL에서는 다른 대부분의 3차원 그래픽스 시스템에서와 같이 지역 조명 모델에 기반을 둔 퐁의 조명 모델(Phong's illumination model)이라는 방법을 사용하여 주어진 지점의 색깔을 계산한다. 이 방법은 물리학적으로 정확한 모델이 아니고 계산 효율을 위하여 단순화시킨 실험적인 방법인데, 그 단순함에도 불구하고 우수한 효과를 내기 때문에 그래픽스 렌더링에서 대표적으로 사용되는 계산 방식이다. 두 번째로 OpenGL에서는 대부분의 실시간 렌더링 API에서와 같이 다면체 모델의 모든 표면에 대하여 퐁의 조명 모델을 적용하는 것이 아니라, 물체를 구성하는 꼭지점에 대해서만 조명 모델을 사용하여 색깔을 계산한다. 이는 퐁의 조명 모델이 비록 물리학적으로 복잡한 모델을 많이 단순화시키기는 했지만, 비교적 적지 않은 양의 부동 소수점 연산을 수행해야하기 때문에 계산의 효율을 위하여 꼭지점에 대해서만 적용을 하는 것이다. 마지막으로 다면체 모델이 윈도우에 투영이 되었을 때, 윈도우 좌표계상에서 다각형의 꼭지점들에 대해서만 색깔이 정의된 상태이기 때문에, 그 내부에 존재하는 화소에 어떠한 색깔을 지정을 할 것인가를 결정하여야 한다. OpenGL에서는 기본적으로 다각형의 내부를 꼭지점 색깔중의 하나를 선택하여 전체를 칠하는 방법과 꼭지점의 색깔들을 선형 보간법을 사용하여 그 내

부를 부드럽게 변하는 색깔로 칠하는 방법 등 두 가지를 제공한다.

OpenGL에서는 색깔 계산 모델을 지칭하기 위하여 조명 모델보다는 라이팅 모델이라는 용어를 사용한다. 반면 라이팅 모델을 사용하여 꼭지점들의 색깔을 계산한 후, 그것들을 사용하여 다각형 내부의 화소에 대한 색깔을 구하는데 사용하는 모델을 쉐이딩 모델(shading model)이라 한다. 여기서의 쉐이딩 모델이란 좀 더 정확하게 기술하면 다면체 모델을 위한 쉐이딩 모델(polygonal shading model 또는 shading model for polygonal models)이 되는데, 이는 라이팅 모델과는 전혀 다른 것으로 4장에서 설명할 래스터화 과정에서 적용이 된다. 컴퓨터 그래픽스 분야에서는 문맥에 따라 조명 모델, 라이팅 모델, 쉐이딩 모델이라는 용어들이 상황에 따라 조금씩 다른 의미로, 혹은 같은 의미로 사용이 된다. 어떤 사람은 넓은 의미의 조명 모델 대신에 물체 표면까지 빛이 어떻게 들어오는가에 관련된 모델을 조명 모델, 그리고 그러한 빛이 물체 표면에서 어떻게 반사가 되는가에 대한 모델을 반사 모델(reflection model)이라 부르기도 한다. 이 책에서는 위에서 설명한 바와 같은 의미로 조명 모델(또는 라이팅 모델)과 쉐이딩 모델이라는 용어를 주로 사용하겠다[1]. 이 장에서는 무엇보다도 퐁의 조명 모델에 대하여 기본적인 사항들을 살펴보고, OpenGL 시스템에서 이를 구현하기 위하여 사용하는 정확한 조명 모델을 이해한 후, 예제 프로그램을 통하여 라이팅에 관련된 프로그래밍 기법을 익도록 하겠다.

## 제 2 절 glBegin(*) 함수와 glEnd() 함수의 수행 과정

이제는 여러분들이 잘 알고 있겠지만 OpenGL을 사용하여 기하 물체를 그리기 위해서는 glBegin(*) 함수와 glEnd() 함수 사이에 기하 물체에 대한 정보를 기술해주어야 한다. 이 두 함수 사이에서 기술되어야 하는 것 중 가장 중요한 것은 물론 물

[1] 이 책에서는 조명 모델과 라이팅 모델을 같은 목적을 가지는 모델로 간주하겠다.

체의 형태에 대한 정보를 제공하는 꼭지점들의 좌표이고, 이와 함께 라이팅 및 텍스춰 매핑 계산을 하는데 필요한 각 꼭지점에서의 관련 속성들도 기술이 되어야 한다. 꼭지점에 관련지어지는(associated) 속성으로 법선 벡터, 텍스춰 좌표, 색깔, 에지 플래그(edge flag) 등이 있는데, 에지 플래그는 이 장에서 설명하려는 라이팅 계산에 직접적인 관련이 없으므로 앞의 세 가지 속성에 대해서만 생각을 하겠다. 이 세 가지 속성들은 glBegin(*) 함수와 glEnd() 함수 안에서 각각 glNormal3*(*) 함수, glTexCoord*(*) 함수, glColor*(*) 함수를 사용하여 지정할 수가 있다. 우선 다음과 같은 코드를 예를 들어 glBegin(*) 함수부터 시작하여 glEnd() 함수로 끝나는 부분이 OpenGL 시스템에서 내부적으로 어떻게 수행이 되는지를 살펴보자.

```
glBegin(GL_POLYGON);
  glColor3f(0.2, 0.3, 0.1);

  glTexCoord2f(0.200690, 0.0);
  glNormal3f(0.716563, 0.669553, 0.195543);
  glVertex3f(5.532551, -19.070305, 156.488525);

  glTexCoord2f(0.187873, 0.0);
  glNormal3f(0.905996, 0.263676, 0.331129);
  glVertex3f(5.202378, -18.438967, 156.514755);

  glTexCoord2f(0.178366, 0.213281);
  glNormal3f(0.241297, 0.133705, -0.961197);
  glVertex3f(5.309465, -17.926474, 156.593536);
        ⋮
glEnd();
```

꼭지점의 속성을 기술하는 함수들은 glBegin(*) 함수와 glEnd() 함수 사이에서 자유롭게 사용할 수 있는데, 이들의 목적은 각 속성의 현재 값을 지정하는 것이다. 그림 3.2를 보면 왼편에(현재 에지 플래그는 제외하고) 현재 법선 벡터(Current

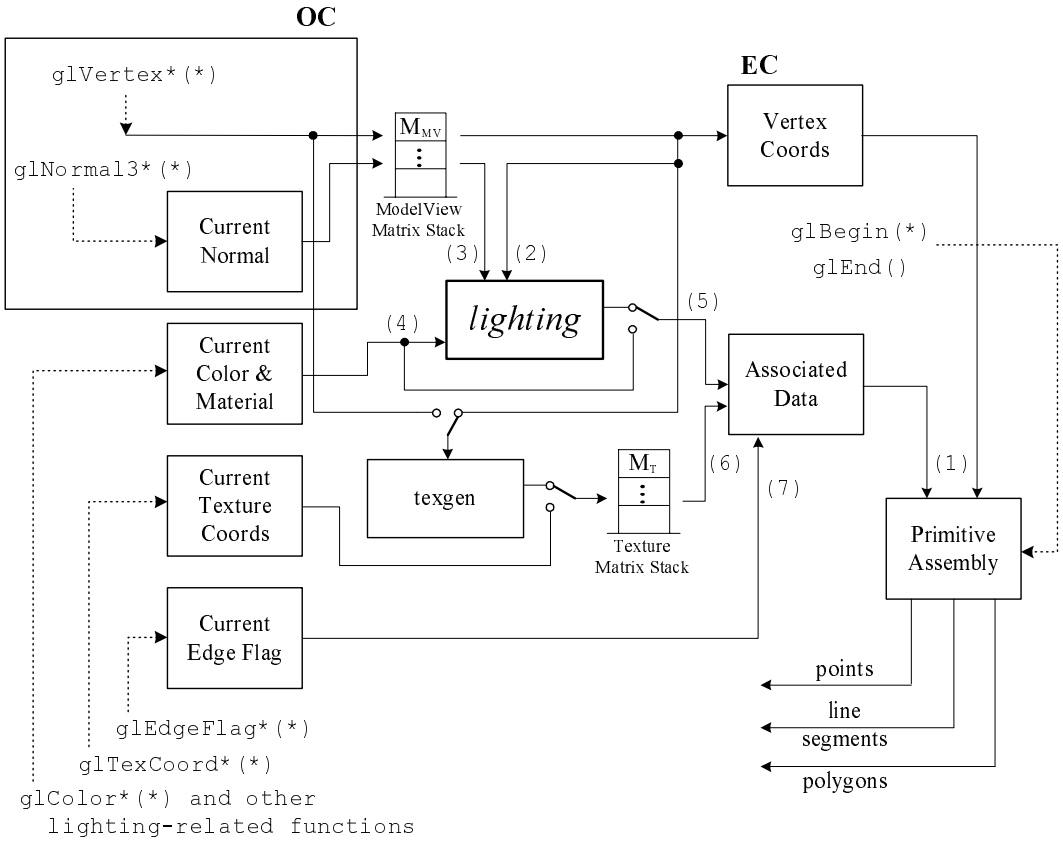



그림 3.2: 프리미티브들의 조합

Normal), 현재 색깔(Current Color), 그리고 현재 텍스춰 좌표(Current Texture Coords)에 해당하는 사각형들이 있는데, 각자 대응 함수에 의해 그 값이 설정이 된다. OpenGL은 상태 머신으로서 이 세 가지 종류의 상태들의 값들이 사용되기 전에 적절히 지정이 되어야 한다. 물론 처음에는 모두 적절한 값으로 초기화가 되어 있다.

항상 이 세 가지 속성에 해당하는 함수는 현재 속성 값을 지정하는 역할만 하고, glVertex*(*) 함수를 수행하면 이 그림에서처럼 꼭지점의 좌표가 렌더링 파이프라인으로 입력되어 흘러가면서 현재 설정이 되어 있는 속성 값들과 함께 적절한 변환을 통하여 프리미티브 조합(Primitive Assembly) 모듈로 흘러가게 된다(그림 3.2의 (1)). glVertex*(*) 함수를 통하여 하나의 꼭지점을 기술하였을 때, 그 좌표가 우선 모델뷰 행렬 스택의 탑에 곱해져 물체 좌표계에서 눈 좌표계로 변환이 된 후 각각

프리미티브 조합 모듈((1))과 라이팅 모듈로 흘러간다((2)). 이와 동시에 현재 법선 벡터로 지정된 값이 역시 눈 좌표계로 변환이 되어(물론 스택의 탑 행렬의 역행렬의 전치 행렬에 곱해져), 그 값이 라이팅 모듈로 흘러가게 된다((3)). 또한 현재 색깔과 관련 인자들도 라이팅 모듈로 흘러가((4)) 라이팅 계산에 영향을 미치게 된다.

꼭지점에 색깔이 관련지어지는 방법은 크게 두 가지로 나누어진다. OpenGL 프로그램에서 라이팅 계산을 하겠다고 선언을 하면, 현재 색깔과 함께 이 그림에는 아직 도시되어 있지 않은 현재 물질 성질(Current Material Properties) 등과 같은 다른 관련 속성 값들을 사용하여 라이팅 계산을 한 후[2], 그 결과 색깔이 관련 데이터(Associated Data) 모듈로 흘러가게 된다((5)). 만약 라이팅 계산을 수행하지 않겠다고 선언을 하면[3], 단순히 현재 색깔이 관련지어진다. 한편 현재 텍스춰 좌표도 5장에서 설명하겠지만 적절한 계산을 수행한 후 관련 데이터 모듈로 가게 된다((6)). 이 모듈에서는 색깔 값(라이팅 계산을 통하여 얻어졌건 현재 색깔을 사용하건)과 텍스춰 좌표, 그리고 현재 에지 플래그 값((7))을 묶어 프리미티브 조합 모듈로 보내게 된다.

이렇게 glBegin(*) 함수와 glEnd() 함수 사이에서 glVertex*(*) 함수를 사용하여 각 꼭지점을 나열할 때, 좌표뿐만 아니라 해당 속성들이 관련지어져 하나씩 프리미티브 조합 모듈로 흘러들어 오지만 이러한 꼭지점들 간에는 아직 아무런 관계가 설정되어 있지 않다. 여기서의 꼭지점간의 관계란 glBegin(*) 함수의 인자에 의해 설정된 프리미티브들의 타입에 따라 결정되는 관계, 즉 꼭지점들이 선분을 구성하는지, 다각형을 구성하는지 등의 관계를 뜻한다. 개념적으로 생각하면 프리미티브 조

[2]라이팅 계산을 할 때는 꼭지점 좌표, 그 점에서의 법선 벡터, 물질의 성질 외에도 여러 인자들이 영향을 미친다. 302쪽의 그림 3.15에는 좀 더 자세한 라이팅 계산 구조가 도시되어 있는데, 여기서는 glBegin(*) 함수와 glEnd() 함수의 수행 과정에 초점을 맞추고, 라이팅 계산에 대해서는 6절에서 자세히 설명을 하도록 하겠다.

[3]사실 이것이 디폴트 상황이므로 라이팅 계산을 하겠다고 명시적으로 선언을 하지 않으면 라이팅 계산이 수행이 안 된다.

합 모듈에서는 계속해서 들어오는 꼭지점 좌표 및 그에 연관된 속성 값들을 저장하고 있다가 glEnd() 함수가 수행이 될 때 glBegin(*) 함수의 인자로 설정한 프리미티브의 타입에 맞게 조합을 해준다. 따라서 이 순간부터는 기하 물체에 대한 데이터가 꼭지점 단위에서 점, 선분, 다각형 등 기하 프리미티브 단위로 바뀌어 흘러가게 되고, 추후 계속 되는 렌더링 파이프라인에서 각 프리미티브 타입에 맞는 렌더링 계산을 수행하게 된다. 지금까지 glBegin(*) 함수와 glEnd() 함수, 그리고 그 안에서 수행이 되는 꼭지점 좌표 및 속성 지정 함수가 OpenGL 시스템 내부에서 어떻게 수행이 될 지를 개념적으로 살펴보았다. 이것을 잘 이해하였다면 이 절에서 주어진 OpenGL 코드를 좀 더 정확한 관점에서 바라보기 바란다.

마지막으로 다음 절로 넘어가기 전에 이 장의 주제인 OpenGL 라이팅에 대하여 몇 가지 간략하게 알아보자. 물체 좌표계에서 설정된 꼭지점과 법선 벡터는 모델뷰 행렬 스택에 의하여 곧바로 눈 좌표계로 변환이 되어 라이팅 모듈로 흘러간다. 따라서 라이팅 계산은 눈 좌표계 공간에서 수행이 됨을 알 수 있다. 조명 계산의 특징에 대하여 몇 가지 나열하면,

1. 지역 조명 모델이 사용된다.

2. 라이팅 계산은 다면체 모델의 꼭지점에 대해서만 적용이 된다.

3. 주어진 기하 정보는 눈 좌표계로 변환이 된 후 이 좌표계에서 라이팅 모델을 적용하여 색깔을 계산한다.

4. 라이팅 계산을 할 경우 그 결과로 기본 색깔(primary color)과 보조 색깔(secondary color)이 계산이 된다.

5. 이렇게 계산이 되는 색깔을 쉐이딩이 된 색깔(shaded color)라고 부를 것인데, 다른 속성들과 함께 해당 꼭지점 좌표에 관련지어져 계속하여 렌더링 파이프

라인으로 흘러간다.

## 제 3 절 RGBA 색깔 모델과 이미지 합성

OpenGL 라이팅 모델에 대하여 구체적으로 살펴보기 전에 대부분의 그래픽스 시스템에서 사용되는 모델인 RGBA 색깔 모델(RGBA color model)에 대하여 살펴보자. 1장에서는 RGB 모델이 컴퓨터 그래픽스 분야에서 가장 기본적으로 사용되는 색깔 모델이라고 하였는데, 실제로 OpenGL을 비롯하여 다른 고급 렌더링 시스템에서는 RGBA 색깔 모델을 기본 모델로 사용한다. 이 모델에는 기존의 R, G, B 채널 외에 A라는 채널이 하나 더 추가가 되는데, 이 채널을 종종 알파 채널(alpha channel)이라 부른다. OpenGL에서는 glColor4*(*) 함수를 통하여 $(R, G, B, A)$ 형태의 색깔을 지정할 수가 있는데, 지금까지 살펴본 예제 프로그램에서 화면의 배경 색깔을 지정하기 위한 함수 glClearColor(*)가 바로 이런 형태의 색깔을 인자로 받아들임을 보았을 것이다.

단순한 렌더링을 한다면 대개의 경우 알파 채널을 사용할 필요가 없지만, 보다 사실적인 이미지를 생성하려 한다면 알파 채널은 종종 중요한 역할을 하게 된다. OpenGL에서 알파 채널을 사용하면 두 개 이상의 색깔을 다양한 방식으로 혼합(blending)하여 여러 가지 효과를 낼 수 있다. 예를 들어 투명한 물체를 렌더링하거나, 주어진 물체에 자연스럽게 텍스춰를 붙여 주려면 알파 채널을 사용하여 색깔을 적절히 혼합을 해주어야 한다. 이 외에도 알파 채널을 통하여 앤티앨리어싱(antialiasing)이나 안개와 같은 환경 효과(atmospheric effect)를 낼 수 있으며, 또한 알파 테스트(alpha test)와 같은 프래그먼트별 연산을 할 수가 있다.

알파 채널을 사용한 이미지 합성에 대한 개념은 1984년도 ACM SIGGRAPH 학회에서 발표된 논문 “Compositing Digital Images”에서 처음 소개가 되었는데, 그후

이 아이디어는 컴퓨터 그래픽스 렌더링 분야에 커다란 영향을 미치었다[4]. RGBA 모델을 사용하면 래스터 이미지의 각 화소는 $(R, G, B, \alpha)$ 값을 가진다. 이 절에서는 알파 채널을 나타내기 위하여 $A$ 대신에 $\alpha$라는 기호를 사용하려 하는데, 여기서 $R, G, B$는 이전과 같이 그 화소에 칠할 색깔을 나타내고, $\alpha$ 값은 그 화소에 대한 어떤 성질을 나타내는 값으로서, 응용하려는 문제에 따라 그 값의 의미가 조금씩 다르게 사용이 된다.

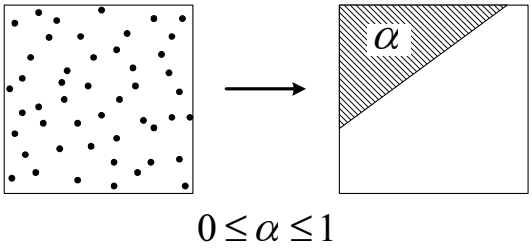



그림 3.3: 알파 값의 의미

일반적으로 알파 값은 그 화소의 불투명도(opacity)를 의미한다고 생각하면 편리하다. 이를 쉽게 이해하기 위하여 래스터 이미지라는 창문을 통하여 세상을 들여다보고 있다고 생각해보자. 이 창문은 화소라는 조그마한 사각형 단위로 나누어져 있는데, 각 화소의 색깔들이 모여 전체 이미지를 구성하게 된다. 한 화소에 대한 색깔이 $(R, G, B, \alpha)$라 할 때, 알파 채널의 값, 즉 알파 값의 의미를 그림 3.3을 통하여 이해해보자. 왼쪽 사각형은 한 화소에 해당하는 영역을 확대한 것인데, 이 때 $(R, G, B)$의 색깔을 가지는 미립자들이 랜덤하게 분포되어 있다고 가정하자. 이 조그마한 알갱이들은 화소 뒤로부터 들어오는 빛을 가로막게 되는데, 화소 안의 임의의 지점을 선택하였을 때, 그 지점을 통해 들어오는 빛이 이러한 미립자에 의해 가로막힐 확률이 $1 - \alpha$ $(0 \leq \alpha \leq 1)$라고 하자. 이 경우 화소 뒤에서 들어오는 빛이 아니라 $(R, G, B)$ 색깔의 알갱이들이 보이게 될 확률은 $\alpha$가 되는데, 이것이 알파 값에 대한 가장 직관적인 해석이 아닐까 한다.

화소의 각 변의 길이가 1이라고 하면(즉 화소의 면적이 1이라 하면), 이 미립자들의 면적을 다 더할 경우 $\alpha$가 되는데, 바로 이 면적만큼만 $(R, G, B)$ 색깔이 나타

[4] T. Porter and T. Duff. "Compositing Digital Images," *ACM SIGGRAPH* '84, pp. 253-259, 1984.

나게 된다. 만약 알파 값이 1이라면 화소 전체가 그 색깔로 보이게 될 것이고, 만약 0.3이라면 그 화소에 대해서는 30%만 그 색깔로 보이고 나머지 부분은 아직 정해지지 않은 상태라 할 수가 있다. 또한 그 값이 0이라면 $R, G, B$ 내용에 상관없이 완전히 투명한 색깔을 나타내게 된다. 이런 관점에서 알파 값은 그 화소의 불투명도를 나타낸다고 할 수가 있는 것이다. 이러한 알파 채널 값을 나타내기 위하여 불투명도라는 용어 외에도 응용 문제에 따라 매트 정보(matte information), 포함 정보(coverage information), 혼합 인자(mixing factor)와 같은 용어들이 사용되는데, 그 기본적인 의미는 동일하다고 할 수가 있다. 다시 요약하면 $(R, G, B, \alpha)$는 $\alpha$의 비율만큼만 $(R, G, B)$ 색깔이 나타나는 색깔이라 할 수 있는데, 이 책에서는 $\alpha$값에 대하여 불투명도라는 용어를 사용하도록 하겠다. 또한 $(R, G, B, \alpha)$의 색깔을 가지는 화소를 편의상 그림 3.3의 오른쪽 사각형과 같이 면적이 $\alpha$이고 색깔이 $(R, G, B)$인 삼각형을 사용하여 나타내도록 하겠다.

지금까지 RGBA 모델로 표현된 색깔의 의미에 대하여 간략히 살펴 보았는데, 알파 채널을 사용한 이미지의 합성(image composition) 방법에 대하여 알아보기 전에 합성의 상황에 대한 간단한 예에 대하여 생각해보자. 어떤 두 이미지를 합성한다는 것은 결국 두 이미지의 대응되는 화소끼리 합성하는 것이 되므로, 여기서는 두 화소간의 합성에 대하여 생각하겠다[5]. 지금 두 개의 이미지 S와 D가 있는데 D는 카메라로 실사 촬영한 이미지이고, S는 렌더링을 통하여 생성한 자동차에 대한 이미지이다[6]. 지금 S 이미지에서 자동차 부분만 뽑아내어 실사 이미지에 합성을 하려 한다. 이 경우 D 이미지는 배경에 해당하기 때문에 각 화소의 $\alpha$값은 1이 되고,

[5]물론 두 이미지가 대응되는 화소끼리 정렬이 안 되어 있다면, 그에 대한 적절한 처리를 하여야 하지만 여기서는 설명의 편의상 이미 대응 화소끼리 서로 정렬이 되어 있다고 가정하자.

[6]S와 D라는 기호가 어색할 지 모르겠으나 S는 source, 그리고 D는 destination에 왔다고 생각하면 되는데, 이는 OpenGL과 같은 렌더링 시스템에서 S는 지금 프레임 버퍼에 그리고자 하는 이미지, D는 현재 프레임 버퍼에 들어 있는 이미지를 나타낸다. 물론 여기서는 이 두 기호가 프레임 버퍼와는 아무런 관계가 없다

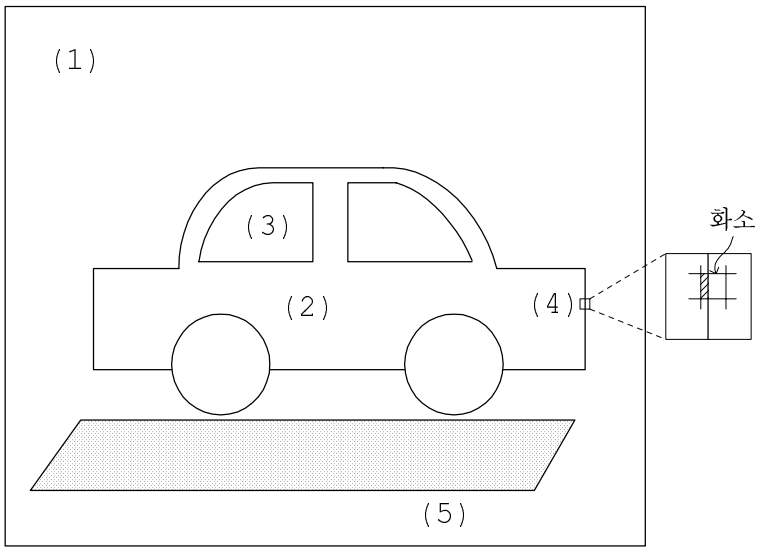



그림 3.4: 렌더링한 자동차

$(R, G, B)$는 원래 촬영의 결과 얻어진 색깔로 하면 된다.

반면에 S 이미지의 자동차는 그림 3.4에서 보듯이 불투명한 자동차 몸체와 바퀴, 그리고 약간 투명한 유리창으로 구성되어 있다. 이 경우 자동차가 나타나지 않는 영역((1))은 $R, G, B$ 값에 상관없이 $\alpha$가 0이 되고, 자동자의 몸체나 바퀴에 해당하는 영역((2))은 렌더링하여 얻어진 색깔 $(R_b, G_b, B_b)$에 대하여 $(R_b, G_b, B_b, 1)$와 같이 표현하고, 유리창 부분((3))의 화소는 유리의 색깔이 $(R_g, G_g, B_g)$라 할 때, 유리의 불투명도에 따른 적절한 알파 값, 예를 들어 $\alpha = 0.15$,을 사용하여 $(R_g, G_g, B_g, 0.15)$와 같이 나타낼 수 있을 것이다. 그리고 몸체도 실제로 경계 지역((4))에 대하여 화소 내에서 자동차가 차지하는 비율을 계산하여, 그 비율을 알파 값으로 사용하면 경계 부분을 훨씬 부드럽게 해줄 수가 있다. 그림 3.4에 한 화소가 확대되어 있는데, 이 화소 내에서의 비율이 0.25이라면 $(R_b, G_b, B_b, 0.25)$과 같이 색깔을 나타내면 된다. 그리고 그림자 영역((5))의 경우 얼마나 그림자를 짙게 할 것인가에 따라 적절한 알파 값을 설정하여, 예를 들어 $\alpha = 0.3$, (0.2, 0.2, 0.2, 0.3)과 같이 색깔 값을 부여할 수가 있다. 지금까지 합성을 하려하는 간단한 예를 생각해보았는데, 이와 같이 원

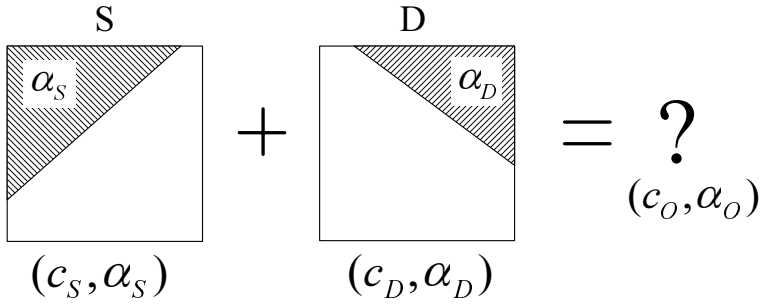



그림 3.5: 두 화소 색깔의 합성

하는 목적에 따라 각 화소의 알파 값을 적절히 설정할 수가 있다.

이제 각 화소끼리의 합성 방법에 대하여 알아보면, RGBA 모델로 주어진 색깔 $(R, G, B, \alpha)$는 종종 $R, G, B$ 각 채널에 $\alpha$를 곱하여 $(r, g, b, \alpha) \equiv (\alpha R, \alpha G, \alpha B, \alpha)$와 같이 표현하는데, 이러한 방법의 장점은 곧 알게 되겠듯이 이미지 합성 계산을 효율적으로 하게 해준다는 것이다. 이렇게 표현한 색깔을 미리 곱한 색깔(pre-multiplied color) 또는 연합 색깔(associated color)이라고 부르는데, 이 절에서는 원래의 $R$, $G$, $B$ 값과 구별하기 위하여 연합 색깔의 각 채널을 소문자 $r$, $g$, $b$로 표현하겠다. 만약 연합 색깔 (0.32, 0.0, 0.0, 0.4)로 화소를 칠한다면 이는 화소의 40%만 약간 어두운 빨간색 (0.8, 0.0, 0.0)으로 칠하는 것을 의미할 것이다. 반면에 연합 색깔 (0.32, 0.32, 0.32, 0.32)는 순순한 흰색으로 화소의 32%만 칠하는 것을 의미할 것이다.

이제 합성하려고 하는 두 이미지 S와 D 안에서 서로 대응되는 임의의 두 화소를 생각해보자. 각 이미지로부터의 대응되는 색깔을 편의상 RGB 채널과 알파 채널을 나누어 $(C_S, \alpha_S)$와 $(C_D, \alpha_D)$와 같이 나타내자(여기서 $C_S = (R_S, G_S, B_S)$이고 $C_D = (R_D, G_D, B_D)$.). 또한 이들의 연합 색깔을 각각 $(c_S, \alpha_S)$와 $(c_D, \alpha_D)$라 하자(즉 $c_S = (\alpha_S R_S, \alpha_S G_S, \alpha_S B_S)$이고 $c_D = (\alpha_D R_D, \alpha_D G_D, \alpha_D B_D)$). 그림 3.5는 이 두 화소를 합성하려고 하는 상황을 보여주고 있는데, S와 D의 화소에는 각각 $C_S$와 $C_D$의 색깔을 가지는 미립자들이 $\alpha_S$와 $\alpha_D$의 확률로 랜덤하게 흩어져 있다. 이들을

합성할 때 서로 어떻게 영향을 미치는가에 따라 그 결과가 달라지는데, 두 화소를 겹칠 때의 상황을 그림 3.6과 같이 생각할 수 있다. 미립자들이 확률적으로 랜덤하게 분포되어 있다면 S의 입자와 D의 입자가 같은 곳에 있을 확률은 $\alpha_S \cdot \alpha_D$가 될 것이다(그림의 SD 영역). 같은 이치로 단순히 S의 입자만 있을 확률은 $\alpha_S \cdot (1-\alpha_D)$가 되고(S 영역), D의 입자만 나타날 확률(D 영역)과 둘 다 모두 나타나지 않을 확률(N 영역)은 각각 $(1-\alpha_S) \cdot \alpha_D$와 $(1-\alpha_S) \cdot (1-\alpha_D)$가 된다고 볼 수가 있다.

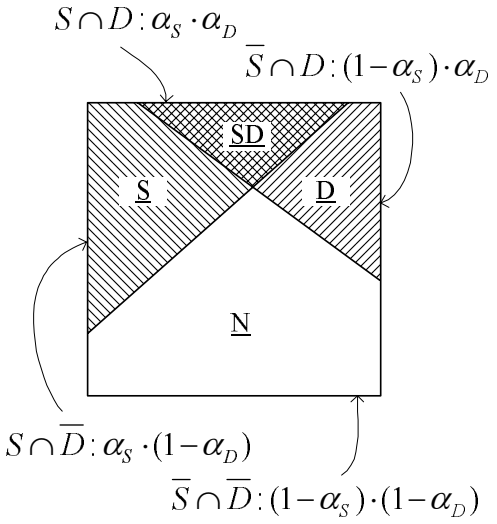
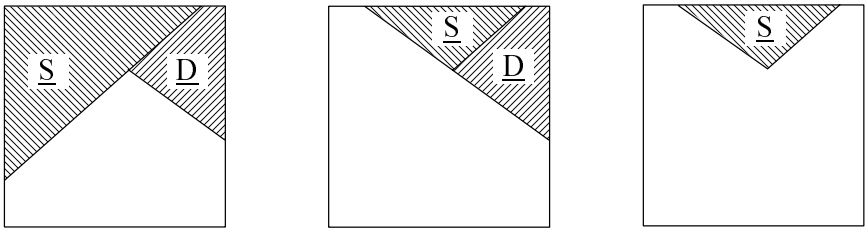



그림 3.6: 합성의 경우

따라서 두 화소를 겹쳐 합성할 때 화소를 네 개의 영역으로 나눌 수가 있는데, 문제는 각 영역을 어떠한 색깔로 칠하는가에 따라 그 결과 색깔이 달라지게 된다. 각 영역들을 생각해보면 S 영역은 S 입자의 색깔만 보일 수 있는 지역이라고 할 수가 있다. 마찬가지로 영역 D는 D 입자의 색깔만 보일 수 있고, SD 영역은 S와 D 입자 중 어떤 입자의 색깔도 보일 수가 있다. 반면에 영역 N은(여기서 N은 null, 즉 아무 것도 없다는 뜻임.) 어느 입자에도 막히지 않고 뒤에서 빛이 통과하여 들어오는 영역이라 볼 수 있다. 두 화소를 합성할 때 각 영역에 대한 색깔을 결정해야 하는데, 이 때 보일 수 있는 색깔 중에서 하나를 선택하거나 아무 것도 선택하지 않는 방법을 택한다. 예를 들어 SD 영역에 대해서는 S의 색깔을 선택하거나, D의 색깔을 선택하거나, 아무 색깔도 선택하지 않는 세 가지 경우를 생각할 수 있다. 마지막 경우는 이 영역을 빈 공간으로 놔두겠다는 것인데, S와 D 영역에 대해서는 각각 자신의 색깔을 선택하거나 아무 것도 선택하지 않




(a) S **over** D (b) S **atop** D (c) S **in** D

그림 3.7: 세 가지 합성 예

는 두 가지 경우가 있음을 알 수가 있다. 마지막으로 $\underline{\mathrm{N}}$ 영역은 아무 것도 선택하지 않는 한 가지 경우만 있게 된다. 따라서 네 개의 영역에 대하여 색깔을 선택할 때 $12(=3\cdot 2\cdot 2\cdot 1)$가지의 경우가 존재하게 되고, 각 지역에 대하여 어떠한 선택을 하는가에 따라 다양한 합성의 효과를 낼 수 있다.

예를 들어 $\underline{\mathrm{SD}}$, $\underline{\mathrm{S}}$, $\underline{\mathrm{D}}$, $\underline{\mathrm{N}}$ 각 영역에 대한 선택의 결과가 (S, S, D, N)일 때 어떠한 합성 효과가 나타날까? 그림 3.7(a)를 보면 쉽게 이해할 수 있는데, 이는 바로 D 이미지의 그림 위에 S 이미지의 그림을 덮어쓰는 결과를 낳게 된다(S **over** D). 반면에 (S, N, D, N)과 같이 선택을 하면 그림 3.7(b)에서 보듯이 D 이미지의 그림이 있는 곳에만 S 이미지의 그림이 나타나게 된다(S **atop** D). 이러한 효과는 한 이미지를 다른 이미지 안의 어떤 물체에 갖다 붙일 때 유용하게 쓰일 수가 있는데, 예를 들어 S 이미지는 알파 채널의 값이 모두 1인 벽지 그림이라 하고, D 이미지의 경우 방 안의 벽면 부분에 대해서만 0보다 큰 알파 값을 갖도록 한 후 위와 같이 합성을 한다면 바로 벽면에만 벽지가 붙여지는 효과를 낼 수 있을 것이다. 그림 3.7(c)의 세 번째 선택은 (S, N, N, N)에 해당하는데 이는 S와 D가 겹치는 부분만 S의 색깔로 칠하는 경우이다(S **in** D). 표 3.1에는 12가지 모든 경우에 대하여 그 이름과 함께 설명이 되어 있는데, 그 의미를 곰곰이 생각해보기 바란다.

| 연산의 이름 | 선택 방법 | $F_S$ | $F_D$ |
|---|---|---|---|
| **clear** | (N, N, N, N) | 0 | 0 |
| S | (S, S, N, N) | 1 | 0 |
| D | (D, N, D, N) | 0 | 1 |
| S **over** D | (S, S, D, N) | 1 | $1-\alpha_S$ |
| D **over** S | (D, S, D, N) | $1-\alpha_D$ | 1 |
| S **in** D | (S, N, N, N) | $\alpha_D$ | 0 |
| D **in** S | (D, N, N, N) | 0 | $\alpha_S$ |
| S **held-out-by** D | (N, S, N, N) | $1-\alpha_D$ | 0 |
| D **held-out-by** S | (N, N, D, N) | 0 | $1-\alpha_S$ |
| S **atop** D | (S, N, D, N) | $\alpha_D$ | $1-\alpha_S$ |
| D **atop** S | (D, S, N, N) | $1-\alpha_D$ | $\alpha_S$ |
| S **xor** D | (N, S, D, N) | $1-\alpha_D$ | $1-\alpha_S$ |

표 3.1: 12가지 선택의 경우

이제 원하는 합성의 종류를 결정하였을 때, 합성의 결과 생성되는 화소의 연합 색깔 $(c_O, \alpha_O)$를 어떻게 계산하는지에 대하여 알아보자. 이를 위하여 $F_S$와 $F_D$를 각각 S와 D의 화소에서의 $\alpha_S$와 $\alpha_D$의 비율로 존재하는 미립자들 중 결과 이미지 O에 살아남는 미립자의 비율이라 하자. 예를 들어보면 S **over** D의 경우 S의 색깔은 합성 후 전체가 살아남으므로 $F_S = 1$이 되고, 반면에 D의 색깔은 가리고 남은 부분만 보이게 되므로 그 비율 $F_D$는 $1-\alpha_S$가 된다. 또한 S **atop** D의 경우 $F_S = \alpha_D$와 $F_D = 1-\alpha_S$가 되고, S **in** D는 $F_S$와 $F_D$는 각각 $\alpha_D$와 0이 된다. 위에서 설명한 12가지 선택에 대하여 $F_S$와 $F_D$가 고유하게 결정이 되므로, 원하는 합성의 종류는 이 두 값을 통하여 정의할 수 있다(표 3.1 참조).

합성을 한 결과 화소의 알파 값이 $\alpha_O$라 하면, 이는 합성 후 S와 D의 입자에 의해 가린 영역의 비율이므로, $\alpha_O = \alpha_S F_S + \alpha_D F_D$가 됨을 알 수 있다. 또한 합성 후의 연합 색깔과 실제 색깔을 각각 $c_O$와 $C_O$라 하면 $c_O$는 다음과 같이 계산을 할 수 있다.

$$c_O \quad = \quad \alpha_O \cdot C_0 = \alpha_O \cdot \frac{(\alpha_S F_S) \cdot C_S + (\alpha_D F_D) \cdot C_D}{\alpha_S F_S + \alpha_D F_D}$$

$$
\begin{aligned}
&= \alpha_O \cdot \frac{(\alpha_S F_S) \cdot C_S + (\alpha_D F_D) \cdot C_D}{\alpha_O} \\
&= \alpha_S F_S C_S + \alpha_D F_D C_D \\
&= \alpha_S F_S \frac{c_S}{\alpha_S} + \alpha_D F_D \frac{c_D}{\alpha_D} \\
&= c_S F_S + c_D F_D
\end{aligned}
$$

따라서 합성의 결과 생성되는 연합 색깔은 다음과 같이 간결하게 표현할 수 있는데, 물론 그 값은 $F_S$와 $F_D$의 함수임을 알 수가 있다.

$$
\begin{pmatrix} c_O \\ \alpha_O \end{pmatrix} = F_S \begin{pmatrix} c_S \\ \alpha_S \end{pmatrix} + F_D \begin{pmatrix} c_D \\ \alpha_D \end{pmatrix}
$$

이 결과를 보면 화소의 색깔을 연합 색깔 형태로 표현하는 주된 이유를 알 수 있다. 연합 색깔을 사용하면 합성 계산을 할 때 $r, g, b$ 채널과 $\alpha$ 채널의 구분 없이 동일한 방식을 사용할 수 있게 되므로, 합성을 효율적으로 할 수 있다. 기본 합성 중 가장 널리 쓰이는 합성은 S **over** D라 할 수 있다. 이 경우 $F_S = 1$와 $F_D = 1 - \alpha_S$이므로 합성 공식은 아래와 같다.

$$
\begin{aligned}
\begin{pmatrix} c_O \\ \alpha_O \end{pmatrix} &= \begin{pmatrix} c_S \\ \alpha_S \end{pmatrix} + (1 - \alpha_S) \begin{pmatrix} c_D \\ \alpha_D \end{pmatrix} \\
&= \begin{pmatrix} \alpha_S C_S \\ \alpha_S \end{pmatrix} + (1 - \alpha_S) \begin{pmatrix} \alpha_D C_D \\ \alpha_D \end{pmatrix}
\end{aligned}
$$

이 합성은 OpenGL뿐만 아니라 대부분의 렌더링 시스템에서 투명한 물체를 렌더링하거나, 안개와 같은 기상 효과, 텍스춰의 혼합, 앤티앨리어싱 기법들을 구현

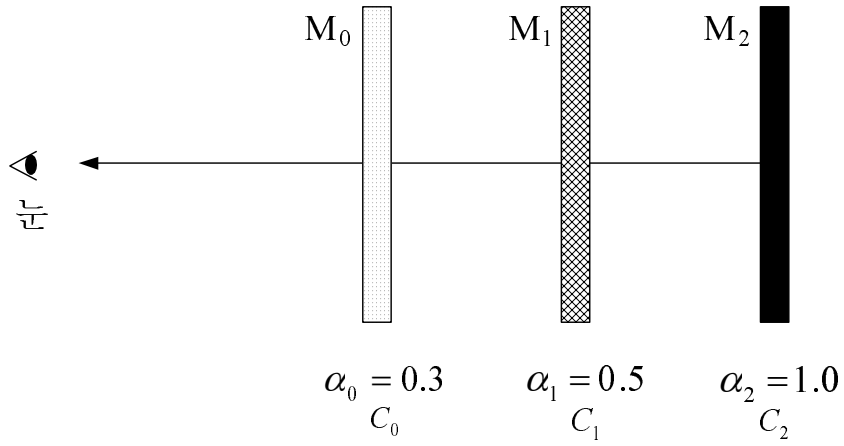



그림 3.8: S **over** D 합성의 예

하는데 기본적으로 사용되는 합성이다. 따라서 이에 대하여 정확한 이해가 필요한데, 간단한 예로 그림 3.8과 같은 상황을 생각해 보자. 지금 관찰자가 세 개의 물체 $M_0$, $M_1$, $M_2$를 바라보고 있는데, $M_0$는 실제 색깔이 $C_0$인 유리로서 뒤에서 들어오는 빛의 70%를 통과시킨다(즉 $\alpha_0 = 0.3$). $M_1$은 색깔이 $C_1$이고 빛을 50% 통과($\alpha_1 = 0.5$)시키고, 제일 오른쪽에 있는 $M_2$는 불투명한 벽($\alpha_2 = 1.0$)으로서 $C_2$의 색깔을 가진다. 이 때 눈에 보이는 색깔은 ($M_0$ **over** $M_1$) **over** $M_2$와 같은 합성에 의하여 구할 수 있다. 우선 $M_0$과 $M_1$을 합성해보면 다음과 같고,

$$\begin{aligned} c_{01} &= \alpha_0 \cdot C_0 + (1-\alpha_0) \cdot \alpha_1 \cdot C_1 \\ &= 0.3 \cdot C_0 + (1-0.3) \cdot 0.5 \cdot C_1 = 0.3C_0 + 0.35C_1 \\ \alpha_{01} &= \alpha_0 + (1-\alpha_0) \cdot \alpha_1 \\ &= 0.3 + (1-0.3) \cdot 0.5 = 0.65 \end{aligned}$$

여기에 같은 방식으로 $M_2$를 합성하면 $c_{012} = 0.3C_0 + 0.35C_1 + 0.35C_2$이고 $\alpha_{012} = 1.0$이 되므로 실제 색깔이 $C_{012} = c_{012}$인 색깔이 보이게 된다.

**잡깐 3.1** 두 이미지 S와 D의 화소의 합성에 대한 계산은 두 개의 피연산자를 가지는 이진 연산(binary operation)으로 볼 수가 있다. 예를 들어 A **over** B의 경우 덧셈이나 뺄셈처럼 **over**라는 연산자를 사용한 연산이라 생각할 수가 있는데, 이 연산자의 중요한 특징중의 하나는 결합 법칙이 성립한다는 사실이다. 이는 여러 개의 물질에 대하여 **over** 연산자를 사용하여 일정한 순서대로 합성을 할 경우, 실제로 합성을 하는 순서에 상관이 없음을 의미한다[7]. CT나 MRI 데이터와 같은 의료 데이터에 대하여 3차원 렌더링 기법을 적용할 때, 화소에 대한 시선 상의 점들을 균일한 간격으로 샘플링을 하여 쉐이딩이 된 RGBA 색깔을 앞에서 뒤로 가면서 **over** 연산자를 사용하여 합성을 한다. 데이터가 비교적 클 경우 이러한 합성은 생성하려는 이미지의 각 화소에 대하여 계산이 되어야 하므로, 방대한 양의 부동 소수점 연산을 필요로 한다. 연산자에 결합 법칙이 성립한다는 것은 그것을 통한 계산에 동시성이 존재한다는 것을 의미한다. 따라서 **over** 연산을 통하여 512개의 색깔을 순서대로 합성해야 한다면 이를, 예를 들어, 64개씩 나누어 8개의 프로세서를 사용하여 병렬적으로 합성한 후, 각각의 결과를 앞에서 뒤로 합성을 해주면 계산의 속도를 높이면서 정확한 합성을 할 수 있게 된다.

다음 절로 넘어가지 전에 표 3.1에 주어진 12개의 합성 연산 외에 유용하게 쓰일 수 있는 연산 몇 가지에 대하여 알아보자. 지금까지 살펴본 연산은 모두 이진 연산이었는데, 종종 한 이미지 S의 화소에 대한 일진 연산(unary operation)도 이미지 합성에 중요한 역할을 하게 된다. 첫째로, **darken**(S, $\phi$) = $(\phi \cdot r_S, \phi \cdot g_S, \phi \cdot b_S, \alpha_S)$ $(0 \leq \phi \leq 1)$은 주어진 값 $\phi$를 연합 색깔의 $r, g, b$ 채널에 곱해주는 연산이다. 만약 1보다 작은 $\phi$값을 사용하면 불투명도 $\alpha_S$가 고정된 상황에서 $c_S$ 부분만 일률적으로 작아지므로, 결과적으로 실제 화소에 칠하는 색깔 $C_S$의 각 채널의 값이 작아져 이미지

[7] 물론 이 연산은 교환 법칙이 성립하지 않으므로 전후 순서를 유지하여야 한다

전체가 어두워지는 효과를 낳는다. 두 번째로 **fade**(S, $\sigma$) = ($\sigma \cdot r_S, \sigma \cdot g_S, \sigma \cdot b_S, \sigma \cdot \alpha_S$) ($0 \leq \sigma \leq 1$) 연산은 마찬가지로 0과 1사이의 값을 가지는 $\sigma$를 각 채널에 곱해주는데, 그 결과 $C_S$는 변함이 없이 알파 채널 값만 작아지게 된다. 따라서 $\sigma$를 1에서 0으로 조금씩 변화시켜가면서 이 연산을 반복적으로 수행하면 이미지 전체에 대한 알파 값이 점점 작아지므로, 이미지의 내용이 점점 사라지는 페이드 아웃(fade-out) 효과가 있게 된다. 세 번째 일진 연산의 예로 **opaque**(S, $\omega$) = ($r_S, g_S, b_S, \omega \cdot \alpha_S$) ($0 \leq \omega \leq 1$)와 같이 정의되는 합성을 생각할 수가 있다. 이 연산은 오로지 알파 채널 값만 변화시키는데, 만약 1보다 작은 $\omega$를 사용하면 이는 불투명도는 낮추면서 실제로 칠할 색깔은 더 밝게 해주는 효과가 있을 것이다.

마지막 연산자로서 S **plus** D = ($r_S + r_D, g_S + g_D, b_S + b_D, \alpha_S + \alpha_D$)와 같이 단순히 각 채널의 값을 더하는 합성을 생각해보자. 이 연산자는 특히 $\alpha_S + \alpha_D = 1$일 때 유용하게 쓰일 수가 있다. 그러한 예로 O = (**fade**(S, $t$)) **plus** (**fade**(D, 1-$t$)) ($0 \leq t \leq 1$)와 같은 복합 연산을 생각해보자. 결과 이미지 O는 $t$의 함수로 볼 수 있는데, 이 값을 0에서 1로 조금씩 증가시키면서 반복적으로 합성한 이미지 O를 화면에 도시할 경우 바로 영화나 드라마에서 널리 쓰이는 D 이미지로부터 S 이미지로의 전이(transition) 효과가 발생할 것이다. 지금까지 알파 채널을 사용한 이미지 합성에 대하여 알아보았는데, OpenGL에서도 이러한 계산을 할 수 있도록 알파 혼합(alpha blending) 기능을 제공한다. 이에 대해서는 6장에서 다루기로 하겠다.

## 제 4 절 광원의 종류

이제 본 3장의 주제인 조명 모델, 즉 라이팅 모델에 대하여 살펴보자. 이를 위해서 이 절에서는 우선 라이팅 모델에 있어 가장 기본이 되는 요소중의 하나인 광원, 즉 스스로 빛을 발하는 물체를 어떻게 표현하는지에 대하여 알아보겠다. 앞에서도 설명한 바와 같이 광원으로부터 출발한 빛이 복잡한 경로를 통하여 색깔을 구하려하는 물체의 지점에 들어와 그 지점에서 어떻게 반사가 되는지를 결정해주는 것이 라이팅 모델의 목적이다. 대부분의 컴퓨터 그래픽스 렌더링 기법들은 이러한 물리학적인 현상에 대하여 단순한 모델을 설정하여 그러한 계산을 수행한다고 하였는데, 이를 위하여 다음과 같은 점을 고려해야 한다.

- [문제 1] 과연 광원이라는 물체를 어떻게 정의할 것인가?
- [문제 2] 색깔을 계산하려는, 즉 쉐이딩을 하려는 지점에 들어오는 빛의 경로, 방향, 밝기, 그리고 색깔 등을 어떠한 모델로 정의할 것인가?
- [문제 3] 물체 표면에서 빛이 반사되는 방식은 그 물체를 이루는 물질의 성질에 의하여 결정이 되는데, 이러한 물질의 빛의 반사 형태를 어떻게 정의할 것인가?
- [문제 4] 쉐이딩 하려는 지점에 들어오는 빛과 물체 표면의 반사 성질이 결정되었을 때, 과연 어떠한 모델을 사용하여 시선의 방향으로 반사되는 빛의 색깔을 계산할 것인가?

라이팅 모델을 설정하기 위해서는 이러한 사항들을 정확하게 정의하여야 한다. 이 절에서는 우선 [문제 1]과 [문제 2]에 대하여 살펴보고, 나머지는 5절에서 알아보도록 하겠다.

그래픽스 렌더링에서 사용되는 광원이라는 용어는 우리가 보통 생각할 수 있는 전구나 태양과 같이 스스로 빛을 발하는 물체뿐만 아니라 좀 더 넓은 의미로 사용이 된다. 예를 들어 밀폐된 방에서 뿌연 유리창을 통하여 빛이 들어온다고 할 때, 렌더링 입장에서는 이 유리창도 하나의 광원으로 간주가 될 수가 있다. 따라서 렌더링을 위하여 여러 형태의 광원을 생각할 수 있으나, 보통 몇 가지 형태의 광원으로 제한을 둔다. 일반적으로 가장 흔하게 쓰이는 광원으로 점 광원(point light source)과 평행 광원(parallel light source 또는 directional light source)을 들 수 있다. 광원도 하나의 부피를 가지는 물체이기 때문에 정확하게 광원을 모델링하려면 광원 표면의 각 지점에서 사방으로 빛을 발하는 상황을 처리해주어야 하지만, 일반적으로 이는 과다한 계산을 초래할 것임을 쉽게 예측할 수가 있다.

만약 광원의 크기가 렌더링을 하려는 전체 장면에 비해 상대적으로 작다면, 계산의 편의상 이 광원이 한 점에서 사방으로 빛을 발한다고 가정할 수가 있는데, 이러한 광원이 바로 점 광원이다. 물체의 표면에 빛이 들어오는 각도, 즉 입사각은 라이팅 계산에서 있어 중요한 인자중의 하나인데, 예를 들어 백열 전구와 같은 광원을 생각하면 전구 표면의 여러 지점에서 빛을 발하기 때문에, 물체 표면의 한 지점의 입장에서 보면 여러 각도로 빛이 들어오게 된다. 하지만 전구가 물체에서 충분히 떨어져 있고 상대적으로 크기가 작다면, 한 지점으로 들어오는 빛의 각도의 차이가 그리 크지 않게 된다. 따라서 전구를 3차원 공간의 한 점에서 빛을 발하는 광원으로 모델링을 함으로써, 물론 그로 인한 시각적인 오차가 발생하겠지만, 필요한 계산량을 줄일 수가 있다.

반면에 광원이 물체로부터 상당히 멀리 떨어져 있다면 물체의 각 지점으로 들어오는 빛의 각도는 거의 변함이 없을 것이다. 이러한 경우 빛이 일정한 방향으로 평행하게 들어온다고 가정을 해도 큰 문제가 없는데, 이러한 광원이 바로 평행 광원

이다. 태양이 평행 광원의 좋은 예인데, 이러한 광원은 물체의 어느 지점에서나 빛이 들어오는 각도가 변함이 없기 때문에, 점 광원처럼 물체 표면의 각 지점에서의 입사 각도를 계산할 필요가 없어, 계산량 입장에서 볼 때 매우 효율적인 광원이라고 할 수 있다.

요약을 하면 광원을 설정하기 위하여 필요한 기하 속성은 점 광원의 경우에는 광원의 위치(점)가 되고, 그리고 평행 광원의 경우에는 빛이 들어 오는 방향(벡터)이 된다. 이 속성은 동차 좌표계 $(x\ y\ z\ w)^t$를 사용하여 나타낼 수가 있는데, $w$가 0이 아니면, 다시 말해서 $w = 1$이면 점 광원의 위치, 그리고 $w$가 0이라면 평행 광원의 방향을 나타낸다. 이러한 관점에서 보면 평행 광원은 단지 무한대 점에 위치하는 점 광원이라 생각할 수 있다. 기하 속성 외에 광원의 밝기와 색깔도 중요한 속성인데, 이는 RGB 모델을 사용하여 $I_l = (I_R, I_G, I_B)$와 같이 나타낸다. 그림 3.9(a)와 (b)를 보면 각각 점 광원과 평행 광원을 사용하였을 때의 차이를 알 수 있는데, 평행 광원과는 달리 점 광원의 경우 물체 표면의 각 지점에 들어오는 빛의 각도가 변하기 때문에 반사되는 빛의 밝기가 많이 변하게 된다. 반면에 평행 광원을 사용하면 상대적으로 완만하게 밝기가 변하는 이미지를 생성하게 된다.

이 두 광원과 함께 많이 쓰이는 광원으로 스폿 광원(spot light source)을 들 수 있다. 이 광원은 특수한 제한을 가지는 점 광원으로서 사방으로 빛을 발하는 것이 아니라, 원뿔과 같이 일정한 범위로 빛을 발하는 광원이다. 이 광원을 사용하면 무대 조명, 손전등, 자동차 헤드라이트와 같은 광원의 효과를 낼 수가 있는데, 이 광원의 기하 속성으로 광원의 위치, 빛을 발하는 중심 방향과 범위를 나타내는 각도를 설정해주어야 한다. 그림 3.9(c)를 보면 스폿 광원의 효과를 분명히 볼 수 있다.

또 다른 광원으로 처음에는 그 용도가 직관적으로 명확하게 이해가 되지 않을지 모르겠으나, 매우 중요하게 쓰이는 앰비언트 광원(ambient light source)을 생각할


(a) 점 광원


(b) 평행 광원


(c) 스폿 광원


(d) 앰비언트 광원


(e) 분산 광원

그림 3.9: 광원의 종류

수 있다[8]. 이는 광원이 분명히 존재하나 어디에 있는지 알기 힘들거나, 광원에서 들어오는 빛을 정확하게 또는 단순하게 모델링하기 힘든 상황에 대처하기 위하여 사용되는 광원이다. 앞에서도 언급한 바와 같이 OpenGL과 같은 실시간 그래픽스 렌더링 파이프라인에서는 계산량을 줄이기 위하여 지역 조명 모델을 사용하는데, 이 모델에서는 광원에서 물체로 직접 들어오는 빛만 고려하기 때문에 전역 조명 모델에서처럼 간접적으로 들어오는 빛의 효과를 내지 못한다. 직접 조명만 사용하여 렌더링을 한 물체를 보면 빛이 들어오는 부분과 안 들어오는 부분간의 차이가 너무 심해 상당히 거친 느낌을 준다. 간접 조명은 매우 복잡하기 때문에 실시간 렌더링 파이프라인에서는 직접적으로 지원을 해 줄 수는 없고, 단지 간접 조명의 흉내를 내기 위한 일환으로 어느 정도 밝기의 빛이 사방에 골고루 퍼져있다고 가정을 하는데, 이 것이 앰비언트 광원의 중요한 목적이라 할 수 있다. 예를 들어 방안에 세 개의 광원이 있다고 가정하면 라이팅 계산 시 물체 표면의 한 지점에 대하여 각 광원에서 직접 들어오는 빛의 효과를 더하고, 나머지 간접적으로 들어오는 빛은 하나로 뭉뚱그려 앰비언트 광원으로 생각을 한다. 따라서 이 광원은 스스로가 중요한 역할을 한다기보다는 지역 광원 모델의 문제를 보완하는 것을 목표로 하기 때문에, 그림 3.9(d)에서처럼 비교적 약한 밝기를 가진다.

마지막으로 살펴볼 광원은 분산 광원(distributed light source)으로서 이는 광원이 전체 장면이나 물체에 대하여 무시할 수 없는 면적을 가져 점 광원으로 근사화를 하기 힘들 때 사용한다. 예를 들어 형광등이 비교적 가까운 위치에서 물체를 비추고 있을 때 점 광원에서처럼 형광등을 한 점으로 단순화시킬 경우 상당히 부자연스러운 효과를 얻게 된다. 즉 형광등 표면의 여러 지점에서 물체로 빛이 들어오기 때문에 한 점에서 빛이 들어오는 것과는 판이한 결과를 낳게 된다. 이러한 경우 형

[8] 또는 단순히 앰비언트 빛(ambient light)이라고 한다.

광등을 선분, 또는 길쭉한 직사각형으로 생각하고 그 위의 각 지점에서 발하는 빛의 효과를 계산한다. 실제로 우리가 사용하는 대부분의 광원은 점 광원이라고 하기보다는 분산 광원이라 할 수 있는데, 이 광원을 사용하면 매우 자연스러운 효과를 낼 수 있다(그림 3.9(e)). 반면에 길이, 면적, 또는 경우에 따라 체적을 가지는 물체의 여러 지점에서 출발하는 빛의 효과를 계산해주어야 하기 때문에 계산량이 상당히 많아 실시간 그래픽스에서는 직접 지원하지 않고 고급 기법을 사용하여 분산 광원 효과를 흉내내게 된다. 여기서 비단 렌더링뿐만 아니라 그래픽스 분야에서 사용되는 모델의 단순화 정도에 따른 장단점의 차이를 알 수 있는데, 점 광원이나 평행광원처럼 극도로 단순화 된 모델에서는 계산량이 적어 빠른 시간 안에 렌더링을 할 수 있으나, 분산 광원에 비해 상대적으로 생경한 이미지를 만들게 된다.

지금까지 설명을 한 내용을 종합해보면 앞에서 언급한 네 가지 문제 중 처음 두 가지에 대하여 답을 할 수가 있다. 렌더링 계산에서 사용되는 광원 모델로 점 광원, 평행 광원, 스폿 광원, 앰비언트 광원, 분산 광원 등이 있는데, 보통 실시간 렌더링 파이프라인에서는 앞의 네 가지 모델을 지원한다([문제 1]). 또한 실시간 파이프라인에서는 기본적으로 지역 조명 모델을 사용하기 때문에 광원에서 직접 들어오는 빛만 고려를 하고(경로), 방향은 광원의 기하 속성과 물체 표면의 위치에 의해 결정이 되며, 마지막으로 밝기나 색깔은 각 광원을 설정하는데 사용하는 색깔을 사용한다([문제 2]). 물론 밝기는 물체와 광원간의 거리를 사용하여 조절하기도 하는데 자세한 내용은 뒤에서 설명하겠다.

## 제 5 절 퐁의 조명 모델

이제 나머지 두 문제 [문제 3]과 [문제 4]에 관련된 사항에 대하여 알아보자. 앞 절의 내용은 물체 표면 상의 점으로 들어오는 빛을 어떻게 결정할 것인가에 관한 것

이었다. 이제 빛이 물체를 향해 들어 왔을 때, 우리가 바라보고 있는 시선의 방향으로 반사되는 빛의 색깔을 어떠한 방식으로 계산해야 할 지를 결정하여야 한다. 우리가 물체를 특정 색깔로 느낀다는 것은 광원으로부터의 입사 광선 중 일부는 물체에 흡수가 되고, 나머지 빛이 눈을 향하여 반사가 될 때 그러한 빛의 색깔로 느끼는 것인데, 여기에는 주로 광원의 밝기와 색깔, 빛이 들어오는 방향, 물체를 바라보는 방향, 그리고 물체의 기하 및 반사 성질 등이 영향을 미친다. 그래픽스 렌더링에서 사용되는 대부분의 라이팅 모델의 기반이 되는 퐁의 조명 모델(Phong's illumination model)은 바로 물체 표면에서 빛이 어떻게 반사가 되는가를 계산하는데 사용이 된다. 특히 이 모델은 주어진 물체의 빛의 반사 형태를 결정하는데 사용되므로 퐁의 반사 모델(Phong's reflection model)이라고 하기도 한다. 이 모델은 물리학적으로 정확한 모델은 아니나 비교적 계산량이 적고, 실험적으로 우수한 성능을 나타내기 때문에, 컴퓨터 그래픽스 렌더링 분야에서 거의 표준처럼 사용이 되고 있다. 실제로 퐁의 모델을 상황에 맞게 조금씩 변형을 하여 사용하나, 그 기본은 동일하다고 할 수 있다. 이 절에서는 퐁의 조명 모델 전반에 대하여 살펴보고, 다음절에서 OpenGL 시스템에서 실제로 사용되는 모델에 대하여 구체적으로 알아보도록 하겠다.

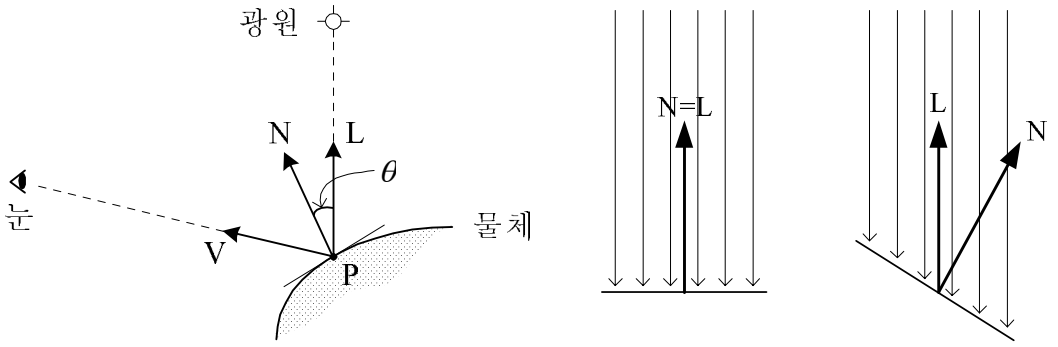
## 5.1 세 가지 종류의 반사

퐁의 조명 모델에서 고려하는 빛의 반사는 기본적으로 앰비언트 반사(ambient reflection), 난반사(diffuse reflection), 그리고 정반사(specular reflection) 등 세 가지 형태의 반사로 나누어진다.

### 5.1.1 앰비언트 반사

앰비언트 반사는 물체가 앰비언트 광원에 대하여 어떻게 반응을 할 것인가에 관한 것이다. 앰비언트 빛이란 광원에서 직접 들어오는 빛이 아니라 간접적으로 들어오는 빛을 모델링 하기 위하여 사방에 고르게 퍼져 있다고 가정하는 빛을 의미하는데, 광원의 색깔은 파장 $\lambda$의 가시 광선 영역에 대하여 $I_{a\lambda}$와 같이 설정할 수 있다. 광원의 밝기 또는 색깔을 정확하게 설정하려면 모든 파장 $\lambda$에 대하여 정의가 되어야 하나, 그래픽스 렌더링에서 사용되는 조명 모델에서는 RGBA 모델을 사용하여 $\lambda = R, G, B, A$에서 정의가 된다. 일단 이 절에서는 앞에서 언급한 바와 같이 편의상 RGB 색깔 모델을 사용한다고 가정하자. 이 경우 빛의 색깔은 $I_{a\lambda} = (I_{aR}\ I_{aG}\ I_{aB})^t$와 같이 세 개의 원소를 가지는 벡터로 표현된다.

앰비언트 빛이 물체 전체에 균일하게 들어올 때 물체를 이루는 물질의 성질에 따라 이를 반사하는 방식이 다르게 된다. 빛의 색깔의 각 채널에 대하여 물체가 각각 $k_{aR}$, $k_{aG}$, $k_{aB}$ $(0 \leq k_{aR}, k_{aG}, k_{aB} \leq 1)$의 비율만큼만 반사를 한다면 결과적으로 반사되어 보이는 빛은 $k_{a\lambda} \equiv (k_{aR}\ k_{aG}\ k_{aB})^t$라 할 때, $I_\lambda = I_{a\lambda} \cdot k_{a\lambda}$와 같이 표현할 수 있는데, 여기서의 곱셈은 각 채널 값끼리의 곱셈을 의미한다. 이 때 세 개의 원소를 가지는 벡터 $k_{a\lambda}$를 앰비언트 반사 계수(ambient reflection coefficient)라고 하는데, 물체가 들어오는 앰비언트 빛을 어떻게 반사를 시킬 것인가를 결정하는 물질의 고유 성질이다. 앰비언트 반사의 경우 빛이 사방에서 고르게 들어온다고 가정을 하며, 반사 또한 사방으로 균일하게 반사된다고 가정을 하기 때문에 관찰자의 위치에 상관없이 같은 색깔로 반사되어 보이게 된다. 이렇게 하여 엉성하기는 하나 지역 조명 모델에서 전역 조명 모델의 흉내를 내게 되는 것이다.




(a) 세 가지 중요한 방향 (b) 코사인 법칙의 원리

그림 3.10: 난반사의 원리

### 5.1.2 난반사

앰비언트 반사와는 달리 광원에서 직접 들어오는 빛은 크게 난반사와 정반사 등의 두 가지 종류의 반사를 한다. 우선 난반사는 입사 광선을 사방으로 고르게 동일한 밝기로 반사하는 형태의 반사를 시뮬레이션하는데 사용이 된다. 따라서 앰비언트 반사의 경우와 같이 만약 물체가 순수하게 난반사만 한다면, 관찰자의 시점이 어디에 있건 같은 밝기로 반사되어 보인다. 난반사는 물체 표면이 종이, 분필, 석고상 등과 같이 반짝거리지 않고 좀 둔탁해 보이는 물체를 렌더링 하는데 사용된다. 이러한 물체들은 시점을 이리 저리 옮겨가면서 보아도 비교적 보이는 색깔이 변하지 않는 성질을 가지고 있다. 그림 3.10(a)를 보자. 지금 물체 표면 상의 한 점 P에서 반사되는 빛의 색깔을 계산하려 한다. 광원에서 출발한 빛이 직접 물체로 들어오는 방향의 반대 방향을 L이라고 하고, 관찰자를 향해 반사되는 방향과 P에서 물체의 표면에 수직인 법선 벡터를 각각 V와 N이라는 벡터로 나타내자[9].

어떤 물체를 이루는 물질이 순수하게 난반사를 한다면 그 물체를 이상적인 난반사체(idea diffuse reflector)라고 하는데, 다른 말로 램버트 반사체(Lambertian re-

[9] 편의상 이 벡터들은 길이가 1인 단위 벡터로 가정하겠다.

flector)라고 부르기도 한다. 이러한 물체들은 표면에서 빛을 반사시킬 때 램버트의 코사인 법칙(Lambert's cosine law)을 따른다. 이 법칙에 의하면 두 벡터 L과 N사이의 각도를 $\theta$라 할 때 반사되는 빛 에너지의 양은 $\cos\theta$에 비례한다. 따라서 반사되는 빛의 색깔은 광원으로부터의 입사 각도와 바라보는 점에서의 법선 벡터에 영향을 받지만, 반면에 어디에서 바라보건(즉 V 벡터 방향이 어떻게 변하건) 동일하게 보인다. $\theta$값이 커질수록 $\cos\theta$의 값이 작아지므로, 이 법칙에 따르면 빛이 수직으로 내려 쪼일 때 가장 세게 반사가 되고, 비스듬하게 빛을 비출 수록 반사되는 빛의 밝기가 약해지는 성질을 가진다. 램버트의 법칙은 그림 3.10(b)를 보면 직관적으로 이해할 수 있다. 즉 왼쪽에서처럼 수직으로 빛이 내려 쪼일 때에 비해 오른쪽에서와 같이 $\theta$만큼의 입사 각도를 가지고 동일한 크기의 빛 에너지를 내려 쪼이면, 물체 표면의 단위 면적 당 빛의 입사량은 $\cos\theta$에 비례하여 줄어 들게 되고, 따라서 반사되는 빛의 양도 그에 따라 줄어 든다고 생각하면 된다. 순수한 난반사를 통하여 반사되는 빛의 색깔은 다음과 같은 공식에 의하여 계산된다.

$$I_\lambda = I_{l\lambda} \cdot k_{d\lambda} \cdot \cos\theta = I_{l\lambda} \cdot k_{d\lambda} \cdot (N \cdot L)$$

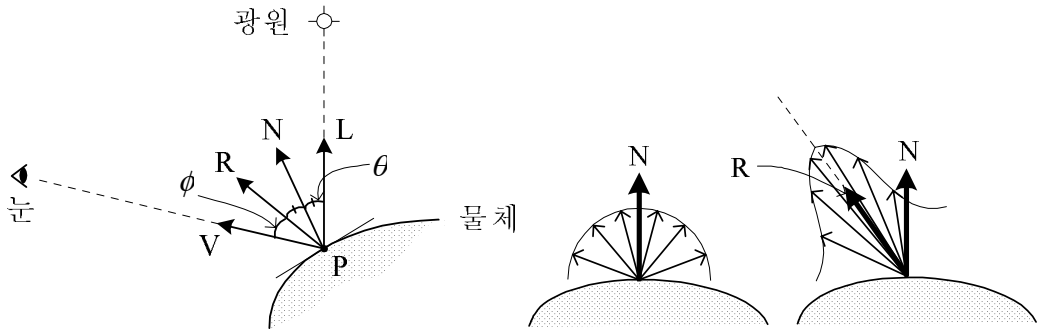
여기서 $I_{l\lambda}$는 광원의 색깔이고, $k_{d\lambda}$는 물체가 난반사를 할 때 각 채널의 빛을 어떠한 비율로 반사를 시킬 것인지를 결정하는 난반사 계수(diffuse reflection coefficient)로서, 앰비언트 반사 계수와 같이 세 개의 0과 1 사이의 값으로 구성된 벡터이다. 또한 여기서 두 방향 벡터 N과 L의 곱은 벡터의 내적을 의미하는데, 이 둘의 길이가 1이므로 두 방향의 각도에 대한 코사인 값은 내적 값과 같게 된다. 이 때 $I_{l\lambda}$와 $k_{d\lambda}$는 각 채널끼리 곱하고 상수 $\cos\theta$는 각 채널에 곱하게 된다. 난반사는 보통 $\theta$가 0도와

90도 사이의 값을 가질 경우에만 고려한다. 만약 이 값이 90도보다 크다면 이는 물체 표면의 뒤쪽에서 빛이 들어오는 것을 의미하므로, 대개의 경우 이 코사인 값을 0으로 처리하는데, 퐁의 모델을 어떻게 변형하는가에 따라 각도 $180 - \theta$에 대한 코사인 값을 적용하기도 한다.

난반사에 대하여 요약을 하면 물체가 순수하게 난반사를 할 경우 관찰자의 시점에는 상관없이 빛의 입사 각도에 대한 코사인 값, 광원의 밝기, 그리고 물체의 난반사 계수에 비례하여 빛을 반사한다. 한 가지 중요한 사실은 우리가 어떤 물체에 대한 반사 성질을 지정할 때 물체의 전반적인 색깔을 난반사 계수 $k_{d\lambda} = (I_{dR}\ I_{dG}\ I_{dB})^t$를 통하여 지정을 한다는 점이다. 즉 어떤 물체를 푸른색으로 하고 싶다면 바로 그 색깔을 난반사 계수로 사용을 하면 되는데, 따라서 이 계수 값이 다른 조명 인자들과 비교하여 물체의 전반적인 색깔에 가장 영향을 많이 미치는 인자라 할 수가 있다.

### 5.1.3 정반사

입사 광선을 사방으로 고르게 반사시키는 난반사와는 달리 정반사는 물체 표면으로 들어오는 빛을 특정 방향을 중심으로 집중적으로 반사를 하는 성질을 가진다. 정반사는 주로 자동차 표면이나 보석과 같은 금속, 또는 광택을 낸 사과와 같이 반짝거리는 물체를 표현하는데 사용이 된다. 이러한 물체들의 반사에 대한 특징중의 하나는 표면에 상대적으로 특별히 반짝거리는 부분이 생기게 되는데, 이러한 부분을 하이라이트(hightlight)라 한다. 거울은 정반사를 하는 극단적인 예라고 할 수 있는데 입사 광선을 입사각과 대칭이 되는 반사각 방향으로만 빛을 반사를 한다. 일반적으로 정반사를 많이 하는 반짝거리는 물체들은 거울과 같이 한 방향으로만 빛을 반사시키지는 않고, 특정 방향 둘레로 어느 정도의 범위를 가지고 집중적으로 빛을 반사시킨다.




(a) 네 가지 중요한 방향 (b) 난반사 및 정반사의 반사 형태

그림 3.11: 정반사 및 빛의 반사 형태

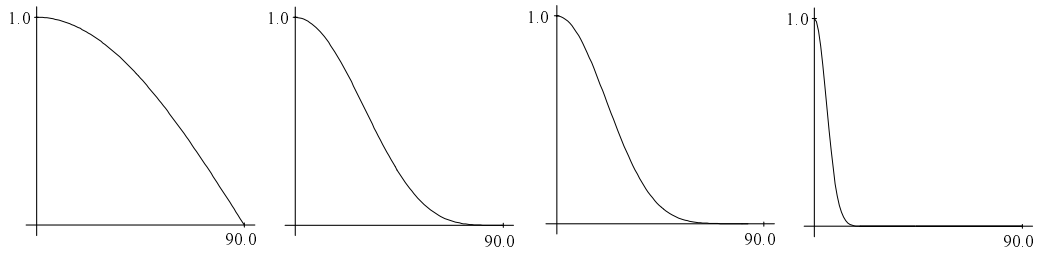
그림 3.11(a)를 보면 정반사에 영향을 미치는 네 개의 방향이 도시되어 있는데, 단위 벡터 L, V, 그리고 N의 의미는 앞에서와 같고, R 벡터는 L을 N 벡터를 중심으로 하여 반대 방향으로 반사를 시킨 방향을 나타낸다. 이 방향을 정반사 방향(specular reflection direction)이라고 하는데, 정반사의 경우 한 지점으로 들어오는 빛을 R 방향으로 가장 강하게 반사를 시키고, 방향이 R에서 벗어날 수록 반사되는 빛의 세기가 급격히 약해지게 된다. 그림 3.11(b)를 보면 난반사와 정반사의 반사 형태의 차이를 쉽게 이해할 수 있다. 물체와 광원을 고정시키고 관찰자 시점을 움직이면서 한 지점을 바라볼 때, 순수하게 난반사를 하는 물체의 경우에는 항상 같은 밝기의 색깔로 보이나, 정반사를 하는 물체의 경우에는 보는 시점에 따라 밝아졌다 어두워졌다 하게 된다. 특히 R 벡터 방향에서 바라볼 때 가장 밝게 보이게 되는데, 이를 거꾸로 생각을 해보면 우리가 정반사를 하는 물체를 바라볼 때, 표면의 각 지점에서의 R 벡터 방향과 관찰자를 향한 V 벡터 방향이 서로 비슷한 지역에 하이라이트가 생김을 알 수 있다.

이러한 정반사의 형태를 퐁이라는 사람이 코사인 함수를 사용하여 다음과 같은 실험적인 모델을 설정하였다.

$$\begin{aligned} I_\lambda &= I_{l\lambda} \cdot w(\theta, \lambda) \cdot \cos^n \phi \\ &\approx I_{l\lambda} \cdot k_{s\lambda} \cdot \cos^n \phi \\ &= I_{l\lambda} \cdot k_{s\lambda} \cdot (R \cdot V)^n \end{aligned}$$

여기서 $I_{l\lambda}$는 앞에서와 같이 광원의 색깔이고 $w(\theta, \lambda)$는 정반사 계수(specular reflection coefficient)로서 빛의 입사각 $\theta$와 파장 $\lambda$의 함수로 표현이 된다. 정반사 계수는 물질의 고유한 성질로서 들어오는 빛을 어떤 비율로 반사를 시킬 것인가를 결정하는데, 입사각이 90도일 경우 들어오는 빛을 전부 반사시키는 반면 90도보다 작을 경우에는 물질의 성질에 따라 반사되는 빛의 비율이 달라진다. 일반적으로 대부분의 불투명한 물체들에 대하여 이 값이 각도에 따라 크게 변하지 않기 때문에 상수 벡터인 $k_{s\lambda} = (I_{sR}\ I_{sG}\ I_{sB})^t$ 값으로 근사화한 모델을 사용한다. $\phi$는 정반사 방향 R과 시선의 방향 V와의 각도인데, R과 V를 단위 벡터라 할 경우 $\cos\phi$는 R과 V의 내적과 같은 값을 가지게 되므로 마지막에 있는 공식을 얻게 된다. 난반사의 경우와 다른 점은 코사인 값을 그냥 사용하는 것이 아니라 주어진 값 $n$에 대하여 $n$ 제곱을 한다는 사실이다. 이러한 값 $n$을 정반사 지수(specular reflection exponent)라 하는데, 이는 물체의 고유 성질로서 보통 1부터 수백 사이의 값을 가진다. 과연 정반사 모델에서 정반사 지수의 역할은 무엇일까?

앞에서도 강조한 바와 같이 물체가 정반사를 할 때 정반사 방향으로 빛을 가장 강하게 반사를 하고, 그 방향에서 벗어날수록 반사되는 빛의 세기가 약해진다고 하였다. 이러한 성질은 정반사를 위한 공식에서 $\cos\phi$에 의하여 표현이 되고 있는데,




(a) $n = 1$ (b) $n = 4$ (c) $n = 8$ (d) $n = 128$

그림 3.12: 함수 $\cos^n \phi$의 그래프

이 값에다 $n$ 제곱을 함으로써 약해지는 속도를 조절할 수 있게 된다. 그림 3.12를 보면 $\cos^n \phi$ 값이 $n$ 값에 따라 어떻게 변하는가를 알 수 있다. 잘 알다시피 코사인 값은 $\phi$ 값이 0도에서 90로 증가함에 따라 그 값이 1에서 0으로 점점 작아지므로, $n$ 값이 커질수록 $\cos^n \phi$ 값은 더 빠른 속도로 작아진다. 따라서 정반사 지수 $n$을 크게 하면 바라보는 방향이 정반사 방향에서 조금만 벗어나도 반사되는 빛의 밝기가 급격히 줄어들고, 그 결과 시점을 고정시키고 물체를 바라볼 때 밝게 보이는 하이라이트 지역의 크기가 작아진다. 다시 말해서 하이라이트의 크기를 크게 하려면 $n$값을 작은 값으로 설정하고, 반대로 작게 하려면 이 값을 크게 하면 되는데, 물론 여기에는 물리학적으로 정확한 어떤 수치가 있는 것이 아니라 프로그래머의 판단에 따라 정반사 지수 $n$ 값을 실험적으로 바꾸어 가며 가장 적절한 값을 선택하여야 한다.

정반사에 대하여 한 가지 더 고려를 해야 할 것은 많은 경우 정반사에 의해 반사되는 빛의 색깔은 물체의 고유 색깔보다는 주로 광원의 색깔에 좌우된다는 점이다. 예를 들어 검은색의 자동차 표면에 흰색 광원의 빛이 반사되는 장면을 상상해보면, 하이라이트가 하얗게 생김을 쉽게 알 수 있다. 따라서 정반사의 공식을 사용할 때 이러한 점을 고려하여 각 인자 값을 설정하여야 한다. 광원의 색깔 $I_{l\lambda}$ 값이 이미 이 공식에 있으므로 물체의 고유 색상으로 설정하는 난반사 계수 $k_{d\lambda}$와는 달리, 일반

적으로 $k_{s\lambda}$의 경우에는 0과 1사이의 적절한 값을 사용하여 광원에서 들어온 빛을 어떤 비율로 반사시킬 것인가를 조절하게 된다.

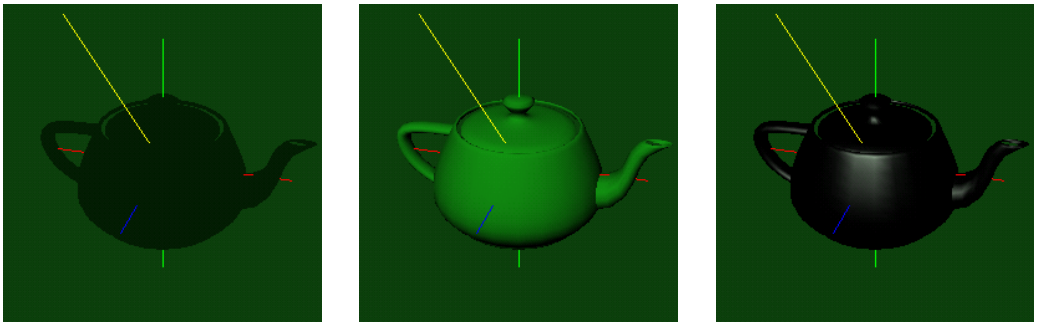
## 5.2 퐁의 조명 모델에 대한 변형

### 5.2.1 기본적인 퐁의 조명 모델

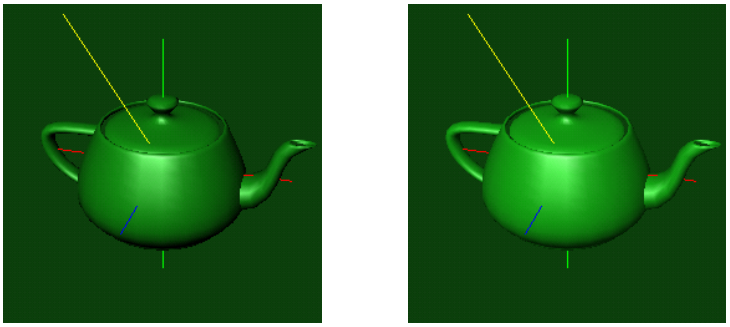
앞에서 순수하게 난반사를 하거나 또는 정반사를 하는 물체가 그 표면에서 빛을 어떻게 반사시키는지에 대하여 알아보았는데, 일반적으로 보통 물체들은 순수하게 한 가지 종류의 반사가 아니라 두 가지 모두를 적절히 혼합하여 빛의 반사 형태를 시뮬레이션하게 된다. 간접 조명의 효과를 위한 앰비언트 반사까지 포함하여 다음과 같은 기본적인 퐁의 조명 모델을 얻을 수 있다.

$$\begin{aligned} I_\lambda &= I_{a\lambda} \cdot k_{a\lambda} + I_{l\lambda} \cdot k_{d\lambda} \cdot (N \cdot L) + I_{l\lambda} \cdot k_{s\lambda} \cdot (R \cdot V)^n \\ &= I_{a\lambda} \cdot k_{a\lambda} + I_{l\lambda} \cdot \{k_{d\lambda} \cdot (N \cdot L) + k_{s\lambda} \cdot (R \cdot V)^n\} \end{aligned}$$

그림 3.13은 평행 광원을 사용하여 에머랄드로 만든 것처럼 보이도록 물주전자를 렌더링한 장면을 보여주고 있다. 여기서 비스듬한 직선은 빛이 들어오는 방향을 나타내는데, 앞에서도 설명한 바와 같이 앰비언트 반사는 빛의 방향에 상관없이 물체 전체에 대하여 아주 약하게, 그리고 고르게 반사함을 알 수 있다(그림 3.13(a)). 난반사의 경우에는 빛이 들어오는 방향과 물체의 법선 벡터 방향간의 각도가 작은 지점일수록 밝게 반사됨을 알 수 있으며, 특정 지역에 하이라이트가 생기는 것이 아니라 물체 표면의 색깔이 전체적으로 부드럽게 반사되어 보인다(그림 3.13(b)). 특히 난반사를 통하여 물체의 전반적인 색깔을 표현하고 있다. 한편 정반사의 경우 물주전자의 표면에 하이라이트를 만들어 마치 금속성의 물질로 만든 것과 같으 효과를



(a) 앰비언트 반사 (b) 난반사 (c) 정반사



(d) 난반사+정반사 (e) 세 가지 반사의 합

그림 3.13: 세 가지 종류의 반사

내려 하고 있다(그림 3.13(c)). 여기서 하이라이트는 물체의 표면을 바라보는 방향과 정반사 방향이 일치하는 지점을 주위로 생김을 알 수 있다. 그림 3.13(d)는 난반사와 정반사를 합쳐 렌더링을 한 것인데, 빛이 내려 쪼이는 부분과 그렇지 않은 부분의 명암 차이가 커서 약간 거친 느낌을 준다. 여기서 앰비언트 반사 효과까지 합하면 훨씬 자연스러운 모습을 보이게 된다(그림 3.13(e)).

일반적으로 이렇게 세 가지 종류의 반사를 더하여 물체의 쉐이딩이 된 색깔을 표현하게 되는데, 각 반사에 대하여 어떻게 가중치를 주는가에 따라 보이는 모습이 다르게 될 것이다. 예를 들어 물체 표면에서 난반사보다 정반사를 더 많이 하게 하여 반짝거리는 효과를 내고 싶으면 $k_{s\lambda}$의 값을 $k_{d\lambda}$보다 상대적으로 크게 하면 된다.

### 5.2.2 해프웨이 벡터

퐁의 조명 모델 공식을 계산하기 위해서는 정반사에 관련된 부분에 대하여 정반사 방향 R을 구해야 한다. 이 벡터는 $R = 2(N \cdot L)N - L$과 같이 됨을 쉽게 유도할 수 있는데, 이를 위해서는 5번의 덧셈/뺄셈 연산과 7번의 곱셈 연산을 수행하여야 한다. 이는 별로 안 되는 계산량으로 생각할 수 있으나, 복잡한 다면체 모델을 렌더링할 때에는 부담이 될 수도 있다. 앞에서도 언급한 바와 같이 OpenGL 렌더링 파이프라인에서는 눈 좌표계 공간에서 각 꼭지점에 대하여 퐁의 조명 모델을 적용하여 쉐이딩이 된 색깔을 계산한다고 하였다. 약간만 복잡한 장면을 렌더링하더라도 처리해야 할 꼭지점의 개수가 크게 증가하곤 한다. 만약 10만개의 꼭지점에 대하여 R 벡터를 계산한다면 120만 번의 부동 소수점 연산을 수행해야 하는데, 이는 상황에 따라 실시간 계산을 하는데 있어 부담이 될 수 있는 계산량이다.

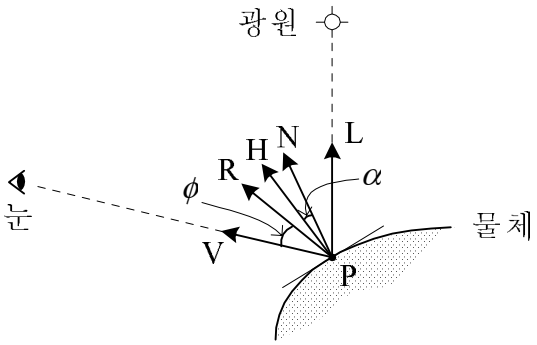


그림 3.14: 해프웨이 벡터

퐁의 조명 모델을 구현할 때 계산량을 줄이기 위한 시도로서 $H = \frac{L+V}{|L+V|}$와 같이 정의되는 해프웨이 벡터(halfway vector)라고 하는 벡터를 사용하여 정반사 공식의 $(R \cdot V)^n$을 $(N \cdot H)^n$으로 대치하여 계산을 한다. 그림 3.14에 도시된 바와 같이 해프웨이 벡터는 광원의 방향 L과 시선의 방향 V의 중간 방향으로서, 물체 표면의 법선 벡터 방향이 이 방향과 일치할 때 가장 강하게 정반사를 하는 방향이 된다.

과연 해프웨이 벡터를 사용하면 어떠한 이점이 있을까? 광원과 관찰자 시점이 렌더링을 하려는 물체로부터 충분히 멀리 떨어져 있다면, 평행 광원과 평행 투영을 해도 무방하고, 이러한 경우 L과 V는 상수 벡터가 된다. 이 때 H 벡터도 상수 벡터

가 되어 L과 V가 고정이 되었을 때, 한 번만 계산을 해주면 된다. R 벡터의 경우 일반적으로 꼭지점 마다 법선 벡터의 방향이 바뀌므로, 각 꼭지점 마다 다시 계산을 해주어야 하기 때문에, H가 상수 벡터인 상황에서는 상당한 양의 계산을 줄일 수가 있다.

한편 퐁의 조명 모델 공식을 계산할 때, $(R \cdot V)^n$을 $(N \cdot H)^n$으로 대치할 경우 원래 계산하려하던 값과 다른 값을 얻게 된다. 두 벡터 N과 H사이의 각도를 $\alpha$라 하면 L, V, R, N이 모두 한 평면에 있을 때 $\alpha = \frac{\phi}{2}$임을 보일 수가 있고, 만약 같은 평면상에 있지 않을 경우에는 $\alpha > \frac{\phi}{2}$와 같은 관계를 가진다. 해프웨이 벡터를 사용하면 원래 사용하던 $\cos\phi$ 대신에 $\cos\alpha$ 값에 대하여 $n$ 제곱을 하기 때문에 그 값이 변하게 된다. 하지만 이것은 심각한 문제는 아니다. 어짜피 이 공식은 실험적으로 구한 것으로서 만약 사용하는 각도가 커져 원래의 코사인 값보다 큰 값을 얻게 된다면, 정반사 지수를 약간 크게 하여 이를 실험적으로 조절하면 된다.

다시 한번 강조하지만 OpenGL은 실시간 렌더링을 위한 그래픽스 시스템이기 때문에 제한된 시간 안에 필요한 계산을 마치기 위해서 원래의 정확한 렌더링 모델을 상당히 단순화시켜 사용을 한다. 퐁의 조명 모델도 그러한 목적으로 단순화된 실험적인 모델중의 하나이고, 해프웨이 벡터도 그러한 관점에서의 시도라 할 수 있다. 해프웨이 벡터는 일반적으로 널리 쓰이는데 이 때의 조명 모델은 다음과 같다.

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda} + I_{l\lambda} \cdot \{k_{d\lambda} \cdot (N \cdot L) + k_{s\lambda} \cdot (N \cdot H)^n\}$$

### 5.2.3 광원의 거리에 따른 빛의 감쇠 효과

어두운 방에서 촛불을 들고 이리저리 움직이면서 물체 표면에 빛이 비추는 광경을 관찰해보면 촛불이 표면에 가까워질수록 밝게 보이고 멀어질수록 점점 어두워짐을

알 수 있다. 점 광원에서 사방으로 고루게 빛이, 즉 복사 에너지가 퍼져 나가갈 때 물체 표면에서 단위 면적 당 받는 에너지의 양은 광원과 물체 표면간의 거리의 제곱에 반비례하게 된다. 이러한 거리에 빛의 감쇠 효과(light attenuation effect)를 퐁의 조명 모델에 집어넣을 수 있으면, 좀 더 사실적인 조명 효과를 낼 수 있을 것이다. 광원에서 현재 쉐이딩을 하려하는 지점까지의 거리를 $d$라 할 때, 점 광원의 밝기 $I_{l\lambda}$에 $f_{att}(d) = \frac{1}{d^2}$을 곱하면 이러한 효과를 낼 수 있다. 어떻게 보면 자연스러운 공식이라 할 수 있으나, 이 함수의 그래프를 생각해보면 $d$값이 작을 때, 즉 광원이 물체에 가까이 있을 때는 광원의 움직임에 따라 밝기가 너무나 급격히 변하고, $d$값이 클 때, 즉 광원이 물체에서 멀리 떨어져 있을 때는 광원의 움직임의 효과가 거의 나타나지 않음을 쉽게 알 수 있다. 실제로 이 함수를 사용하여 렌더링을 할 때 $d$ 값을 적절히 정규화하지 않으면 종종 너무 어둡거나, 너무 밝거나 또는 감쇠 효과가 전혀 나타나지 않는 현상이 발생한다.

이러한 문제를 해결하기 위해 많은 그래픽스 시스템에서는 $f_{att}(d) = \frac{1}{k_0+k_1 \cdot d+k_2 \cdot d^2}$과 같은 형태의 함수를 사용을 한다. 여기서 세 개의 상수 $k_0$, $k_1$, $k_2$는 함수 $f_{att}(d)$가 적절한 범위의 값을 갖도록 실험적으로 선택을 하여 사용을 하면 된다. 이 때에도 이 값이 너무 작아져 전체 이미지가 어두워지는 것을 막기 위하여 $f_{att}(d) = min(\frac{1}{k_0+k_1 d+k_2 d^2}, 1)$과 같이 함수 값에 제한을 두기도 한다. 여하한 경우 광원의 거리에 따른 빛의 감쇠 효과를 고려하는 퐁의 조명 모델은 다음과 같다.

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda} + f_{att}(d) \cdot I_{l\lambda} \cdot \{k_{d\lambda} \cdot (N \cdot L) + k_{s\lambda} \cdot (N \cdot H)^n\}$$

### 5.2.4 다중 광원

광원이 한 개가 아니라 여러 개가 있을 때, 즉 다중 광원(multiple light sources)을 사용할 때에는 그러한 상황을 처리하기 위하여 퐁의 조명 모델에서 각 광원의 직접 조명 효과는 모두 더하고 전체 광원에 기인하는 간접 조명 효과는 앰비언트 광원 $I_{a\lambda}$를 통하여 조절을 한다. 광원이 0번 광원에서 $m-1$ 광원까지 $m$개가 있을 때의 퐁의 조명 모델은 다음과 같다.

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda} + \sum_{i=0}^{m-1} f_{att}(d_i) \cdot I_{l_i\lambda} \cdot \{k_{d\lambda} \cdot (N \cdot L_i) + k_{s\lambda} \cdot (N \cdot H_i)^n\} \quad (3.1)$$

여기서 첨자 $i$가 붙은 값들은 모두 광원의 색깔이나 위치, 방향에 영향을 받는 값들이다. 이 정도가 스폿 광원과 양면 조명 효과를 제외한 일반적으로 많이 쓰이는 퐁의 조명 모델의 한 형태인데, 스폿 광원과 양면 조명에 관련된 사항은 다음절에서 자세히 다루도록 하겠다.

# 제 6 절 OpenGL에서의 조명 계산

## 6.1 조명 모델의 역할

지금까지 조명 모델에 대한 기초적인 사항에 대하여 살펴보았는데, 이제부터 실제로 OpenGL 시스템에서 사용되는 조명 모델에 대하여 구체적으로 알아보도록 하겠다. 그림 3.15는 266쪽의 그림 3.2의 라이팅(*lighting*) 모듈을 좀 더 자세하게 보여주고 있다. 그림 3.2에 도시된 프리미티브의 조합에 관련된 계산은 물체 좌표계(OC)와 눈 좌표계(EC)에서 일어난다. 렌더링을 하려하는 기하 모델에 대한 여러 속성들을 기술해주는 OpenGL 함수를 수행시킬 경우, 그러한 속성들이 적절한 처리를 통하여 눈 좌표계에 다다른 후 프리미티브 조합(Primitive Assembly) 모듈에서 점, 선분, 다각형 등 해당 프리미티브 단위로 조합이 되어 렌더링 파이프라인의 다음 단계로 넘어간다고 하였다. 프리미티브들의 가장 기본이 되는 단위는 점으로서, 앞에서도 언급한 바와 같이 각 점에 대한 여러 관련 정보들이 관련 데이터(Associated Data) 모듈에서 하나로 묶여 프리미티브 별로 조합이 된다. 프로그램에서 OpenGL 함수를 수행을 시키면 처음에는 물체 좌표계에서의 꼭지점 좌표와 법선 벡터, 그리고 색깔, 텍스춰 좌표, 에지 플래그 등이 관련지어지는데, 이들이 프리미티브들이 조합 모듈에 다다르면 눈 좌표계 상에서의 꼭지점 좌표(꼭지점 좌표(Vertex Coords) 모듈)에 대하여 적절한 계산을 통하여 얻어진 색깔, (필요에 따라 변환된) 텍스춰 좌표, 에지 플래그 등의 데이터(관련 데이터 모듈)들이 하나로 묶여져 다음 단계로 흘러가게 된다.

이 절에서 다루려는 문제는 바로 라이팅 모듈에서의 구체적인 계산 과정인데, 이 모듈은 프리미티브들의 각 꼭지점에 연관지어질 색깔을 결정함을 목적으로 한다. 그림 3.15의 계산 과정은 그림 3.2에서의 라이팅 조합 모듈 부분을 좀 더 상세하

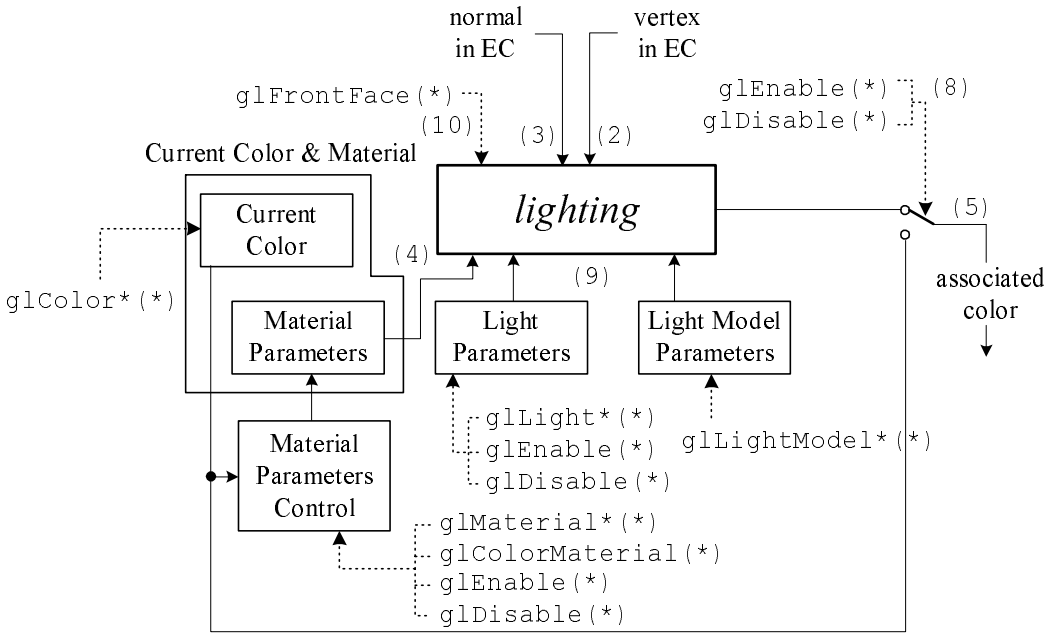



그림 3.15: OpenGL에서의 라이팅 계산 구조

게 나타낸 것인데, 바로 눈 좌표계에서 꼭지점에 관련지어질 색깔에 대한 계산 구조를 보여준다. 여기서 색깔은 크게 두 가지 방법에 의해 결정되는데, 그림 3.15의 (5)번 부분의 스위치를 보면 두 가지의 선택이 가능함을 알 수 있다[10]. 한 가지 선택은 라이팅 모듈의 계산 결과를 택하는 것인데, 이 경우에는 바로 퐁의 조명 모델을 사용하여 현재의 꼭지점에 대하여 쉐이딩을 한 색깔을 사용하게 된다. 다른 선택은 glColor*(*) 함수를 통하여 설정한 현재 색깔(Current Color)을 사용하여 꼭지점에 관련짓는 것이다. 다시 말해서 라이팅 계산을 통하여 얻어지는 색깔과 단순히 현재 색깔로 지정되어 있는 색깔 중 하나를 선택하여 꼭지점에 붙여줄 수가 있다. OpenGL에서는 디폴트로는 현재 색깔이 현재 꼭지점에 붙게 되고, 라이팅 계산을 수행하고 싶으면 glEnable(GL_LIGHTING);과 같은 문장을 수행시켜 명시적으로 그 기능을 선택하여야 한다((8)). 물론 다시 원래의 디폴트 상태로 돌아가고 싶

[10] 그림 3.2와 그림 3.15의 번호는 일치한다.

으면 glDisable(GL_LIGHTING);과 같은 문장을 수행시키면 된다. 여하한 경우 한 꼭지점에 색깔이 관련지어질 때에는 두 종류의 색깔이 붙여지게 된다. 현재 색깔이 선택이 될 때는 주 색깔(primary color)은 현재 색깔이 되고, 보조 색깔(secondary color)은 단순히 (0, 0, 0, 0)이 된다. 라이팅 계산을 할 때는 주어진 라이팅 인자들을 사용하여 적절하게 주 색깔과 보조 색깔을 계산하게 된다. 이 절에서는 바로 이러한 과정에서 프로그래머가 라이팅 계산을 하기 원할 때 OpenGL에서 어떤 조명 모델을 사용하여 각 꼭지점에 대하여 쉐이딩이 된 색깔을 계산하는지에 대하여 알아보려 한다.

## 6.2 라이팅 계산의 입력 인자

그림 3.15의 라이팅 모듈에서의 계산 과정을 알아보기 전에 이 모듈에 대한 입력 데이터, 라이팅 계산에 직접적으로 영향을 미치는 라이팅 인자(lighting parameter)들에 대하여 살펴보자. 그림을 보면 알 수 있듯이 이 모듈의 계산에 영향을 미치는 입력 데이터로 꼭지점 좌표(vertex in EC (2))와 법선 벡터(normal in EC (3)), 현재 색깔과 물질 인자(Current Color and Material Parameters (4)), 광원 인자와 조명 모델 인자(Light Parameters and Light Model Parameters (9)), 그리고 마지막으로 glFrontFace(*) 함수의 인자((10)) 등을 생각할 수 있는데, 이러한 입력 인자들에 대하여 기본적으로 다음과 같은 형태를 가지는 조명 공식을 사용하여 색깔 계산을 수행한다.

$$\begin{aligned}\mathbf{c}_{pri} &= \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \\ &\sum_{i=0}^{n-1} (att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}]\end{aligned}$$

이 공식은 300쪽의 식 (3.1)과 비교해보면 상당히 유사한 점이 있어 대략적으로 이해를 할 수 있겠으나, 약간은 다른 형태를 취하기 때문에 이에 대하여 자세하게 알아보도록 하겠다. OpenGL에서의 라이팅 모델 공식을 표현하기 위해 SGI의 자료에서 사용된 기호를 사용하려 하는데[11], 그에 대한 정의는 나올 때마다 설명을 하겠다.

### 6.2.1 glFrontFace(*) 함수

우선 라이팅 인자중 첫째로 void glFrontFace(GLenum mode); 함수에 대하여 살펴보자. 일반적으로 대부분의 그래픽스 시스템에서는 다면체 모델을 구성하는 다각형들이 일관된 방향성(orientation)을 가지기를 요구한다. 주어진 다면체 모델을 바깥쪽에서 바라본다고 할 때, 다각형의 꼭지점을 나열을 하면 시계 방향(clockwise)으로 회전하거나 반시계 방향(counterclockwise)으로 회전을 할 것이다. 일관된 방향성을 가진다는 것은 다면체 모델을 이루는 모든 다각형의 꼭지점을 나열을 하였을 때, 동일한 방향으로 회전을 한다는 것을 의미한다. 특별히 시계 방향 또는 반시계 방향 중 어떤 방향을 사용해야 한다는 법칙은 없으나, 항상 회전 방향이 모든 다각형에 대하여 일관성이 있도록 물체를 설계해야 한다. 이 OpenGL 함수는 바로 다면체가 어떠한 방향성를 가지는지를 설정하는데 사용이 되는데, 이 함수의 인자 mode가 GL_CW이면 다각형이 시계 방향, 그리고 GL_CCW이면 반시계 방향의 방향성을 가짐을 의미한다.

물체를 밖에서 바라볼 때 보이는 면을 앞면(front face), 그리고 안쪽에서 바깥쪽으로 바라볼 때 보이는 면을 뒷면(back face)이라 한다. 예를 들어 다면체가 반시계 방향의 방향성을 가질 경우 앞면은 화면에 투영이 되었을 때 꼭지점이 반시계 방

[11] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*, Silicon Graphics, Inc., 1998.

| 인자 | 타입 | 디폴트 값 | 설명 |
|---|---|---|---|
| $\mathbf{a}_{cm}$ | 색깔 | (0.2, 0.2, 0.2, 1.0) | 물질의 앰비언트 색깔 |
| $\mathbf{d}_{cm}$ | 색깔 | (0.8, 0.8, 0.8, 1.0) | 물질의 난반사 색깔 |
| $\mathbf{s}_{cm}$ | 색깔 | (0.0, 0.0, 0.0, 1.0) | 물질의 정반사 색깔 |
| $\mathbf{e}_{cm}$ | 색깔 | (0.0, 0.0, 0.0, 1.0) | 물질의 방사 색깔 |
| $s_{rm}$ | 실수 | 0.0 | 물질의 정반사 지수 (범위: [0.0, 128.0]) |

표 3.2: 물질 인자

향으로 회전을 한다. 만약 화면에 투영된 다각형의 꼭지점이 시계 방향으로 회전을 한다면 이는 다각형의 앞면이 아니라 뒷면을 보고 있음을 의미한다. OpenGL에서는 물체의 표면에 대한 조명 계산을 할 때 필요하다면 앞면과 뒷면을 다르게 보이게 할 수 있는데, 물체의 앞면과 뒷면은 바로 이 함수에 의하여 정의된 방향성을 사용하여 윈도우 좌표계에서 결정을 한다. 한편 닫힌 물체의 경우 그 물체가 불투명하다면 오직 앞면만 보이므로 굳이 뒷면에 해당하는 다각형에 대하여 불필요한 계산을 할 필요가 없다. 따라서 렌더링 파이프라인 과정에서 뒷면을 제거(backface culling)하도록 설정할 수 있는데, 이에 대해서는 뒤에서 설명을 하겠다.

### 6.2.2 물질 인자

다음 물질 인자(material parameter)에 대하여 살펴보면, 이 인자들은 glMaterial*(*) 함수를 사용하거나 지정하거나, glColor3*(*) 함수와 glColorMaterial(*) 함수를 같이 사용하여 지정할 수 있다. 우선 첫 번째 방법에 대하여 살펴보자. 물질 인자는 앞에서도 설명한 바와 같이 물체의 표면으로 들어온 빛이 어떻게 반사가 될 것인가를 결정하는 물질의 고유 성질로서, 물질 인자를 사용하여 렌더링하려는 물체가 원하는 물질처럼 보이도록 해주어야 한다. 표 3.2에는 OpenGL 조명 계산에서 사용되는 물질 인자가 요약이 되어 있다[12]. 여기서 실수 값을 가지는 $s_{rm}$을 제외한 나

[12] 이 책에서는 조명 계산을 할 때 색깔 인덱스 모드와 관련된 인자는 고려하지 않겠다.

머지 네 개의 인자의 타입은 색깔인데, 이는 이 인자들이 (R, G, B, A) 값을 가짐을 의미한다. 각 채널은 0과 1사이의 값으로서, 만약 계산 결과가 0보다 작으면 0으로, 그리고 1보다 크면 1로 변환이 된다. 각 인자들은 미리 디폴트 값으로 지정이 되기 때문에, 프로그램에서 명시적으로 지정을 하지 않으면 디폴트 값들이 사용이 된다. $\mathbf{a}_{cm}$, $\mathbf{d}_{cm}$, $\mathbf{s}_{cm}$, $s_{rm}$은 각각 앞에서 개념적으로 살펴본 퐁의 조명에 대한 식 (3.1)에서 $k_{a\lambda}$, $k_{d\lambda}$, $k_{s\lambda}$, $n$에 대응이 되는 값임을 쉽게 알 수 있다[13]. 한 가지 특이한 것은 OpenGL에서는 물체가 스스로 빛을 발하는 것처럼 보이도록 물질의 성질을 설정할 수 있는데, 이러한 성질은 바로 $\mathbf{e}_{cm}$ 인자를 통해서 설정이 가능하다. 일반적으로 대부분의 물질들은 스스로 빛을 발하지 않기 때문에 이 인자는 자주 사용되지는 않지만, 자동차의 헤드라이트와 같이 스스로 빛을 발하는 물체를 표현하는데 유용하게 쓰일 수가 있다.

이 다섯 가지의 물질 인자는 다음 두 함수에 의해 설정이 가능한데,

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

여기서 실수인 $s_{rm}$ 경우에만 첫 번째 함수로 지정할 수 있다. 조명 계산을 할 때 필요한 경우 물체의 앞면과 뒷면을 다르게 보이게 할 수 있다고 하였는데, 이 함수를 사용하여 물체 표면의 반사 성질을 각 면에 대하여 다르게 설정할 수가 있다. 첫 번째 인자 face는 GL_FRONT, GL_BACK, GL_FRONT_AND_BACK 중의 값을 가지는데, 이는 현재 어떤 면의 물질 성질을 설정하고자 하는지를 나타낸다. 다음 두 번째 인자 pname은 위의 다섯 개 물질 인자 중 어떤 인자의 값을 설정할 것인가를 나타낸다. 표 3.2의 순서대로 나열할 경우 대응되는 OpenGL 상수는 각각 GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS이다. 그리고 많은 경

[13] 여기서 사용하는 변수의 이름 $\mathbf{a}$, $\mathbf{d}$, $\mathbf{s}$와 $s$, $\mathbf{e}$는 각각 ambient, diffuse, specular, emission의 첫 글자를 따온 것이다. 그리고 첨자의 $c$는 그 변수가 색깔(color) 타입, $r$은 실수(real) 타입을 의미하고 $m$은 물질(material)의 성질을 나타내는 인자라는 것을 의미한다.

우 $\mathbf{a}_{cm}$과 $\mathbf{d}_{cm}$을 물체의 기반이 되는 색깔로 동일하게 지정을 하는데, 이럴 경우 GL_AMBIENT_AND_DIFFUSE를 사용하면 한 번에 두 값을 설정할 수 있다. 마지막으로 세 번째 인자 param은 실제로 설정할 값을 나타내는데, 다른 OpenGL 함수들과 마찬가지로 함수 이름에 v가 붙어 있는지에 따라 실제 변수인지 포인터인지가 결정된다. 물론 함수 이름에 i가 들어 있으면 GLint(32 비트 정수) 타입, 그리고 f가 들어 있으면 GLfloat(32 비트 부동 소수점) 타입의 값을 가진다.

예를 들어 물체가 에머랄드처럼 보이게 하려면 다음과 같이 물질 인자를 설정하면 된다.

```
GLfloat mat_ambient[4] = {0.03784, 0.30712, 0.03784, 1.0};
GLfloat mat_diffuse[4] = {0.07568, 0.61424, 0.07568, 1.0};
GLfloat mat_specular[4] = {0.633, 0.727811, 0.633, 1.0};
GLfloat mat_shininess = 22.0;
      ⋮
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);
```

여기서 사용된 수치들은 보통 실험적으로 얻어진 값들인데, 종종 실제로 화면에 보이게 될 물체의 색깔과 직관적으로 잘 연결이 되지 않기 때문에, 이러한 값들을 원하는 방식으로 조절하는 것이 그리 수월하지는 않다. 물질 인자를 설정할 때의 몇 가지 특징을 살펴보면 다음과 같다. 우선 첫째로 가장 중요한 인자는 난반사 색깔 $\mathbf{d}_{cm}$으로서 앞에서도 설명한 바와 같이 이 값이 물체의 기본이 되는 색깔이다. 위에서의 예제 코드에서도 에머랄드 색을 표현하기 위하여 기저가 되는 난반사 색깔로 연두색 계통의 (0.07568, 0.61424, 0.07568, 1.0)를 사용하였다. OpenGL에서는 기본적으로 RGBA 모델을 사용하기 때문에 조명 계산을 할 때 알파 채널에 해당하는 A 값을 어떻게 구할 것인가 하는 것이 중요한 사항 중의 하나인데, 바로 물체의 난반

사 색깔의 A 값이 조명 계산의 결과 산출되는 쉐이딩 된 색깔의 A 채널 값으로 설정된다. 따라서 어떤 물체를 불투명도 0.3을 가지는 약간 투명한 물체처럼 보이도록 하려면 $\mathbf{d}_{cm}$의 네 번째 값을 0.3으로 설정하면 된다. 한편 앰비언트 색깔 $\mathbf{a}_{cm}$은 일반적으로 $\mathbf{d}_{cm}$과 같은 값을 갖도록 설정이 된다. 만약 다른 값을 가질 경우에도 $\mathbf{d}_{cm}$의 색깔과 전체적으로 조화를 이루도록 설정을 하는데, 위의 예에서도 그러한 방식으로 앰비언트 색깔이 설정이 되었다. 마지막으로 정반사 색깔 $\mathbf{s}_{cm}$은 앞에서도 설명한 바와 같이 물체의 기본 색깔을 반영한다기보다는 들어오는 빛을 대체로 어느 정도의 비율로 정반사시킬 것인가를 나타낸다. 여기서도 그러한 맥락에서 전체적으로 회색 계통의 비율을 사용했음을 알 수 있다.

물질 인자는 다른 OpenGL 인자처럼 상태 변수로 OpenGL 시스템에 존재한다. 어떤 물체를 그리기 위하여 glBegin(*)과 glEnd() 함수 사이에서 물체의 기하 정보를 기술할 때, 각 꼭지점이 눈 좌표계로 변환이 되어 그 때 설정이 되어 있는 물질 인자들을 비롯한 라이팅 인자들을 사용하여 조명 계산을 한다. 따라서 물체를 그리라는 명령을 내리기 전에 필요한 물질 인자들을 적절히 설정해주어야 한다. 또한 여러 개의 물체를 그린다면 '인자 설정-물체 그리기' 과정을 반복하면 된다. 이 때 모든 인자를 다시 설정할 필요는 없고, 현재 상태에서 반드시 바뀌어야 하는 인자만 다시 설정하는 것이 효율적임은 두 말할 여지가 없다.

렌더링을 할 때 상황에 따라서 물체의 색깔 인자 중 특정 색깔을 자주 바꾸어야 하는 경우가 있다. 예를 들어 물체의 난반사 색깔만 조금씩 바꾸어 가면서 렌더링을 하고 싶을 때, void glColorMaterial(GLenum face, GLenum mode); 함수를 사용하면, 사용하는 OpenGL 라이브러리의 구현에 따라 더 빠르게 조명 계산을 할 수가 있다. 여기서 인자 face는 glMaterial*(*) 함수와 같은 값을 가질 수 있고, mode 인자는 GL_SHININESS를 제외한 GL_AMBIENT_AND_DIFFUSE를 포함하여 다섯 가

지 색깔 타입의 상수 값을 가질 수 있다. 이 함수의 작동 원리는 다음과 같다. 우선 GL_COLOR_MATERIAL 상수를 사용하여 glEnable(GL_COLOR_MATERIAL); 문장을 수행하여 이 기능을 사용하겠다고 명시적으로 선언을 해준다. 다음 glColorMaterial(*) 함수를 사용하여 앞면, 뒷면, 양면 중 어느 면의 어떤 물질 색깔을 바꾸려 하는지를 설정한다. 다음 물체를 그릴 때 glColor*(*) 함수를 사용하여 원하는 물질의 성질을 동적으로 바꾸면 된다. 마지막으로 이 기능이 더 이상 필요 없으면 glDisable(GL_COLOR_MATERIAL); 문장을 수행시키면 된다. 아래의 코드는 이 함수를 사용하여 물체의 난반사 색깔을 조금씩 바꾸어 가며 렌더링을 하는 예를 보여주고 있다.

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
for (i = 65; i < 256; i++) {
  mat_diffuse[1] = i/255.0;
  glColor4fv(mat_diffuse);
  draw_scene();
}
glDisable(GL_COLOR_MATERIAL);
```

여기서는 물체의 난반사 색깔 중 초록색 채널을 조금씩 바꾸어 가며 렌더링을 하고 있다. glColor4fv(mat_diffuse); 문장을 수행하여 현재 색깔을 새롭게 설정할 때마다 물체의 난반사 색깔도 이 색깔로 바뀌게 된다. glColor*(*) 함수는 기하 물체의 꼭지점에 관련되어 빠르게 구현이 되는 함수이다. 어떤 물체의 색깔을 꼭지점 단위로 바꿀 경우 glColorMaterial(*) 함수를 통하는 것이 glMaterial*(*) 함수보다 빠르게 수행이 될 가능성이 많지만, 이는 사용하는 OpenGL 구현에 그렇지 않을 수도 있음을 명심하기 바란다.

### 6.2.3 광원 인자

5절에서 살펴본 퐁의 조명 모델에서는 난반사와 정반사에 대한 색깔을 계산할 때 각 부분에 영향을 미치는 광원의 색깔을 동일하게 하였었다. 즉 앰비언트 반사에 대한 광원 색깔 $I_{a\lambda}$과 난반사 및 정반사에 대한 광원 색깔 $I_{l\lambda}$ 두 가지만 사용하여 조명 계산을 하였다. OpenGL 시스템에서는 광원의 색깔을 좀 더 세밀하게 조절을 할 수가 있다. 특히 앰비언트 반사에 대하여 모든 광원이 종합적으로 영향을 미치는 앰비언트 광원 색깔과 각 광원이 물체에 직접적으로 영향을 미치는 앰비언트 광원의 색깔을 다르게 할 수가 있다. 또한 각 광원이 앰비언트 반사, 난반사, 정반사에 대하여 서로 다른 색깔로 영향을 미치게 할 수가 있다. 303쪽의 OpenGL의 조명 계산에 사용하는 공식을 보면 이러한 특징을 이해할 수 있는데, 식 앞부분의 $\mathbf{a}_{cm} * \mathbf{a}_{cs}$에서 $\mathbf{a}_{cs}$가 광원들이 장면 전체에 종합적으로 영향을 미치는 간접 조명 효과를 고려하기 위한 광원의 색깔로서, OpenGL에서는 이를 전역 앰비언트 반사(global ambient reflection)라 한다. 반면 각 광원에 대하여 더해지는 부분을 보면 $\mathbf{a}_{cli}$는 각 광원에 기인하는 지역 앰비언트 반사(local ambient reflection)를 위한 앰비언트 광원의 색깔이고, 난반사와 정반사에 대하여 각각 $\mathbf{d}_{cli}$와 $\mathbf{s}_{cli}$의 색깔을 사용하여 필요할 경우 서로 다른 색깔을 설정할 수 있도록 해주고 있다.

#### 광원의 색깔 및 위치

표 3.3에는 OpenGL에서 사용되는 광원 인자(light source parameter)가 요약되어 있다. 우선 $\mathbf{a}_{cli}$, $\mathbf{d}_{cli}$, $\mathbf{s}_{cli}$는 방금 설명한 바와 같이 $i$번째 광원이 각각 지역 앰비언트 반사, 난반사, 그리고 정반사에 직접적으로 영향을 미치는 광원의 색깔에 해당한다. 여기서 첨자 $c$는 앞에서와 같이 해당 변수가 색깔 타입의 값을 가짐을 나타내고, $l$과 $i$는 $i$번째 광원(light source)에 대한 인자임을 의미한다. OpenGL에서는 관

| 인자 | 타입 | 디폴트 값 | 설명 |
|---|---|---|---|
| $\mathbf{a}_{cli}$ | 색깔 | (0.0, 0.0, 0.0, 1.0) | $i$번 광원의 앰비언트 색깔 |
| $\mathbf{d}_{cli}(i=0)$ | 색깔 | (1.0, 1.0, 1.0, 1.0) | 0번 광원의 난반사 색깔 |
| $\mathbf{d}_{cli}(i>0)$ | 색깔 | (0.0, 0.0, 0.0, 1.0) | $i$번 광원의 난반사 색깔 |
| $\mathbf{s}_{cli}(i=0)$ | 색깔 | (1.0, 1.0, 1.0, 1.0) | 0번 광원의 정반사 색깔 |
| $\mathbf{s}_{cli}(i>0)$ | 색깔 | (0.0, 0.0, 0.0, 1.0) | $i$번 광원의 정반사 색깔 |
| $\mathbf{P}_{pli}$ | 위치 | (0.0, 0.0, 1.0, 0.0) | $i$번 광원의 위치 |
| $\mathbf{s}_{dli}$ | 방향 | (0.0, 0.0, -1.0) | $i$번 광원의 스폿 광원 방향 |
| $s_{rli}$ | 실수 | 0.0 | $i$번 광원의 스폿 광원 지수 (범위: [0.0, 128.0]) |
| $c_{rli}$ | 실수 | 180.0 | $i$번 광원의 스폿 광원 절단 각도 (범위: [0.0, 90.0] 또는 180.0) |
| $k_{0i}$ | 실수 | 1.0 | $i$번 광원의 상수 감쇠 인자 (범위: [0.0, $\infty$]) |
| $k_{1i}$ | 실수 | 0.0 | $i$번 광원의 1차 감쇠 인자 (범위: [0.0, $\infty$]) |
| $k_{2i}$ | 실수 | 0.0 | $i$번 광원의 2차 감쇠 인자 (범위: [0.0, $\infty$]) |

표 3.3: 광원 인자

례상 0번부터 시작하여 0번 광원, 1번 광원, ···처럼 광원의 이름을 붙인다. OpenGL 규약에는 OpenGL 시스템을 구현하기 위해서 최소한 8개의 광원을 지원하여야 한다고 되어있는데, 구현에 따라 그 이상의 광원을 지원할 수 있다. 각 반사에 영향을 미치는 광원의 색깔에는 이 표에서 설명이 된 바와 같이 미리 디폴트 값으로 설정이 되어 있는데, 난반사 색깔 $\mathbf{d}_{cli}$와 정반사 색깔 $\mathbf{s}_{cli}$의 경우 0번 광원과 기타 광원들은 서로 다른 초기 값을 가짐을 알 수 있다.

광원의 색깔과 함께 또 다른 중요한 인자로 광원의 위치나 빛을 비추는 방향과 같은 기하 속성을 들 수가 있는데, $\mathbf{P}_{pli}$ 변수가 그러한 정보를 저장한다. 여기서 $\mathbf{P}$는 변수의 위치(Position)를 의미하고, 첨자 $p$는 이 변수가 '넓은 의미'의 위치(position) 타입의 값을 가짐을 나타낸다. $\mathbf{P}_{pli}$는 광원의 기하 속성을 나타내기 위하여 $(x\ y\ z\ w)^t$와 같이 3차원 공간에서의 동차 좌표계를 사용한다. 표 3.3에 있는 광원 인자를 살펴보면 사용하는 광원이 점 광원인지, 평행 광원인지, 또는 스폿 광

원인지를 명시적으로 설정할 수 있는 인자가 존재하지 않는다. 그러면 어떻게 광원의 종류를 구별할 수 있을까? 우선 점 광원인지 평행 광원인지는 $\mathbf{P}_{pli}$를 사용하여 구별한다. 앞에서 설명한 바와 같이 점 광원의 기하 속성은 광원의 위치에 해당하는 유클리드 공간의 한 점에 의하여 기술이 되고, 평행 광원은 빛이 평행하게 들어오는 방향, 즉 벡터에 의하여 기술이 된다. 점과 벡터는 유클리드 공간에서는 서로 다른 개념인 것처럼 보이나, 투영 공간에서는 구별이 없는 동일한 개념으로서, 동차 좌표 $(x\ y\ z\ w)^t$라는 하나의 틀을 사용하여 동시에 표현을 할 수 있다고 설명을 하였다. 만약 $\mathbf{P}_{pli}$의 $w$ 좌표가 0인 경우 광원의 위치는 $(x\ y\ z\ 0)^t$, 즉 무한대 점에 존재하므로 유클리드 공간에서 $(-x\ -y\ -z)^t$인 방향으로 평행하게 비추는 평행 광원으로 정의가 된다. 반면에 $w$가 0이 아니면, 이 광원은 유클리드 공간에서 $w$를 1로 만들었을 때의 점 $(x\ y\ z)^t$에서 사방으로 빛을 비추는 점 광원으로 정의된다.

표 3.3에 있는 인자들은 모두 void glLight{if}(GLenum light, GLenum pname, TYPE param); 함수와 void glLight{if}v(GLenum light, GLenum pname, TYPE *param); 함수를 사용하여 설정할 수 있다. 여기서 첫 번째 인자 light는 광원의 이름을 지정하는데 사용이 되는데, $i$번째 광원은 상수 GL_LIGHTi를 사용하여 설정할 수 있다. 두 번째 인자 pname은 어떤 광원 인자의 값을 설정할 것인가를 지정하는데, 위에서의 $\mathbf{a}_{cli}$, $\mathbf{d}_{cli}$, $\mathbf{s}_{cli}$, $\mathbf{P}_{pli}$에 해당하는 상수는 각각 GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION이다. 마지막으로 인자 param은 다른 함수들에서와 같이 함수 이름에 v가 있는가에 따라 실제 변수 또는 포인터를 사용하는데, 위의 네 개의 인자의 경우에는 항상 인자 값이 저장된 배열의 포인터가 와야 한다.

다음 코드는 0번 광원을 설정하는 OpenGL 프로그래밍의 예를 보여주고 있는데, 이 광원은 위치가 $(0.0\ 5.0\ 5.0)^t$인 점 광원으로서 앰비언트 색깔은 아주 희미한 흰색((0.23, 0.23, 0.23)), 그리고 난반사와 정반사 색깔은 아주 밝은 흰색((0.95, 0.95,

0.95))으로 설정이 되고 있다.

```
GLfloat light_position[4] = {0.0, 5.0, 5.0, 1.0};
GLfloat light_ambient_color[4] = {0.23, 0.23, 0.23, 1.0};
GLfloat light_diff_spec_color[4] = {0.95, 0.95, 0.95, 1.0};
      ⋮
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient_color);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diff_spec_color);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_diff_spec_color);
```

### 스폿 광원 효과

다음, 표 3.3에 있는 인자 중 $\mathbf{s}_{dli}$, $s_{rli}$, $c_{rli}$는 스폿 광원의 효과를 내기 위한 인자들이다. 스폿 광원의 방향 인자 $\mathbf{s}_{dli}$의 $\mathbf{s}$는 스폿(spot)을 나타내고, 첨자 $d$는 방향(direction)을 나타낸다. 따라서 이 인자는 $(x\ y\ z)^t$와 같은 형태의 벡터 값을 가진다. 스폿 광원 지수와 절단(cutoff) 각도를 나타내는 $s_{rli}$와 $c_{rli}$는 실수(real) 타입의 값을 가진다. 앞에서도 설명한 바와 같이 스폿 광원은 점 광원의 특수한 경우로 간주된다. 즉 일반적인 점 광원은 사방으로 빛을 비추는 반면 스폿 광원은 특정 방향으로만 집중적으로 빛을 비춘다.

그림 3.16은 OpenGL에서 스폿 광원에 영향을 미치는 인자들의 의미를 보여주고 있다. 여기서 $\mathbf{P}_{pli}$는 $i$번째 광원의 위치를 나타내는데, 이 점에서 $\mathbf{s}_{dli}$ 방향으로의 직선을 생각하자. 이 때 스폿 광원은 이 직선의 둘레로 절단 각도 $c_{rli}$ 범위 내로만 빛을 비추기 때문에, 그림에서처럼 원뿔 형태로 빛을 발하게 된다. 표 3.3에서 보듯이 이 값은 0과 90도 사이의 값을 갖거나 180도로 설정이 된다. 만약 이 인자의 값이 180도라면, 이는 광원이 사방으로 빛을 비추는 것을 의미하기 때문에, 스폿 광원의 효과가 없는 일반 점 광원으로 취급이 된다. $c_{rli}$의 디폴트 값은 180도이므로, 이 인자를 명시적으로 다른 값으로 설정하지 않는 한 스폿 광원 효과는 나타나지 않는

다.

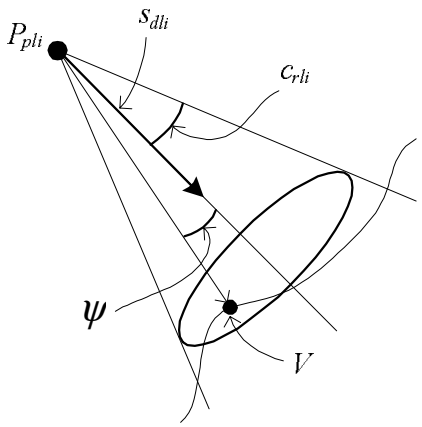



그림 3.16: 스폿 광원 인자

다시 말해서 $c_{rli}$ 인자가 0과 90도 사이의 값을 가질 경우에 마치 무대 조명으로 비추는 것과 같은 효과를 낼 수 있다. 그 경우 $c_{rli}$ 각도 범위 내의 물체만 밝게 보이고 그밖에 존재하는 물체의 표면은 이 광원에 의해 직접적인 영향을 받지 않게 된다. 이 때 또 다른 인자 $s_{rli}$는 이 스폿 광원에 의해 밝게 보이는 부분의 밝기를 조절하는데 사용된다. 무대 조명이나 손전등과 같은 조명을 생각해보면 일반적으로 중심 부분이 가장 밝고 밖으로 벗어날수록 약간씩 어두워짐을 알 수 있다. OpenGL에서는 이 광원의 기본 밝기 또는 색깔은 다른 광원에서와 같이 $\mathbf{a}_{cli}$, $\mathbf{d}_{cli}$, $\mathbf{s}_{cli}$에 의하여 설정이 된다. 이 밝기가 바로 $\mathbf{s}_{dli}$ 방향으로의 밝기가 되고 주변으로 갈수록 조금씩 어두워지는데 그 밝기는 다음과 같이 계산이 된다. 현재 조명 계산을 하려는 물체의 꼭지점의 위치를 $V$라 하자[14]. 이 점이 스폿 조명의 범위에 들어온다고 할 때, $\mathbf{P}_{pli}$에서 이 점으로 연결한 직선과 $\mathbf{s}_{dli}$ 방향과의 각도를 $\psi$라 하자. 이 때 이 지점으로 들어오는 빛의 밝기는 원래의 $\mathbf{a}_{cli}$, $\mathbf{d}_{cli}$, $\mathbf{s}_{cli}$ 밝기로부터 $\cos^{s_{rli}}\psi$에 비례해서 어두워진다. 즉 스폿 조명의 중심으로부터 벗어난 각도 $\psi$에 대하여 코사인을 취하므로 이 각도가 커질수록, 즉 중심으로 많이 벗어날수록 더 어두워진다. 또한 코사인 값은 0과 1 사이의 값을 가지므로 여기에 스폿 광원 지수를 사용하여 $s_{rli}$ 제곱을 하면 더 빠르게 어두워짐을 알 수 있다. 따라서 이 인자를 사용하면 스폿 광원의 주변으로 갈수록 얼마나 빠르게

[14] 5절에서 $V$는 조명 계산을 하려는 점에서 관찰자로의 방향을 의미했다. 여기서는 OpenGL 규약에서 사용하는 기호를 그대로 사용하기 위하여 물체의 꼭지점 좌표를 $V$로 표시하겠다.

어두워지게 할지를 조절할 수가 있다. $\mathbf{s}_{dli}$, $s_{rli}$, $c_{rli}$에 해당하는 OpenGL 상수는 각각 GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF로서 다음 코드는 0번 광원을 스폿 광원으로 설정하는 예를 보여주고 있다.

```
GLfloat light_position[4] = {0.0, 5.0, 5.0, 1.0};
GLfloat spot_direction[3] = {1.0, 0.0, 1.0};
GLfloat light_ambient_color[4] = {0.23, 0.23, 0.23, 1.0};
GLfloat light_diff_spec_color[4] = {0.95, 0.95, 0.95, 1.0};
      ⋮
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient_color);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diff_spec_color);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_diff_spec_color);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 3.0);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 30.0);
```

### 빛의 감쇠 효과

마지막으로 표 3.3에 있는 나머지 광원 인자 $k_{0i}$, $k_{1i}$, $k_{2i}$에 대하여 알아보자. 이 인자는 광원과 물체간의 거리에 따른 빛의 감쇠 효과를 위한 인자들이다. 광원과 현재 쉐이딩을 하려는 물체의 꼭지점까지의 거리를 $d(V)$라 하면 감쇠 효과를 내기 위하여 앞에서 설명한 바와 같이 $\frac{1}{k_{0i}+k_{1i}\cdot d(V)+k_{2i}\cdot d(V)^2}$과 같은 형태의 이차식을 사용한다. 만약 광원이 점 광원이라면 그 광원에 대하여 설정된 색깔에 이 값을 곱해 빛의 감쇠 효과를 낸다. 만약 광원이 무한대의 점에서 설정된 평행 광원이라면 빛의 감쇠 효과가 의미가 없으므로, 이 값 대신에 1.0 값을 사용하게 된다. 디폴트로 $k_{0i} = 1.0$, $k_{1i} = k_{2i} = 0.0$으로 설정이 되는데, 이는 조명 계산시 기본적으로 빛의 감쇠 효과를 계산하지 않음을 의미한다. 이 세 가지 인자에 대한 OpenGL 상수는 각각 다음과 같은데,

GL_CONSTANT_ATTENUATION,
GL_LINEAR_ATTENUATION,
GL_QUADRATIC_ATTENUATION,

다음은 이들에 대한 OpenGL에서의 설정 과정을 보여준다.

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 1.8);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.4);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.3);
```

### 6.2.4 기하 속성을 가지는 광원 인자의 설정

광원 인자 중 광원의 위치 또는 방향을 설정하는 $\mathbf{P}_{pli}$와 스폿 광원의 방향을 의미하는 $\mathbf{s}_{dli}$는 다른 광원 인자들과는 달리 광원의 기하학적인 성질을 기술하는데 사용이 된다. 앞에서도 자세하게 살펴본 바와 같이 OpenGL의 기하 파이프라인은 여러 개의 좌표계로 구성이 된다. 이 두 인자들의 값은 어떤 좌표계를 기준으로 하여 설정되는가에 따라 그 의미가 전혀 달라지기 때문에, 이들이 정의되는 좌표계를 분명히 해야 한다. 과연 OpenGL 프로그램에서 glLight*(*) 함수를 사용하여 이 두 인자의 값을 기술하면, 이 값들은 어느 좌표계에서의 의미를 가질까? 조명 계산은 기하 물체들이 눈 좌표계로 변환이 된 후, 이 좌표계에서 각 꼭지점에 대하여 수행이 된다고 하였다. 따라서 조명 계산을 하기 위해서는 눈 좌표계 또는 그 이전의 좌표계, 즉 물체 좌표계에서 설정이 되어야 한다. 앞에서는 편의상 물체 좌표계를 모델링 좌표계와 세상 좌표계로 분리하여 생각을 하였고, 그러한 관점에서 본다면 광원의 기하 속성은 모델링 좌표계, 세상 좌표계, 눈 좌표계 중 한 좌표계에서 설정이 된다고 생각할 수가 있다.

OpenGL 시스템에서는 glLight*(*) 함수를 통하여 $\mathbf{P}_{pli}$와 $\mathbf{s}_{dli}$의 값을 설정하면 바로 그 순간에 모델뷰 행렬 스택의 탑에 있는 모델뷰 행렬 $M_{MV}$를 사용하여 이 값들

을 눈 좌표계로 변환을 한 후 그 값들을 조명 계산에 사용을 한다. 따라서 프로그램 내에서 이 함수를 호출하는 위치가 중요한데, 구현하려고 하는 광원의 성질에 따라 모델링 변환과 뷰잉 변환을 적절하게 해주어야 한다. glLight*(*) 함수를 호출할 때, 만약 모델뷰 스택의 탑에 단순히 단위 행렬이 올려져 있다면($M_{MV} = I$), 이 함수의 인자 값은 눈 좌표계에서의 의미를 가질 것이다. 만약 뷰잉 행렬이 올려져 있으면($M_{MV} = M_V$), 그 값은 세상 좌표계에서의 값이 되고, 마지막으로 모델링 변환까지 올려져 있다면($M_{MV} = M_V \cdot M_M$), 그 값들은 해당 모델링 좌표계에서의 의미를 가지게 된다.

**프로그램 예 3.1** 광원의 기하 속성의 설정.

```
GLfloat light_position[4] = {0.0, 5.0, 0.0, 1.0};
      ⋮
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(20.0, 1.0, 1.0, 100.0);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// Line (a)
gluLookAt(4.0, 8.0, 14.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
// Line (b)
glTranslatef(0.0, -1.5, 0.0);
glRotatef(-90.0, 1.0, 0.0, 0.0);
// Line (c)
glBegin(GL_TRIANGLES);
      ⋮ // draw objects
glEnd();
```

위의 예제 프로그램은 뷰잉과 관련한 간단한 OpenGL 코드인데, 지금 `glLightfv (GL_LIGHT0, GL_POSITION, light_position);`과 같은 문장을 통해서 점 광원의

위치를 설정하려고 한다. 만약 Line (a)의 위치에서 이 문장을 수행시킨다면 배열 `light_position`에 저장된 점 $(0.0\ 5.0\ 0.0)^t$은 눈 좌표계에서의 좌표가 된다. 따라서 이와 같이 설정되는 광원은 카메라에 대하여 고정이 되는데, 이 경우에는 항상 화면의 바로 위쪽에서 빛을 비추는 효과가 있게 된다. 이 광원의 위치는 Line (a)와 Line (b) 사이의 `gluLookAt(*)` 함수에 의해 설정되는 뷰잉 변환에 영향을 받지 않기 때문에, 이 함수를 통하여 카메라의 위치와 방향을 바꿔 가면서 카메라에 대한 애니메이션을 하더라도 광원은 항상 카메라의 바로 위에서 빛을 비추게 될 것이다. 이러한 광원을 통해서 광부들이 착용하는 모자에 붙어 있는 전구처럼 카메라에 대하여 상대적으로 고정이 되는 헤드라이트 효과를 낼 수 있다. 만약 Line (b)에 광원의 위치를 설정한다면 이는 세상 좌표계의 점 $(0.0\ 5.0\ 0.0)^t$에 광원을 배치하는 결과를 낳게 된다. 다시 말해서 전자는 상대적으로 광원을 카메라에 고정을 시키는 것이고, 후자는 가상의 세상에 광원을 고정시키는 것이다.

만약 모델링 변환에 해당하는 이동 변환과 회전 변환 다음에(Line (c)) 광원의 위치를 설정을 한다면, 광원의 위치도 이러한 모델링 변환과 뷰잉 변환을 거쳐 눈 좌표계로 변환이 된다. 이러한 방식으로 광원의 기하 속성을 설정하면 광원에 대한 애니메이션을 구현할 수가 있다. 예를 들어 광원이 주어진 경로를 따라 계속해서 움직이게 하려한다면, 그러한 움직임을 고려한 모델링 변환을 수행한 후 광원의 위치를 설정하면 된다. 또한 광원이 어떤 기하 물체에 대하여 상대적으로 고정이 된 상황도 물체에 대한 모델링 변환을 통하여 물체와 함께 같이 세상 공간에 배치를 할 수 있을 것이다. 예를 들어 비행기가 어떤 경로를 따라 날아간다고 할 때, 비행기의 헤드라이트를 스폿 광원으로 묘사한다고 생각해보자. 자신의 모델링 좌표계를 기준으로 설정된 비행기를 그리기 위해서는 적절히 그에 대한 모델링 변환을 해주어야 한다. 이 경우에는 광원의 위치가 이 비행기의 모델링 좌표계에 대하여 고정이

되어 있는 상황이므로, 비행기가 변환이 될 때 같은 모델링 변환을 사용하여 광원도 동일하게 변환을 해주면 된다. 사실 광원 자체에 대한 애니메이션이건 어떤 기하 물체에 고정된 광원이건 그 원리는 같은데, 이에 대해서는 6.4절의 예제 프로그램을 통하여 다시 한번 살펴보도록 하겠다.

### 6.2.5 조명 모델 인자

**전역 앰비언트 광원의 색깔**

OpenGL의 조명 계산에 영향을 미치는 요인들 중 마지막 부류인 조명 모델 인자에 대하여 살펴보자. 이 인자들은 표 3.4에 요약이 되어 있는데, OpenGL에서 사용되는 라이팅 모델 전반에 걸친 성질을 정의하는데 사용된다. 앞에서 언급한 바와 같이 OpenGL에서는 두 가지 형태의 앰비언트 반사를 지원한다. 광원 인자를 통하여 설정되는 앰비언트 광원의 색깔은 각 광원이 물체의 앰비언트 반사에 조금씩 다르게 영향을 미칠 수 있게 하기 위한 것으로, 이러한 것을 지역 앰비언트 반사라 하였다. 반면에 조명에 사용되는 광원들이 종합적으로 물체에 영향을 미치는 전역 앰비언트 반사를 위한 전역 앰비언트 광원(global ambient light)의 밝기 또는 색깔은 개개의 광원보다는 조명 모델 전반에 관한 것이므로 조명 모델 인자를 통하여 설정이 된다. 표 3.4의 첫 번째 인자 $\mathbf{a}_{cs}$가 바로 이를 위한 인자로서 첨자 $c$가 의미하듯이 색깔 타입의 값을 가진다. 여기서 첨자 $s$는 이 인자가 장면(scene) 전체에 영향을 미치는 조명 모델 인자라는 것을 뜻한다. 조명 모델 인자는 void glLightModel{if}(GLenum pname, TYPE param); 함수와 void glLightModle{if}v(GLenum pname, TYPE *param); 함수에 의하여 설정이 되는데, $\mathbf{a}_{cs}$ 값은 pname 인자로 상수 GL_LIGHT_MODEL_AMBIENT를 사용하여 설정할 수 있다.

| 인자 | 타입 | 디폴트 값 | 설명 |
|---|---|---|---|
| $\mathbf{a}_{cs}$ | 색깔 | (0.2, 0.2, 0.2, 1.0) | 장면의 전역 앰비언트 광원 색깔 |
| $v_{bs}$ | 부울형 | FALSE | 눈 좌표계에서의 관찰자 시점<br>TRUE : (0, 0, 0) , FALSE : (0, 0, $\infty$) |
| $c_{es}$ | 열거형 | SINGLE_COLOR | 정반사 계산 분리 여부 |
| $t_{bs}$ | 부울형 | FALSE | 양면 조명 모드 사용 여부 |

표 3.4: 조명 모델 인자

**관찰자의 시점**

다음 두 번째 인자인 $v_{bs}$에 대하여 알아보자. 이 인자의 이름 $v$는 관찰자(viewer)를 뜻하고 첨자 $b$는 이 값이 부울형(boolean type)의 값을 가짐을 의미한다. 300쪽의 식 (3.1)의 조명 모델을 다시 생각해보자. 물체 표면에서 정반사가 되는 색깔을 계산할 때에는 해프웨이 벡터 $H_i = \frac{L_i+V}{|L_i+V|}$가 영향을 미치므로, 앰비언트 반사와 난반사의 경우와는 달리 관찰자를 향한 방향에 해당하는 V 벡터에 영향을 받게 된다[15]. 전기한 바와 같이 앰비언트 반사, 난반사, 정반사 중에서 정반사가 유일하게 바라보는 방향에 영향을 받는데, 문제는 정반사 색깔을 계산하기 위해서 다면체 모델의 각 꼭지점마다 V 벡터와 그에 따른 $H_i$ 벡터를 계산하는 것이 부담이 될 수 있다는 사실이다.

이제 OpenGL 조명 계산에서 사용되는 기호를 사용하여 그림 3.17을 살펴보자. 여기서 눈 좌표계에서 V 지점의 꼭지점에 대하여 조명 계산을 하려하는데, 단위 벡터인 $\overrightarrow{\mathbf{VP}}_{pli}$는 광원을 향한 방향을, 그리고 $n$ 벡터는 V에서의 단위 법선 벡터를 나타낸다. 다음 V에서 관찰자를 향한 벡터는 직교 투영을 한다면 눈 좌표계에서의 양의 $z_e$축 방향에 해당하는 $(0\ 0\ 1\ 0)^t$이, 그리고 원근 투영을 한다면 관찰자의 위치는 눈 좌표계의 원점 $P_e = (0\ 0\ 0\ 1)^t$에 해당하므로 V에서 $P_e$를 향한 단위 벡터 $\overrightarrow{\mathbf{VP}}_e$가

[15]다시 한번 언급을 하면 앞에서 설명한 퐁의 조명 모델에서 V 벡터는 쉐이딩 계산을 하려는 지점에서 관찰자 시점을 향한 벡터이다. 반면에 현재 설명하려는 OpenGL에서 사용되는 조명 모델에서는 V는 조명 계산을 하려하는 꼭지점의 좌표를 의미한다.

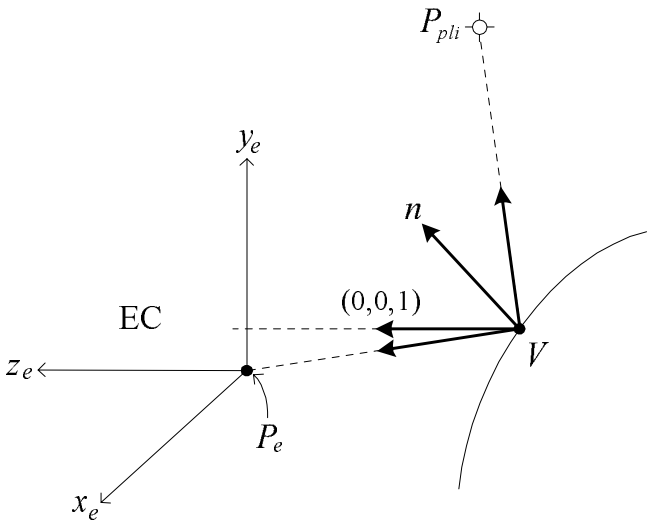



그림 3.17: 관찰자 시점의 조절

될 것이다. 전자에 해당하는 관찰자를 무한 관찰자(infinite viewer)라 하고, 후자의 경우를 지역 관찰자(local viewer)라 한다.

OpenGL에서는 라이팅 계산을 할 때, 실제로 사용하는 투영의 종류와는 상관없이 관찰자의 방향을 임의로 설정할 수 있다. 만약 관찰자를 지역 관찰자로 설정하면 해프웨이 벡터 방향은 $\overrightarrow{\mathbf{VP}}_{pli} + \overrightarrow{\mathbf{VP}}_e$가 되고, 무한 관찰자로 설정하면 $\overrightarrow{\mathbf{VP}}_{pli} + (0\ 0\ 1\ 0)^t$을 해프웨이 벡터 방향으로 사용하여 정반사 계산을 한다. 그러면 OpenGL의 조명 모델에서 관찰자 시점을 임의로 조절할 수 있게 한 이유는 무엇일까? 지역 관찰자를 사용을 하면 렌더링을 하고자하는 다면체 모델의 모든 꼭지점에 대해서 $\overrightarrow{\mathbf{VP}}_e$ 벡터를 매번 계산을 해주어야 하지만, 무한 관찰자의 경우에는 항상 이 벡터가 상수 벡터가 되어 계산량을 줄일 수가 있다. 또한 평행 광원을 사용하면 무한 관찰자의 경우에는 해프웨이 벡터가 상수 벡터가 되어 한 번만 계산을 하면 되나, 지역 관찰자의 경우에는 매 꼭지점마다 단위 해프웨이 벡터를 계산해주어야 한다. 앞에서도 언급한 바와 같이 꼭지점 한 개에 대하여 이를 계산하는 것은 시간적으로 무시할 수 있지만 이를 수십만 번 또는 그 이상으로 반복을 한다면 실시간 계산에 무

시할 수 없는 영향을 미치게 될 것이다. 따라서 직교 투영을 한다면 당연히 무한 관찰자를 사용하여야 하고, 원근 투영을 할 경우에도 가능하면 지역 관찰자 대신 무한 관찰자를 사용하는 것이 계산상 더 효율적이라고 할 수가 있다.

단 주의할 것은 관찰자의 방향을 어떻게 설정을 하건 이는 기하 변환에는 영향을 미치지는 않는다는 사실이다. 즉 어떤 관찰자 시점을 사용하건 뷰잉 관련 함수에 의하여 설정된 투영을 사용하기 때문에 물체의 모양에는 변함이 없고, 단지 정반사 계산을 하는데 필요한 인자의 값만 바꾸는 것이므로 하이라이트가 생기는 형태만 달라지게 된다. 따라서 투영 종류와 다른 형태의 관찰자 방향을 사용하면 약간은 부자연스러운 하이라이트가 생길 수도 있지만, 어차피 조명 모델은 근사화된 실험적인 모델이므로 큰 문제가 되지 않는다. 실제로 3차원 게임과 같이 순간적으로 빠르게 변하는 영상에서 이러한 문제점을 주의 깊게 관찰할 사람은 별로 없기 때문에, 주로 무한 관찰자를 사용하여 계산량을 줄이려는 시도를 하는 것이다.

OpenGL에서는 상수 GL_LIGHT_MODEL_LOCAL_VIEWER를 사용하여 관찰자 시점에 관련된 인자 $v_{bs}$ 값을 설정하는데, 이 값이 참 값(GL_TRUE)을 가지면 지역 관찰자, 그리고 거짓 값(GL_FALSE)을 가지면 무한 관찰자를 사용한다. 디폴트로는 무한 관찰자가 사용이 되는데, 이를 지역 관찰자로 바꾸려면 glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);와 같은 문장을 수행시키면 된다.

### 정반사 색깔의 분리

기존의 OpenGL 1.1 버전과는 달리 1.2 버전에서는 라이팅 계산을 할 때, 필요할 경우 정반사에 관련된 부분과 나머지 부분을 분리(separation of specular color)할 수 있는 기능을 제공한다. 이에 관련된 인자가 표 3.4의 세 번째에 있는 $c_{es}$이다. 여기서

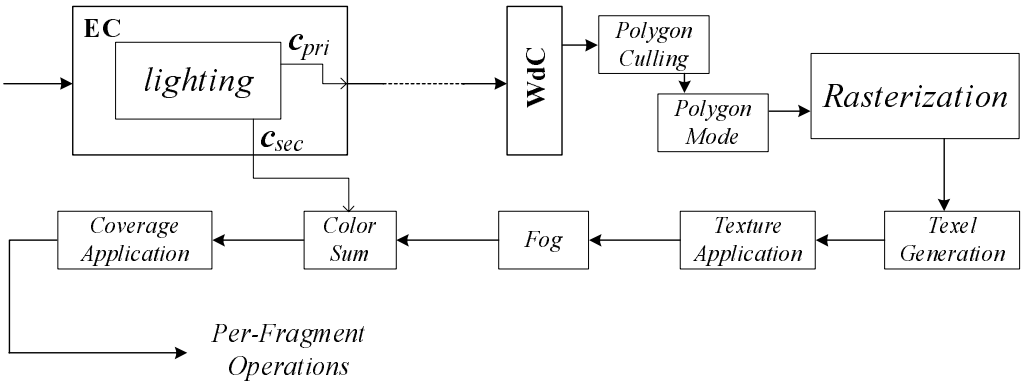



그림 3.18: 정반사 색깔의 분리

$c$는 색깔의 조절(color control)을 나타내고, 첨자 $e$는 이 변수가 열거형(enumeration type)의 값을 가짐을 의미한다. OpenGL에서 이 인자를 지칭하는 상수는 GL_LIGHT_MODEL_COLOR_CONTROL으로서, GL_SINGLE_COLOR와 GL_SEPARATE_ SPECULAR_COLOR 중의 한 개의 값을 가진다. 앞에서 잠깐 언급한 바와 같이 OpenGL에서 사용되는 라이팅 모델은 두 개의 색깔을 계산한다. 이를 각각 주 색깔과 보조 색깔이라고 했는데, 이들을 각각 $\mathbf{c}_{pri}$와 $\mathbf{c}_{sec}$으로 표시하자. $c_{es}$의 디폴트 값은 GL_SINGLE_COLOR인데, 이러한 경우에는 $\mathbf{c}_{pri}$는 다음에 설명할 조명 공식을 사용하여 앰비언트 반사, 난반사, 정반사, 그리고 빛의 방사까지 모두 더한 색깔이 되고, $\mathbf{c}_{sec}$은 (0, 0, 0, 0) 값을 가지게 된다. 만약 다음과 같은 OpenGL 문장을 통하여 정반사 색깔을 분리시키면,

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

$\mathbf{c}_{pri}$는 모든 반사를 더한 색깔에서 정반사 부분만 뺀 색깔이 되고, $\mathbf{c}_{sec}$은 각 광원에 대하여 정반사 부분만 모두 더한 값을 가지게 된다.

정반사 색깔을 분리하면 정반사 효과를 좀 더 자연스럽게 낼 수가 있는데, 텍스춰 매핑에 대한 예를 통하여 이러한 기능의 필요성에 대하여 살펴보자. 정반사 부

분을 제외한 나머지 색깔, 특히 난반사 색깔은 물체의 기반이 되는 색깔인데, 물체의 표면에 텍스춰를 입히면 일반적으로 물체의 기반 색깔이 바뀌게 된다. OpenGL 렌더링 파이프라인에서 텍스춰 매핑에 대한 색깔 계산은 조명 계산과 래스터화 계산이 모두 끝난 후 픽셀 단위로 수행이 된다(그림 3.18). 따라서 기존의 방법에서는 라이팅 계산을 통하여 정반사 색깔을 비롯한 모든 반사 색깔을 다 더한 후, 텍스춰 매핑 계산 과정에서 물체의 기반이 되는 색깔을 수정하려하기 때문에 종종 부자연스러운 결과를 낳고는 하였다. 좀 더 자연스러운 방법은 정반사 부분과 나머지 부분을 분리하여 물체의 기반이 되는 색깔인 $\mathbf{c}_{pri}$에 대하여 텍스춰를 입힌 다음, 정반사에 관련된 정보를 포함하고 있는 $\mathbf{c}_{sec}$을 더함으로써 좀 더 자연스러운 하이라이트 효과를 내는 것이다. 그림 3.18을 보면 알 수 있듯이 눈 좌표계(EC)에서 정반사 색깔이 분리가 될 경우, 주 색깔인 $\mathbf{c}_{pri}$는 다른 정보들과 함께 렌더링 파이프라인을 따라 흘러가 이 색깔에 대하여 텍스춰 매핑이 되고, 그 결과가 색깔 합(Color Sum) 모듈에서 기다리고 있는 보조 색깔 $\mathbf{c}_{sec}$과 더해지게 된다. 물론 기존의 방법을 그대로 사용하고 싶으면 정반사 색깔을 꼭 분리할 필요는 없는데, 이러한 정반사 색깔, 즉 하이라이트의 분리는 자연스러운 텍스춰 매핑 외에도 고급 렌더링 기법을 구현하는데 유용하게 사용될 수 있다.

**양면 조명**

6.2.1절에서 설명한 바와 같이 다면체 모델을 기술할 때 각 다각형의 꼭지점을 일관된 순서로 나열해야 한다고 하였다. 다각형에 대한 방향성은 glFrontFace(*) 함수로 설정할 수 있다고 하였는데, 일단 방향성을 설정하면 관찰자가 어떤 다각형을 바라볼 때 앞면을 바라보고 있는지 뒷면을 바라보고 있는지를 결정할 수 있게 된다. 만약 어떤 물체가 불투명하고 닫힌 물체라면, 이를 바깥쪽에서 바라보면 항상 앞면에

해당하는 다각형만 보이게 된다. 이럴 경우 굳이 뒷면이 보이는 다각형에 대한 계산은 할 필요가 없기 때문에, 그러한 다각형은 렌더링 계산 중에 단순히 제거를 하면 된다.

이러한 계산 과정을 뒷면 제거(backface culling)라고 한다고 했는데, OpenGL에서 뒷면 제거의 수행 여부는 void glCullFace(GLenum mode); 함수에 의하여 설정할 수 있다. 이 함수의 인자 mode는 제거를 하려하는 다각형의 종류에 따라 GL_FRONT, GL_BACK, GL_FRONT_AND_BACK 중의 값을 가진다. 꼭 뒷면만 제거를 할 수 있는 것이 아니라 어떤 면도 제거할 수 있지만, 일반적으로는 GL_BACK을 사용하여 뒷면을 제거한다. 이 때 이 함수는 단지 어떤 면을 제거할 것인가를 설정하는 것이고, glEnable(GL_CULL_FACE); 문장을 통하여 렌더링 파이프라인에서 뒷면 제거 계산이 일어나도록 명시적으로 설정을 해야 한다. 물론 더 이상 제거 기능이 필요하지 않으면 glDisable(GL_CULL_FACE);와 같은 문장을 수행하면 된다.

기하 물체를 밖에서 바라보았을 때 꼭지점들이 반시계 방향으로 나열이 되도록 모델링이 되었다면, 뒷면에 해당하는 다각형을 제거하고 싶으면 다음과 같이 함수를 호출하면 된다.

```
glFrontFace(GL_CCW);
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
```

만약 렌더링 하려고 하는 기하 물체가 닫힌 물체가 아니라면 어떻게 될까? 예를 들어 물주전자를 절단 평면을 사용하여 반쯤 잘랐다면 물주전자의 안쪽에 해당하는 다각형, 즉 뒷면이 보이는 다각형들을 볼 수가 있게 된다. 제대로 모델링을 하였다면 물체의 꼭지점에 법선 벡터를 붙일 때 항상 물체의 바깥쪽을 향하도록 하였을 것이다. 따라서 앞면이 보이는 다각형의 경우에는 그 꼭지점의 법선 벡터들이 관찰자 방향으로 향할 것이고, 뒷면이 보인다면 법선 벡터들은 그 반대쪽을 가리키고 있

을 것이다.

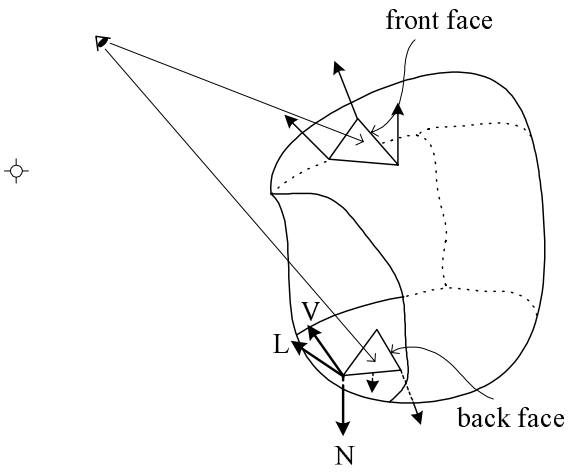



그림 3.19: 앞면과 뒷면

문제는 한 다각형에 대하여 뒷면이 보일 때 어떠한 방식으로 라이팅 계산을 할 것인가 하는 것이다. 편의상 5절에서 퐁의 조명 모델을 설명할 때 사용했던 기호를 다시 사용하여 그림 3.19를 살펴보자. 무엇보다도 난반사와 정반사 색깔을 계산하려면 해당하는 두 단위 벡터간의 내적 $L \cdot N$과 $N \cdot H$를 계산해야 한다. 이는 두 벡터간의 각도에 대한 코사인 값으로서, 두 벡터간의 각도는 0도에서 180도까지 벌어질 수가 있다. 만약 이 그림에서처럼 뒷면에 대한 조명 계산을 한다면 이 각도들은 모두 90도보다 커서 코사인 값이 음수가 나오기 때문에, 이에 대하여 조심을 하지 않으면 문제가 발생한다. OpenGL에서도 그러하듯이 일반적으로 조명 계산을 할 때 두 단위 벡터 $\mathbf{d}_1$과 $\mathbf{d}_2$의 내적은 다음과 같은 방식으로 계산을 한다.

$$\mathbf{d}_1 \odot \mathbf{d}_2 = max\{\mathbf{d}_1 \cdot \mathbf{d}_2, 0.0\}$$

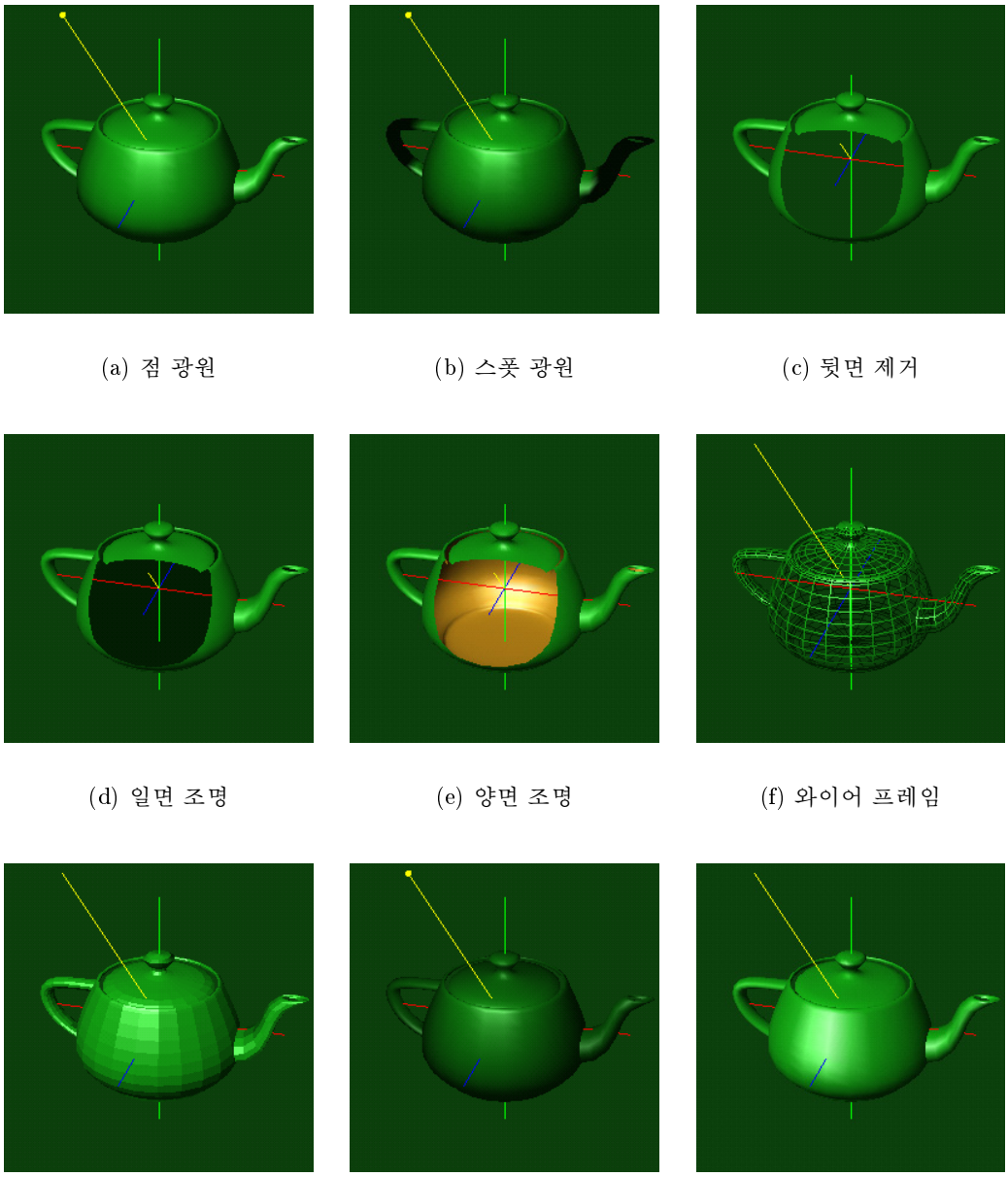
즉 두 벡터간에 내적을 취했을 때, 양수이면 그 값을 취하고 음수이면 0 값을 취하는데, 이는 벡터간의 각도가 90도 이하일 경우에만 해당 광원을 고려하겠다는 것이다. 다시 말하면 물체 표면의 뒤쪽에서 들어오는 빛은 무시를 하겠다는 것을 의미한다. 따라서 이와 같은 방식으로 뒷면에 대한 조명 계산을 하면 물주전자의 안쪽

면은 앰비언트 광원에 대한 반사 색깔만 더해지므로, 일반적으로 어둡게 보일 것이다.

경우에 따라서는 안쪽 면도 밝게 보이게 하고 싶을 때가 있을 것이다. 오히려 이것이 더 자연스러운 상황이라 할 수 있는데, OpenGL에서도 이러한 상황을 처리하기 위하여 앞, 뒤 양면에 대하여 조명 계산을 할 수 있도록 해준다. 이를 결정하는 인자는 표 3.4의 마지막에 있는 $t_{bs}$로서 이에 대응되는 OpenGL 상수는 GL_LIGHT_MODEL_TWO_SIDE이다. 이 인자의 이름에서 $t$는 양면(two-sided)을 의미하고, 첨자 $b$는 이 인자가 부울형의 값을 가짐을 의미한다. 이 인자가 참 값으로 설정될 때만 양면 조명(two-sided lighting)을 수행하는데, 디폴트로는 거짓으로 설정이 되어 있어, 일면 조명(one-sided lighting)만 하게 된다.

그림 3.20을 보면 OpenGL에서의 뒷면 제거와 양면 조명의 효과를 쉽게 이해할 수가 있다. 우선 뒷면 제거를 한다면 주전자의 안쪽에 해당하는 다각형은 모두 제거가 되어 렌더링 계산에 영향을 안 미치게 된다. 따라서 그림 3.20(c)에서와 같이 그 부분에 구멍이 난 것처럼 배경 색깔이 나타난다. 다음 그림 3.20(d)와 (e)는 뒷면 제거를 하지 않는 상황에서 각각 일면 조명과 양면 조명을 한 모습을 보여준다. 일면 조명을 할 경우에는 보이는 다각형이 앞면이건 뒷면이건 항상 앞면의 물질 성질로 설정된[16] 물질의 성질과 원래의 법선 벡터를 사용하여 조명 계산을 한다. 따라서 안쪽 면, 즉 뒷면이 보이는 다각형의 꼭지점에 대하여 조명 계산을 할 때에는 난반사와 정반사에 대한 색깔을 계산하는 부분에서 벡터의 내적 값이 0이 되고, 결국 앰비언트 반사 값만 가지게 된다. 그림에서 보면 알 수 있듯이 뒷면이 어둡게 앰비언트 색깔로 칠해져 있음을 알 수 있다.

[16] 즉 glMaterial*(GL_FRONT, ···, ···); 또는 glMaterial*(GL_FRONT_AND_BACK, ···, ···);와 같은 함수 호출을 통하여 설정된,



(a) 점 광원 (b) 스폿 광원 (c) 뒷면 제거

(d) 일면 조명 (e) 양면 조명 (f) 와이어 프레임

(g) 플랫 쉐이딩 (h) 빛의 감쇠 효과 (i) 무한 관찰자

그림 3.20: 다양한 조명 인자의 설정 예

양면 조명을 하는 경우에는 우선 보이는 다각형이 앞면인지 뒷면인지를 판단해서, 1. 앞면이라면 앞면의 물질 성질과 원래의 법선 벡터를 사용하여 조명 계산을 하고, 2. 뒷면이라면 뒷면에 대한 물질 성질[17]과 원래의 법선 벡터의 방향을 반대로 한 법선 벡터를 사용하여 조명 계산을 한다. 물론 일면 조명인지 양면 조명인지 하는 것은 세 가지 기하 프리미티브들 중 다각형에만 영향을 미치고 점과 선분에는 영향을 미치지 않는다. 그림 3.20(e)에서는 양면 조명의 효과를 뚜렷하게 하기 위하여 다음과 같이 뒷면이 황동처럼 보이게 설정을 하였다.

```
GLfloat mat_ambient2[4] = {0.33, 0.22, 0.03, 1.0};
GLfloat mat_diffuse2[4] = {0.78, 0.57, 0.11, 1.0};
GLfloat mat_specular2[4] = {0.99, 0.91, 0.81, 1.0};
GLfloat mat_shininess2 = 27.8;
      ⋮
glMaterialfv(GL_BACK, GL_AMBIENT, mat_ambient2);
glMaterialfv(GL_BACK, GL_DIFFUSE, mat_diffuse2);
glMaterialfv(GL_BACK, GL_SPECULAR, mat_specular2);
glMaterialf(GL_BACK, GL_SHININESS, mat_shininess2);
```

뒷면 제거나 양면 조명을 구현할 때 가장 중요한 요소 중의 하나는 다각형의 앞면이 보이는지, 또는 뒷면이 보이는지를 어떻게 결정할 것인가 하는 것이다. 한 가지 가능한 방법은 다각형을 포함하는 평면에 대하여 바깥쪽을 향하는 벡터를 구하고, 다음 다각형의 중점에서 시점을 향하는 벡터를 계산한 후 그 두 벡터의 내적을 취하는 것이다. 만약 그 값이 양수이면 앞면, 그리고 음수이면 뒷면이 됨을 쉽게 알 수가 있는데, 이러한 방식은 세상 좌표계나 눈 좌표계에서 개념적으로는 쉽게 구현이 가능하나, 그 계산량이 적지 않아 렌더링 파이프라인에 부담이 된다.

OpenGL 시스템에서는 그러한 판단을 하기 위하여 일단 다각형의 꼭지점 좌표들을 윈도우 좌표계까지 변환을 한 후, 각 다각형에 대한 래스터화 과정의 직전에

[17] 즉 glMaterial*(GL_BACK, ···, ···); 또는 glMaterial*(GL_FRONT_AND_BACK, ···, ···);와 같은 함수 호출을 통하여 설정된,

어떤 면이 보이는지를 결정한다(그림 3.18의 다각형 제거(Polygon Culling) 모듈). $n$개의 꼭지점을 가지는 다각형을 윈도우 좌표계까지 변환을 하였을 때의 꼭지점 좌표를 $(x^i_{wd}, y^i_{wd})$, $i = 0, 1, \cdots, n-1$이라 할 때, $i \oplus 1 \equiv (i+1)\ mod\ n$과 같이 $n$에 대한 모수 함수를 정의하면 다음 공식은 특별한 의미를 가지게 된다.

$$a = \frac{1}{2}\sum_{i=0}^{n-1}(x^i_{wd}y^{i\oplus 1}_{wd} - x^{i\oplus 1}_{wd}y^i_{wd})$$

여기서 $a$의 절대값은 다각형이 볼록한지 오목한지에 상관없이 윈도우 좌표계에 투영되었을 때의 면적을 나타내는데, 한 가지 재미있는 것은 이 값의 부호가 방향성을 가진다는 사실이다. 이해를 쉽게 하기 위하여 다각형이 삼각형인 경우에 대하여 생각해보자.

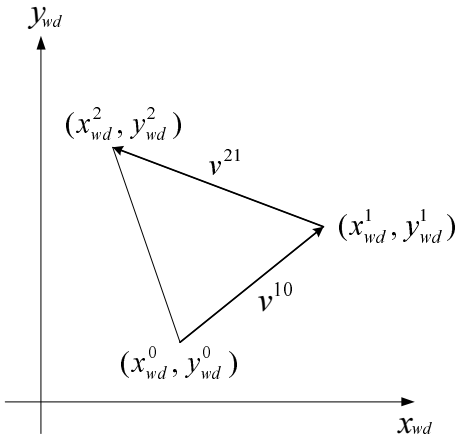



그림 3.21: 방향성을 가지는 면적

이 그림 3.21에서와 같이 삼각형의 꼭지점을 연결하는 두 벡터 $v^{10}$과 $v^{21}$을 생각해보자. 이 때 이 두 벡터를 $z$ 좌표가 0인 3차원 벡터로 생각을 하고, 그에 대하여 외적 $v^{10} \times v^{21}$을 취해보자. $n = 2$일 때 위의 식은 $a = (x^0_{wd}y^1_{wd} - x^1_{wd}y^0_{wd}) + (x^1_{wd}y^2_{wd} - x^2_{wd}y^1_{wd}) + (x^2_{wd}y^0_{wd} - x^0_{wd}y^2_{wd}) = (x^1_{wd} - x^0_{wd})(y^2_{wd} - y^1_{wd}) - (y^1_{wd} - y^0_{wd})(x^2_{wd} - x^1_{wd})$과 같이 되는데, 이 값은 외적 계산의 결과 생성된 벡터의 $z$ 좌표가 됨을 알 수 있다. 즉 이 벡터의 $x$와 $y$ 좌표는 0이 되기 때문에, 이 값의 절대값은 외적의 정의상 이 삼각형 면적의 두 배가 된다. 또한 $a$의 부호는 방향을 나타내는데, 이 또한 외적의 정의상 그 값이 양수이면 꼭지점이 반시계 방향으로 회전하고, 음수이면 시계 방향으

로 회전을 하게 됨을 알 수 있다. 물론 이 값이 0이라면 이 삼각형은 화면에 수직으로 보이는 특별한 경우에 해당한다.

참고로 양면 조명을 할 때 기하 프리미티브들 중 점과 선분에 대해서는 항상 앞면에 대한 물질 성질을 이용하여 생성한 앞면 색깔이 사용이 된다. 반면에 다각형에 대해서는 윈도우 좌표계에서 다각형의 방향성을 판단한 후 앞면과 뒷면에 대한 색깔 중 해당하는 색깔을 선택하게 된다. 예를 들어 glFrontFace(GL_CCV);와 같이 방향성을 선택을 하였을 때, $a$ 값이 음수라면 각 꼭지점에 대하여 뒷면에 대한 색깔이 선택이 될 것이다. 한 가지 문제는 이러한 판단이 윈도우 좌표계에서 이루어지기 때문에, 양면 조명을 할 경우 라이팅 계산이 수행되는 눈 좌표계에서는 처리하려는 다각형에 대하여 어떤 면이 보이는지를 모르기 때문에, 각 꼭지점에 앞면과 뒷면의 색깔을 모두 계산을 해주어야 한다는 사실이다. 따라서 다각형에 대한 각 꼭지점에 대하여 두 가지 종류의 색깔이 계산이 되어 렌더링 파이프라인을 따라 가다가, 윈도우 좌표계에서 그 중 하나가 선택이 되는 것이다. 이에 대하여 좀 더 효율적으로 구현하는 방법을 생각할 수 있겠으나, 여하한 경우 양면 조명은 일면 조명보다 더 많은 계산을 요구하므로 꼭 필요한 경우가 아니라면 이 기능은 사용하지 않는 것이 효율적인 프로그램 작성에 도움이 될 것이다.

## 6.3 라이팅 관련 OpenGL 프로그래밍

이제 지금까지 살펴본 내용을 종합하여 조명을 사용하는 간단한 OpenGL 프로그램에 대하여 살펴보자. 라이팅 계산을 하기 위해서는 2장에서와 같이 단순하게 물체의 골격, 즉 와이어 프레임만 그리는 경우와는 달리 몇 가지 사항을 더 고려해야 한다. 이 절에서는 조명 계산을 위하여 추가적으로 다루어야 하는 중요한 사항들을 그림 3.20(a)와 같은 이미지를 생성해주는 프로그램 예 3.2를 통하여 하나씩 살펴보도록 하자(예제 프로그램 3.A 참조).

1. 기하 물체의 각 꼭지점을 기술할 때 그 점에서의 법선 벡터도 함께 정의한다.
   각 꼭지점에 대한 법선 벡터 방향, 즉 그 지점에서 물체 표면에 수직인 방향은 조명 계산에 있어 매우 중요한 역할을 한다고 했다. 따라서 렌더링을 하는데 있어 조명을 사용하기 위해서는 꼭지점의 좌표와 함께 법선 벡터도 정의해주어야 한다. 이 예제 프로그램에서는 그리려고 하는 물주전자에 대한 다면체 모델이 편의상 삼각형과 사각형으로 구성되어 있다고 가정하고 있는데, 이러한 기하 물체(teapot)를 스트럭춰 타입인 `Polygon` 타입으로 정의하였다(Line (a)). `draw_teapot()` 함수에서 물체의 기하 정보를 기술할 때, glNormal3fv(*) 함수를 사용하여 법선 벡터를 기술하고 있다(Line (b)). 물론 glNormal3fv(*) 함수는 현재 값을 설정하는 함수이므로, 대응되는 glVertex3fv(*) 함수 이전에 호출되어야 한다. 여기서는 편의상 미리 법선 벡터의 길이를 1로 만들어 사용하였는데, 길이가 1이 아닌 법선 벡터의 처리에 대해서는 2장의 3.16.2절의 내용을 다시 참조하기 바란다.

2. 기하 물체의 표면에서의 빛의 반사 형태를 결정하는 물질 성질을 정의한다.
   물질의 빛의 반사 형태는 앞에서도 설명한 바와 같이 glMaterial*(*) 함수에

의해 설정이 되는데, 307쪽에서 예를 든 바와 같이 원하는 물질의 물질 인자를 설정하면 된다(Line (g)).

3. 사용하려는 조명 모델의 일반적인 성질을 설정한다.

   이를 위해서 glLightModel*(*) 함수를 사용하면 되는데, 여기서는 전역 앰비언트 광원의 색깔 `global_ambient[4]`를 (0.15, 0.15, 0.15, 1.0)으로 하였고(Line (h)), 그 다음 문장에서 지역 관찰자를 사용하도록 설정하였다.

4. 사용하려는 광원의 성질을 정의한다.

   각 광원의 성질은 glLight*(*) 함수를 사용하여 설정한다고 하였다. 본 예제 프로그램에서는 위치 `light_position[4]`가 $(0.0\ 5.0\ 5.0\ 1.0)^t$인 점 광원을 사용하였는데(Line (n)), 광원의 색깔도 313쪽의 예에서와 같이 원하는 색깔로 설정하면 된다. 광원의 성질 중 위치, 또는 방향을 설정하는 함수의 호출 위치는 중요한 의미를 가진다고 하였다. 이 예에서는 `gluLookAt(*);` 문장 직후에 위치를 설정하고 있기 때문에, 모델뷰 형렬 스택의 탑에 뷰잉 변환만 올라간 상태이고, 따라서 $(0.0\ 5.0\ 5.0\ 1.0)^t$라는 좌표는 세상 좌표계에서의 의미를 가진다.

5. 명시적으로 조명 계산을 하도록 선언하고, 적절하게 광원을 점등한다.

   OpenGL에서는 눈 좌표계에서 꼭지점에 색깔을 붙일 때 두 가지 방법이 가능하고, 디폴트 상태는 현재 색깔로 지정된 색깔을 사용하는 것이라고 하였다. 따라서 라이팅 계산을 수행한 결과의 색깔을 사용하려면 명시적으로 glEnable(GL_LIGHTING); 문장을 수행시켜주어야 한다(Line (i)). 그리고 한 개 또는 그 이상의 광원을 glLight*(*) 함수를 통하여 설정하였을 때, 실제로 사용하려는 모든 광원을 glEnable(GL_LIGHTi);와 같은 문장을 통하여 명시적

으로 점등해주어야 한다(Line (i) 다음 문장). 물론 사용중인 광원을 끄려면 glDisable(GL_LIGHTi)와 같은 문장을 사용하면 된다. 이 예제 프로그램의 수행 결과 렌더링 된 이미지를 보면 조명 계산을 통하여 얻어진 물주전자와 함께 세상 좌표계에의 세 축(각각 빨간색, 초록색, 파란색 직선)과 점 광원에 대한 정보(노란색 직선)를 볼 수가 있다. 이들 직선은 `draw_axes_and_light_info()` 함수에서 라이팅 계산을 통한 색깔이 아니라 현재 색깔을 사용하여 꼭지점에 색깔을 연관 시켰는데, OpenGL에서는 Line (c)와 Line (d)에서와 같이 조명 계산의 여부를 동적으로 조절할 수가 있다.

6. 깊이 버퍼 기능을 사용하도록 한다.

물체의 와이어 프레임만 그리는 프로그램과는 달리 조명 계산을 추가하기 위해서는 기하 물체에 껍질을 입히는 작업을 수행해주어야 하는데, 이 때 이미지를 생성하였을 때 제대로 앞면에 해당하는 면이 보이도록 해주어야 한다. 기하 물체를 그릴 때 대부분의 경우 그 물체를 구성하는 다각형을 나열하는 순서는 고정이 되어 있다. 따라서 임의의 지점에서 시점이 설정된다면, 다각형들은 무작위 순서로 렌더링 파이프라인에 들어온다고 할 수 있다. 만약 단순히 들어오는 순서대로 계속해서 프레임 버퍼에 그린다면, 렌더링을 마친 후에는 앞쪽의 다각형에 가려 보이지 않는 뒤쪽의 다각형이 그려질 수도 있을 것이다. 따라서 항상 앞쪽에 보이는 다각형만 그려지도록 렌더링 계산을 해주어야 하는데, 이러한 과정, 즉 앞쪽에 보이는 다각형은 그리고 이들에 가려 안 보이는 뒤쪽의 다각형은 그리지 않는 과정을 은면 제거라 한다고 하였다.

여러 은면 제거 알고리즘 중에서 OpenGL에서는 깊이 버퍼 기법(depth buffering), 또는 Z 버퍼 기법(Z buffering)이라는 방법을 사용하여 은면을 제거하는

데, 이에 대한 자세한 내용은 6장에서 다루도록 하겠다. 다행히도 OpenGL 시스템이 이러한 계산을 하도록 설정하는 것은 간단하다. 우선 디스플레이 모드를 설정할 때 은면 제거 계산을 하는데 필요한 정보를 저장해주는 깊이 버퍼(depth buffer)를 요청한다. 이는 glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH); 문장에서처럼 GLUT_DEPTH 상수를 추가하면 된다. 다음 Line (f)에서와 같이 명시적으로 OpenGL 시스템이 은면 제거 계산을 하도록 설정한다. 그리고 매번 그림을 그릴 때마다 glClear(*) 함수에 GL_DEPTH_BUFFER_BIT를 추가하여 색깔 버퍼를 초기화 할 때 깊이 버퍼도 같이 초기화하도록 한다(Line (o)). 이 함수가 수행될 때에는 잘 알다시피 색깔 버퍼는 glClearColor(*) 함수로 설정된 색깔(Line (e))로 초기화가 된다. 깊이 버퍼를 초기화할 때 사용하는 값은 glClearDepth(*) 함수를 사용하여 명시적으로 설정할 수 있으나, 여기서는 디폴트 값을 사용하여 전형적인 은면 제거 계산을 하도록 했다.

7. 조명 계산에 영향을 미치는 다각형에 대한 성질을 설정한다.

   다음 세 가지 사항은 눈 좌표계에서 수행되는 라이팅 계산에 직접적으로 영향을 미치는 인자들을 설정하는 것은 아니고, 전체 렌더링 파이프라인에서 물체의 외관을 결정하는데 영향을 미치는 다각형의 속성들의 설정에 대한 것이다.

   - 다각형의 방향성을 설정한다.

     앞에서도 설명한 바와 같이 다면체 모델을 구성하는 다각형은 일관된 방향성을 가져야 하는데, 그러한 방향성을 glFrontFace(*) 함수를 통하여 설정해주어야 한다. 여기서는 반시계 방향으로 설정을 하였는데(Line (j)),

이 방향이 디폴트 방향이므로 이러한 경우에는 굳이 이 함수를 수행시킬 필요는 없으나 프로그램을 좀 더 명확하게 하려면 명시적으로 설정하는 것이 좋은 습관이라 하겠다.

- 뒷면 제거를 할 지를 결정한다.

  Line (k)에서 렌더링 계산 중에 꼭지점 정보들이 윈도우 좌표계에 다다랐을 때 뒷면으로 판정이 나는 다각형은 제거하라고 설정하고 있다. 명심해야 할 것은 Line (k)의 다음 문장에서처럼 명시적으로 뒷면 제거 계산이 수행되도록 해주어야 한다는 사실이다.

- 보이는 다각형을 어떻게 그릴 것인가를 결정한다.

  마지막으로 그리려고 하는 다각형이 어떻게 래스터화가 될 지에 대한 성질을 설정하여야 한다. 이는 조명 계산에 직접적으로 영향을 미치는 것은 아니고, 래스터화 과정에 영향을 미치게 되는데, 다각형의 외관을 결정한다. 하나는 다각형 모드(polygon mode)에 관한 것이고, 다른 하나는 아래에서 설명할 쉐이딩 모델(shading model)에 대한 것이다. 다각형 모드는 void glPolygonMode(GLenum face, GLenum mode); 함수에 의하여 설정이 된다. face는 다른 관련 함수들과 같이 GL_FRONT, GL_BACK, GL_FRONT_AND_BACK 중의 값을 가지고, mode는 GL_POINT, GL_LINE, GL_FILL 중에서 선택을 할 수 있다. 이 함수를 통하여 다각형을 어떻게 그릴 것인가를 설정할 수 있다. 즉 위의 세 상수는 다각형을 그릴 때 각각 꼭지점만 가지고 그릴지, 선분만 가지고 그릴지, 아니면 다각형 내부까지 채워 그릴지를 설정한다. 보통은 이 예제 프로그램에서와 같이 내부를 채워 그리나(Line (l)), 상황에 따라 점이나 선분만 사용하여 그리는 방법이 유용하게 쓰일 수도 있다. 만약 Line (l)의 함수를

`glPolygonMode(GL_FRONT, GL_LINE);`으로 대치한다면 그림 3.20(f)와 같이 다각형의 골격만 그려지게 될 것이다.

8. 사용하려는 쉐이딩 모델을 설정한다.

마지막으로 쉐이딩 모델을 결정하여야 한다. 앞에서 설명한 바와 같이 눈 좌표계에서 각 기하 프리미티브를 구성하는 꼭지점들에 대하여 프로그래머가 원하는 방식으로 적절한 색깔이 붙어진 후, 계속하여 렌더링 파이프라인으로 흘러가게 된다. 윈도우 좌표계에 이르렀을 때 래스터화 과정이 수행이 되는데, 이는 화면에서 각 프리미티브들이 차지하는 화소들을 모두 찾아 적절한 색깔로 칠해주는 과정이다. 문제는 과연 무엇이 적절한 색깔인가 하는 것인데, 물론 꼭지점에 관련지어진 색깔을 사용하는 것이 당연하다고 할 수 있다. 점의 경우에는 한 가지 색깔이 연관지어지므로 그 색깔을 사용하면 되나, 선분이나 다각형의 경우에는 두 개 이상의 꼭지점으로 구성이 되므로, 그들에 붙여진 색깔들을 사용하여 화소에 칠할 색깔을 결정하여야 한다.

OpenGL의 래스터화 과정에서는 색깔을 선택하는 방식으로 두 가지 방법을 사용한다. 하나는 플랫 쉐이딩(flat shading)이라는 방법이고, 다른 하나는 스므드 쉐이딩(smooth shading)이라고 하는 방법이다. 전자는 선분이나 다각형에 해당하는 모든 화소들을 하나의 색깔로 칠하는 것이다. 선분의 경우에는 두 번째 꼭지점에 붙여진 색깔을 사용하고, 다각형의 경우에는 프리미티브 타입에 따라 몇 번째 꼭지점의 색깔을 선택할지가 달라진다. 이와는 달리 OpenGL의 디폴트 모드인 스므드 쉐이딩을 선택하면, 각 꼭지점에 연관된 색을 부드럽게 혼합하여 선분이나 다각형의 내부를 칠하게 된다. 그림 3.20(g)는 플랫 쉐이딩을 하여 렌더링을 한 모습인데, 물주전자를 구성하는 다각형마다 각

각 한 가지의 색깔로 칠해졌음을 알 수 있다. 이 때 라이팅 계산을 하도록 했으므로 전체적으로 그러한 조명 효과가 나고 있다. 반면에 그림 3.20(a)는 이 예제 프로그램에서와 같이 스므드 쉐이딩을 사용을 한 경우로서(Line (m)), 조금씩 변하는 각 꼭지점의 색깔을 선형 보간법을 사용하여 혼합을 하기 때문에 자연스러운 효과가 나게 된다. 어떤 쉐이딩 모델을 사용할 지는 void glShadeModel(GLenum mode); 함수를 사용하여 설정할 수 있다. 인자 mode는 GL_SMOOTH와 GL_FLAT 중 하나의 값을 선택하게 되는데 그 의미는 분명하다.

**프로그램 예 3.2** 조명 계산 관련 코드의 추가 예.

```
typedef struct {
 int nvertex;
 GLfloat vertex[4][3]; GLfloat normal[4][3];
} Polygon;

Polygon *teapot; int npoly_teapot; // Line (a)

void draw_teapot(void) {
 int i, j;
 Polygon *ptr;

 glPushMatrix();
    ⋮       // Modeling transform
  i = 0; ptr = teapot;
  while(i++ < npoly_teapot) {
   glBegin(GL_POLYGON);
    for (j = 0; j < ptr->nvertex; j++) {
      glNormal3fv(ptr->normal[j]); // Line (b)
      glVertex3fv(ptr->vertex[j]);
    }
   glEnd();
   ptr++;
```

```
    }
  glPopMatrix();
}


void draw_axes_and_light_info(void) {
  glDisable(GL_LIGHTING); // Line (c)
  glBegin(GL_LINES);
    glColor3f(1.0, 0.0, 0.0); // draw the x-axis
    glVertex3f(-3.0, 0.0, 0.0);
    glVertex3f(3.0, 0.0, 0.0);
      ⋮       // draw the other lines and light source
  glEnable(GL_LIGHTING); // Line (d)
}


void init_OpenGL(void) {
  glClearColor(0.05, 0.25, 0.05, 1.0); // Line (e)
  glEnable(GL_DEPTH_TEST); // Line (f)
  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient); //
Line (g)
      ⋮
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT,
                          global_ambient); // Line (h)
  glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER, 1.0);
  glEnable(GL_LIGHTING); // Line (i)
  glEnable(GL_LIGHT0);
  glFrontFace(GL_CCW); // Line (j)
  glCullFace(GL_BACK); // Line (k)
  glEnable(GL_CULL_FACE);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); // Line (l)
  glShadeModel(GL_SMOOTH); // Line (m))
      ⋮       // Projection transform
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(4.0, 8.0, 14.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position); //
Line (n)
      ⋮
}
```

```
void display (void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //
Line (o)
  draw_axes_and_light_info();
  draw_teapot();
  glFlush();
}
```

## 6.4 광원과 카메라의 움직임에 관한 프로그래밍 예

이 절에서는 카메라와 광원이 움직일 때, 이들을 어떻게 하면 적절하게 처리를 해 줄 수 있는지를 프로그램 예 3.3을 통하여 살펴보자(예제 프로그램 3.B참조). 이 프로그램을 수행시키면 처음에는 물주전자가 세상 좌표계의 $y_w$축(초록색 축)을 따라 상하로 움직이고 있고, 카메라는 $y_w$축 둘레로 원점을 바라보며 회전을 하고 있다. 카메라에는 스폿 광원인 헤드라이트(GL_LIGHT1)가 고정이 되어 물주전자를 비추고 있다. 한편 조그마한 노란색 정육면체로 도시된 또 다른 스폿 광원(GL_LIGHT0)이 세상 공간에서 카메라의 여덟 배 빠른 속도로 회전을 하고 있다. 오른쪽 마우스를 클릭하면 메뉴가 나오는데 이를 사용하여 회전하고 있는 카메라나 스폿 광원을 선택적으로 정지하거나 또는 다시 움직이게 할 수가 있다.

이 예에서는 세 가지 종류의 움직임을 처리해주어야 한다. 우선 물 주전자가 상하로 움직이는 것과 같은 것은 물체에 대한 모델링 변환을 통하여 쉽게 처리를 해 줄 수가 있다. 다음 카메라에 대한 움직임을 생각해보면, 카메라가 움직일 경우 기하 물체에 대한 변환과 같이 매 프레임마다 카메라의 기하 속성, 즉 위치와 방향 등을 다시 구해 뷰잉 변환을 새롭게 설정해주어야 한다. 여기서는 카메라가 $y_w = 8$인

평면 상에서 반지름이 14.5인 원을 따라 회전을 하고 있다. 앞에서와 같이 아이들 함수 `next_frame()`이 수행이 될 때, 카메라에 대한 회전 각도 `rotate_cam_angle`이 360을 주기로 1도씩 증가하는데(Line (c)), 프로그램 수행 중의 불필요한 계산을 줄이기 위하여 초기에 `set_rotate_cam_tab()` 함수를 수행시켜 각 각도에 대한 카메라의 위치를 배열 `rotate_cam_tab[360][3]`에 저장을 해놓고, 프로그램의 실행 시에는 그 값을 사용하여 뷰잉 변환을 해주고 있다(Line (i)).

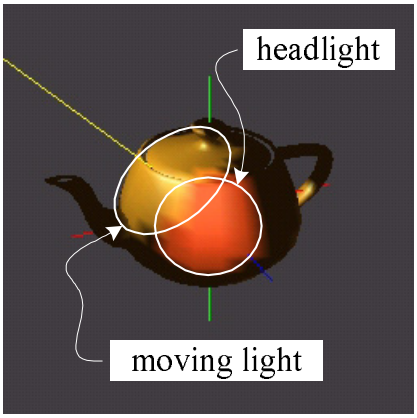


그림 3.22: 움직이는 광원과 카메라

문제는 이러한 카메라의 움직임이 광원에 어떻게 영향을 미치는가 하는 것인데, 앞에서도 설명한 바와 같이 광원의 위치나 방향과 같은 기하 속성을 정의하는 OpenGL 함수를 호출하면, 그러한 기하 속성들이 모델뷰 행렬 스택의 탑에 있는 행렬의 의하여 눈 좌표계로 변환이 되어 저장이 된 후 조명 계산에 사용이 된다. 두 개의 광원 중 우선 카메라에 고정된 헤드라이트부터 생각해 보자. 이 스폿 광원은 카메라의 위쪽에서 약간 아래로 향하고 있음을 관찰할 수가 있다. 즉 카메라를 기준으로 하는 눈 좌표계의 점 $(0.0\ 0.5\ 0.0)^t$에서 $(0.0\ -0.05\ -1.0)^t$의 방향으로 빛을 비추고 있는데(Line (b)와 그 다음 문장), 이 값들은 카메라가 어디에 위치하건 카메라에 상대적으로 고정된 값이어야 한다. 따라서 언뜻 보면 카메라가 움직일 때마다 이 광원에 대한 기하 속성을 다시 설정을 해야 할 것 같지만, 이 광원 인자는 눈 좌표계에서 한 번만 설정을 해주면 카메라가 어떻게 움직이건 다시 설정을 할 필요가 없다. 따라서 이 예제 프로그램에서는 `init_OpenGL()` 함수를 호출하여 OpenGL과 관련한 초기화 작업을 해줄 때, 뷰잉 변환을 위한 `gluLookAt(*);` 문장의 수행 직전에 눈 좌표계에서 헤

드라이트의 기하 속성을 설정하였다(Line (d)와 그 다음 문장).

**프로그램 예 3.3** 움직이는 광원과 카메라.

```
GLfloat light_position0[4] = {0.0, 5.0, 5.0, 1.0}; //
Line (a)
GLfloat spot_direction0[3] = {0.0, -5.0, -5.0};
     ⋮
GLfloat light_position1[4] = {0.0, 0.5, 0.0, 1.0}; //
Line (b)
GLfloat spot_direction1[3] = {0.0, -0.05, -1.0};
     ⋮
int translate_obj = 0, rotate_cam_angle = 0,
rotate_splgt_angle = 0;
float translate_in_y_tab[360], rotate_cam_tab[360][3];
     ⋮
void set_rotate_cam_tab(void) {
 int i;
 double angle_in_rad;

 for (i = 0; i < 360; i++) {
  angle_in_rad = i*TO_RAD;
  rotate_cam_tab[i][0] = 14.5*sin(angle_in_rad);
  rotate_cam_tab[i][1] = 8.0;
  rotate_cam_tab[i][2] = 14.5*cos(angle_in_rad);
 }
}

void next_frame(void) {
 translate_obj = (translate_obj + 1) % 360;
 if (rotate_spotlight)
  rotate_splgt_angle = (rotate_splgt_angle + 8) % 360;
 if (rotate_camera)
  rotate_cam_angle = (rotate_cam_angle + 1) % 360; //
Line (c)
 glutPostRedisplay();
}
```

```
void init_OpenGL(void) {
     ⋮ // projection transform and other initialization
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glLightfv(GL_LIGHT1, GL_POSITION, light_position1); //
Line (d)
  glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction1);
  gluLookAt(0.0, 8.0, 14.5, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  if (!rotate_camera && !rotate_spotlight) { // Line (e)
    glPushMatrix();
      // Line (f1)
      glRotatef((float) rotate_splgt_angle, 0.0, 1.0, 0.0);
      draw_light_info(); // Line (g)
    glPopMatrix();
  }
  else {
    if (rotate_camera) { // Line (h)
      glLoadIdentity();
      gluLookAt(rotate_cam_tab[rotate_cam_angle][0],
                rotate_cam_tab[rotate_cam_angle][1],
                rotate_cam_tab[rotate_cam_angle][2],
                0.0, 0.0, 0.0, 0.0, 1.0, 0.0); // Line (i)
    }
    glPushMatrix();
      // Line (f2)
      glRotatef((float) rotate_splgt_angle, 0.0, 1.0, 0.0);
      // Line (j)
      glLightfv(GL_LIGHT0, GL_POSITION, light_position0);
      glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION,
spot_direction0);
      draw_light_info();
    glPopMatrix();
  }
  draw_axes();
  draw_teapot();
```

```
  glutSwapBuffers();
}
```

다음 빠르게 회전하는 스폿 광원(GL_LIGHT0)의 기하 속성을 어떻게 설정할 것인가에 대하여 알아보자. 중요한 것은 OpenGL 시스템이 이 광원의 기하 속성을 눈 좌표계에서 올바르게 사용을 할 수 있도록 해주어야 한다는 점이다. 이러한 광원은 매 프레임에 대한 렌더링 계산을 하려고 할 때, 물주전자와 같이 하나의 기하 물체로 생각하여, 적절한 모델링 및 뷰잉 변환을 통하여 그 위치와 방향을 눈 좌표계로 보내주면 된다. 이제는 잘 알다시피 이러한 과정에 영향을 미치는 것은 바로 모델뷰 행렬 스택의 탑에 있는 행렬의 내용을 구성하는 모델링 변환 $M_M$과 뷰잉 변환 $M_V$이다. 개념적으로 이 광원은 세상 좌표계에서 회전을 하고 있기 때문에, 우선 각 프레임에 대한 광원의 위치 정보(rotate_splgt_angle)를 구해 세상 좌표계로 보내주어야 한다. Line (a)와 그 다음 문장에서 설정하는 위치와 방향은 이 광원에 대한 모델링 좌표계에서의 의미를 가지는데, Line (f1)와 Line (f2)의 회전 변환에 해당하는 모델링 변환 $M_M$에 의해 세상 좌표계의 위치로 변환이 되고, 이는 계속해서 Line (i)에서 설정되는 뷰잉 변환 행렬 $M_V$에 의해 눈 좌표계의 값으로 변환이 된다.

여기서는 디스플레이 컬백 함수인 display()에서 현재 프레임에 대한 렌더링을 할 때, 두 가지 경우로 나누어 생각을 하였다. 첫째로 광원과 카메라가 고정이 되어 있고 물주전자만 움직인다면(Line (e)), 이전 프레임과 비교하여 $M_M$과 $M_V$가 변하지 않았고, 눈 좌표계에서의 광원의 위치와 방향이 변함이 없기 때문에 광원에 대한 성질을 다시 설정할 필요가 없이 단지 광원에 대한 정보를 다시 그리기만 하면 된다(Line (g)).

반면 그렇지 않은 경우 만약 카메라가 회전중이라면(Line (h)), 모델뷰 행렬 스택의 탑에 $M_V$를 수정하여 다시 올려놓고(Line (i)), 회전하는 스폿 광원에 대한 모델링 변환을 한 후(Line (f2)), 이 광원의 기하 성질을 다시 설정을 하고 있다(Line (j)와 그 다음 문장). 물론 이 광원이 회전을 하고 있지 않는 경우에도 $M_V$가 변했기 때문에 $M_M$에 대한 변환을 다시 해주어야 한다. 결론적으로 말해서 광원의 기하 속성을 설정하려 할 때, 그 좌표 값의 의미를 정확하게 분석을 하여, 어떤 좌표계에서의 광원인가를 파악하면 광원에 대한 조작 역시 물체에 대한 기하 변환과 같은 방식으로 어렵지 않게 수행을 할 수가 있을 것이다.

## 6.5 OpenGL의 조명 공식

이제 OpenGL 시스템에서 실제로 사용되는 조명 공식에 대하여 구체적으로 알아보기 전에, 266쪽의 그림 3.2에 도시되어 있는 라이팅 모듈 부분에 대한 전체적인 상황을 다시 상기해보자. 프로그래머가 다면체 모델에 대한 여러 성질(기하 프리미티브 타입, 꼭지점 좌표, 법선 벡터, 꼭지점 색깔, 물질 정보, 텍스춰 좌표, 에지 플래그 등)과 조명 계산에 관련된 정보들을 OpenGL 함수를 통하여 설정하였을 때, 이러한 정보들이 눈 좌표계로 흘러와 각 꼭지점에 대하여 색깔, 텍스춰 좌표, 에지 플래그 등 관련 데이터들이 연관되어 각 기하 프리미티브 단위로 조립이 된 후 계속해서 렌더링 파이프라인으로 흘러간다. 이러한 과정에서 라이팅 계산을 하라고 명시적으로 설정을 하였을 때, 퐁의 조명 모델에 기반을 둔 OpenGL의 조명 공식을 사용하여 꼭지점에 연관시킬 색깔을 계산을 한다.

OpenGL의 조명 공식은 쉐이딩을 하려는 지점의 꼭지점 좌표 $V$와 그 점에서의 법선 벡터 $\mathbf{n}$과 표 3.2, 표 3.3, 그리고 표 3.4에 요약된 조명 인자를 사용하여 계산하게 된다. 광원이 여러 개가 있을 경우에 지역적인 반사 효과는 각 광원에 대한 효과

를 더하게 되는데, 우선 $i$번째 광원($0 \leq i \leq n-1$)의 지역적인 효과에 대해 살펴보자. OpenGL에서는 앞에서도 설명한 바와 같이 한 광원이 물체에 직접 영향을 미치는 과정을 다음과 같이 앰비언트 반사, 난반사, 그리고 정반사 등 세 가지로 나누어 생각을 한다.

$$\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli} \tag{3.2}$$

조명 공식을 적용하면 그 결과 RGBA 색깔이 구해지는데, OpenGL에서는 R, G, B 채널과 A 채널에 대하여 다른 계산 방식을 취한다. 여기서는 우선 네 개의 채널을 가지는 조명 인자의 색깔 값 중 R, G, B 채널에 대한 계산 방법에 대하여 알아보자. OpenGL의 조명 공식에서 RGB 색깔 타입간의 곱셈은 각 채널간의 곱셈을 의미하고, 상수와 색깔 타입과의 곱셈은 그 상수와 각 채널간의 곱셈을 의미한다. $\mathbf{a}_{cm} * \mathbf{a}_{cli}$는 $i$번 광원에 대한 지역 앰비언트 반사 색깔로서, 앞에서도 기술한 바와 같이 OpenGL에서는 각 광원에 대한 앰비언트 반사를 전역적인 것과 지역적인 것으로 나누어 생각을 한다.

다음 $(\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli}$는 식 (3.1)에서의 난반사 색깔 $I_{l_i\lambda} \cdot k_{d\lambda} \cdot (N \cdot L_i)$에 대응되는 것으로서 그에 대한 의미는 쉽게 추측할 수가 있다. 약간은 지루하게 느껴질 수 있으나 조명 공식에서 사용되는 기호를 하나씩 살펴보면서 OpenGL에서의 조명 계산 방식을 정확하게 이해하도록 해보자. 3차원 공간에서의 두 개의 동차 좌표 $\mathbf{P}_1$과 $\mathbf{P}_2$가 주어졌을 때, $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$가 어떤 벡터를 나타내는지는 다음과 같이 정의된다. $\mathbf{P}_1$과 $\mathbf{P}_2$의 $w$ 좌표(네 번째 좌표)를 각각 $w_1$과 $w_2$라 할 때 1. 이 값이 모두 0이 아니면(즉 두 점이 모두 아핀 공간에서 유한 거리만큼 떨어진 점을 의미하면), $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$는 점 $\mathbf{P}_1$에서 $\mathbf{P}_2$로 향한 방향으로 길이가 1인 단위 벡터를 의미한다. 2. 만

약 $w_1$은 0이 아니고 $w_2$가 0이라면(즉 $\mathbf{P}_2$가 아핀 공간에서 $(x\ y\ z)$인 벡터를 의미한다면), $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$는 $\mathbf{P}_2$에 해당하는 벡터의 길이를 1로 만들어 준 단위 벡터를 나타낸다. 3. 반면 $w_1$이 0이고 $w_2$가 0이 아니라면, $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$는 $\mathbf{P}_1$이 나타내는 방향의 반대 방향에 대하여 길이가 1인 벡터를 의미하고, 4. 마지막으로 $w_1$과 $w_2$가 모두 0이면, $\mathbf{P}_2 - \mathbf{P}_1$이 가리키는 방향으로의 단위 벡터를 나타낸다.

이러한 정의를 통하여 난반사 관련 부분에서의 $\overrightarrow{\mathbf{VP}}_{pli}$의 의미를 살펴보자. $\mathbf{V}$는 현재 조명 계산을 하려하는 꼭지점의 좌표이므로 $w$ 좌표는 0이 아니라고 가정을 해도 무방하다[18]. $\mathbf{P}_{pli}$는 광원의 위치(점 광원을 사용한다면 $\mathbf{P}_{pli}$의 $w$ 좌표가 0이 아님) 또는 방향(평행 광원을 사용한다면 $\mathbf{P}_{pli}$의 $w$ 좌표가 0임)을 나타내는데, 이 정의에 의하면 어떤 종류의 광원을 사용하건 $\overrightarrow{\mathbf{VP}}_{pli}$는 광원에서 빛이 들어오는 방향의 반대 방향에 대하여 길이가 1인 벡터, 즉 식 (3.1)에서의 L에 해당이 됨을 알 수 있다. 두 벡터의 내적 $\odot$는 앞에서 정의를 하였는데, OpenGL에서의 내적 $(\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})$의 계산 방식을 통하여 앞에서 살펴본 $N \cdot L$에 대하여 양수일 경우에만 그 값을 취하고, 그 이외의 경우에는 0 값을 취하는 방식을 사용함을 알 수가 있다. 이는 OpenGL에서는 기본적으로 뒤에서 들어오는 빛은 고려를 하지 않음을 의미한다[19].

다음 $i$번째 광원에 대한 정반사 색깔 $(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}$에 대하여 생각을 해보자. 이 식은 본질적으로 식 (3.1)에서의 $I_{l_i\lambda} \cdot k_{s\lambda} \cdot (N \cdot H_i)^n$과 같다고 보면 되는데, 여기서 해프웨이 벡터 $\mathbf{h}_i$는 다음과 같이 정의된다.

$$\mathbf{h}_i = \begin{cases} \overrightarrow{\mathbf{VP}}_{pli} + \overrightarrow{\mathbf{VP}}_e, & v_{bs} = \texttt{TRUE}, \\ \overrightarrow{\mathbf{VP}}_{pli} + (0\ 0\ 1\ 0)^t, & v_{bs} = \texttt{FALSE} \end{cases}$$

[18] 실제로 OpenGL 규약에 의하면 이 값이 0인 경우에는 조명 계산의 결과가 정의가 되지 않는다.
[19] 양면 조명을 사용하면 뒷면에 대해서는 법선 벡터의 방향을 반대로 바꾸어 사용함을 상기하자.

해프웨이 벡터는 앞에서 광원에 대한 방향과 관찰자 방향의 중간 방향으로의 단위 벡터로 정의가 되었다($\frac{L+V}{|L+V|}$). 위의 정의에서 $\overrightarrow{\mathbf{VP}}_{pli}$가 광원에 대한 방향인데, 관찰자 방향은 지역 관찰자를 사용하는지 여부에 따라 다르게 선택이 된다. 만약 조명 모델에서 지역 관찰자를 사용한다면($v_{bs} = \texttt{TRUE}$), 이는 관찰자가 눈 좌표계의 원점 $\mathbf{P}_e = (0\ 0\ 0\ 1)^t$에 있는 상황이므로, 꼭지점 좌표 $\mathbf{V}$에서 원점으로 향한 벡터 $\overrightarrow{\mathbf{VP}}_e$가 관찰자 방향이 되고, 만약 무한 관찰자를 선택하면($v_{bs} = \texttt{FALSE}$), 눈 좌표계에서 양의 $z_e$축 방향인 $(0\ 0\ 1\ 0)^t$이 관찰자 방향으로 사용이 된다. 해프웨이 벡터는 단위 벡터이어야 하므로, $\hat{\mathbf{h}}_i$가 실제로 사용이 되는 해프웨이 벡터가 된다[20].

따라서 $(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}$에서의 $(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}$의 의미는 분명한데, 여기서 $f_i$는 약간의 설명이 필요하다. $f_i$는 0 또는 1 값을 가지는 변수로서 $\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli}$가 0보다 크면 1 값을, 아니면 0 값을 가진다. 이 내적 값이 0보다 크다는 사실은 두 벡터 $\mathbf{n}$과 $\overrightarrow{\mathbf{VP}}_{pli}$간의 각도가 90도보다 작음을 뜻한다. 법선 벡터 $\mathbf{n}$은 물체 표면에서 바깥 쪽을 향하기 때문에, 이는 광원이 물체 표면에 대하여 앞쪽에서 빛을 비추고 있음을 의미한다. 다시 말하면 정반사도 난반사의 경우에서처럼 광원이 표면의 앞쪽에서 빛을 비출 경우에만 반사 색깔을 더하게 됨을 암시하고 있다.

이제 $i$번째 광원에 대한 앰비언트 반사, 난반사, 정반사 값을 모두 더해보면 식 (3.2)와 같이 되는데, 이 값은 물체에 대한 광원의 직접적인 효과라 하기에는 아직 불충분하다. 실제로는 다음과 같이 두 개의 값 $att_i$와 $spot_i$를 곱해 이를 $i$번째 광원이 물체에 직접적으로 영향을 미치는 반사 색깔로 사용을 한다.

$$(att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}] \quad (3.3)$$

[20] 여기서 임의의 벡터 $\mathbf{d}$에 대하여 $\hat{\mathbf{d}}$는 그 벡터의 길이를 1로 만들어준 단위 벡터를 의미한다.

여기서 $att_i$는 빛의 감쇠 효과를 위한 값으로서, OpenGL에서는 다음과 같이 정의가 된다.

$$att_i = \begin{cases} \frac{1}{k_{0i} + k_{1i}\| \mathbf{VP}_{pli} \| + k_{2i}\| \mathbf{VP}_{pli} \|^2}, & \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise} \end{cases}$$

이 정의에서 두 번째 경우, 즉 $\mathbf{P}_{pli}$의 $w$ 좌표가 0이라는 것은 이 광원이 평행 광원임을 의미하는데, 이러한 경우에는 $att_i$는 1.0이 되어 식 (3.3)에서 아무런 영향을 미치지 않는다. 빛의 감쇠 효과는 광원과 현재 쉐이딩을 하려는 지점간의 거리에 의존하기 때문에, 무한 거리만큼 떨어진 평행 광원에 대해서는 빛의 감쇠 효과를 내지 못함은 당연하다고 하겠다. 반면 점 광원을 사용을 할 때는($\mathbf{P}_{pli}$'s $w \neq 0$), 임의의 벡터 $\mathbf{d}$에 대하여 $\| \mathbf{d} \|$는 이 벡터의 길이라고 할 때, $\| \mathbf{VP}_{pli} \|$는 쉐이딩 지점에서 광원까지의 거리에 해당이 된다. 따라서 앞에서 설명한 바와 같이 OpenGL에서도 거리에 대한 이차식의 역수를 사용하여 점 광원에 대한 감쇠 효과를 낸다. 만약 프로그램에서 빛의 감쇠 효과에 대하여 명시적으로 설정을 하지 않으면 어떤 결과가 발생할까? 빛의 감쇠 인자의 디폴트 값은 $k_{0i} = 1$이고 $k_{1i} = k_{2i} = 0$이므로, OpenGL에서는 디폴트로 빛의 감쇠 효과 기능을 사용하지 않는다.

다음 $spot_i$는 $i$번째 광원이 스폿 광원일 경우에 이를 처리하기 위한 것인데 이 값은 다음과 같이 정의가 된다.

$$spot_i = \begin{cases} (\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0 \ \& \ \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos c_{rli}, \\ 0.0, & c_{rli} \neq 180.0 \ \& \ \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos c_{rli}, \\ 1.0, & c_{rli} = 180.0 \end{cases}$$

스폿 광원의 절단 각도 $c_{rli}$의 디폴트 값은 일반 점 광원을 사용하는 것에 해당하는

180.0이고, 이 때의 $spot_i$ 값은 1.0이므로 프로그램에서 이 값을 0.0과 90.0 사이의 값으로 설정하여 스폿 광원을 사용하겠다고 명시적으로 밝히지 않는 한 스폿 광원 효과는 나지 않게 된다. 만약 스폿 광원을 사용을 하겠다고 설정을 하면($c_{rli} \neq 180.0$), 두 가지 경우로 나누어 생각을 할 수 있는데, 이에 대하여 314쪽의 그림 3.16을 다시 한번 보면서 생각해 보자. 스폿 조명 효과는 점 $\mathbf{V}$가 절단 각도 범위 내에 들어올 경우에만 적용이 된다. 따라서 스폿 조명의 범위 밖에 떨어질 경우에는 이 광원에 대한 반사 색깔은 각 채널이 0인 검은색이어야 한다. 그림 3.16을 보면 알 수 있듯이 $\overrightarrow{\mathbf{P}_{pli}\mathbf{V}}$는 조명의 위치에서 꼭지점을 향한 방향에 대한 단위 벡터이고, 스폿 조명의 중심축 방향에 해당하는 단위 벡터 $\hat{\mathbf{s}}_{dli}$와의 내적을 취한 값 $\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli}$는 $\cos\psi$ 값을 가진다. 만약 $\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos c_{rli}$이라면 이는 점 $\mathbf{V}$가 스폿 조명의 범위 밖에 있는 상황을 의미하기 때문에, $spot_i$는 0.0 값을 가지고 따라서 이 광원은 이 지점에 대하여 아무런 기여를 하지 않게 된다. 만약 $\mathbf{V}$가 스폿 조명 범위에 들어오면($\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos c_{rli}$), OpenGL에서는 앞에서도 설명한 바와 같이 주변으로 갈 수록 어두운 효과를 내기 위하여 $\cos^{s_{rli}}\psi$에 해당하는 값, 즉 $(\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}$을 사용하여 스폿 조명 효과를 내준다.

이렇게 하여 한 개의 광원이 물체 표면에 직접적으로 미치는 영향을 어떻게 수식으로 표현할 지에 대하여 살펴보았다. $n$개의 광원이 있다면 각 광원에 대한 반사 색깔을 다 더한 후($\sum_{i=0}^{n-1}(att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}]$), 전역 앰비언트 반사($\mathbf{a}_{cm} * \mathbf{a}_{cs}$)와 물질의 방사 색깔($\mathbf{e}_{cm}$)을 더하면 다음과 같은 OpenGL의 기본 조명 공식을 얻게 된다.

$$\begin{aligned}\mathbf{c} \;=\; & \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \\ & \sum_{i=0}^{n-1}(att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}]\end{aligned}$$

앞에서 라이팅 모듈에서는 기하 프리미티브의 각 꼭지점에 대하여 주 색깔($\mathbf{c}_{pri}$)과 보조 색깔($\mathbf{c}_{sec}$) 두 가지의 색깔이 붙여진다고 하였다. 조명 계산을 할 경우에는 정반사 색깔을 분리할지 여부에 따라 그 결과 색깔이 달라진다. 만약 분리를 안 한다면($c_{es} = \mathsf{SINGLE\_COLOR}$), $\mathbf{c}_{pri}$의 RGB 채널은 위에서 표현한 $\mathbf{c}$, 그리고 A 채널은 물질의 난반사 색깔 $\mathbf{d}_{cm}$의 A 채널 값이 된다. 또한 $\mathbf{c}_{sec}$은 A 채널 포함 (0, 0, 0, 0)이 된다. 만약 정반사 색깔을 분리한다면($c_{es} = \mathsf{SEPARATE\_SPECULAR\_COLOR}$), 두 색깔의 RGB 채널은 다음과 같고,

$$\begin{aligned}
\mathbf{c}_{pri} &= \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \sum_{i=0}^{n-1} (att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli}], \\
\mathbf{c}_{sec} &= \sum_{i=0}^{n-1} (att_i)(spot_i)(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli},
\end{aligned}$$

A 채널에는 각각 $\mathbf{d}_{cm}$의 A 채널 값과 0이 지정이 된다.

**잡깐 3.2** 지금까지 자세히 살펴본 조명 공식에서는 여러 개의 반사 색깔을 더해서 최종 색깔을 구한다. OpenGL에서는 각 채널이 0과 1사이의 값으로 정규화된 색깔 값을 사용하는데, 경우에 따라서 어떤 특정 채널의 값이 1보다 커질 수가 있다. 예를 들어, 붉은 계통의 물체에 대하여 여러 개의 흰색 조명을 사용할 때 붉은 색깔이 여러 번 더해지게 되는데, 결과적으로 계산한 색깔이 (2.1, 0.9, 0.2)라 가정해보자. OpenGL에서는 조명 계산을 한 후, 두 가지 결과 색깔의 각 채널에 대하여 0보다 작은 값은 0으로, 그리고 1보다 값은 무조건 1값으로 바꾸어 준다. 따라서 지금과 같은 경우에 이 색깔은 (1.0, 0.9, 0.2)가 되어 원래의 붉은 색깔과는 약간 다른 색깔로 변하게 된다. 따라서 조명 인자를 설정할 때 이러한 일이 발생하지 않도록 조심을 하여야 한다. 참고로 이러한 문제를 해결하는 다른 방법은 어떤 채널이 1보다 커질

때 각 채널을 가장 큰 채널 값으로 나누는 것이다. 따라서 위의 예에서 결과 색깔은 (1.0, 0.429, 0.095)가 되는데, 이러한 방법이 계산된 색깔의 색조를 보존하는데 있어 더 좋은 방법이 될 수도 있다.

## 6.6 고정된 조명 공식과 프로그램 가능한 조명 계산

지금까지 OpenGL의 라이팅 계산 과정에 대하여 살펴보았는데, OpenGL에서는 조명 계산을 하는데 있어 '고정된 조명 공식'을 사용한다고 할 수가 있다. 즉 조명 공식의 변수에 해당하는 조명 인자 값들을 OpenGL 함수로 설정을 하면, 렌더링 파이프라인의 라이팅 모듈에서 현재 설정되어 있는 값들을 사용하여 '미리 고정된' 공식을 적용하여 조명 계산을 한다. '미리 고정된' 조명 공식을 사용하면 이를 소프트웨어나 하드웨어로 구현할 때 모든 최적화 기법을 사용하여 효율적으로 구현할 수가 있는 반면, 이러한 방식을 취한다면 라이팅 계산을 하는데 있어 상당한 제약을 받는다. 즉 OpenGL 시스템의 경우 프로그래머는 단지 '고정된' 조명 모델에서 제공하는 옵션을 선택할 뿐이기 때문에 조명 공식에서 제공하는 기능 외의 다른 방식의 라이팅 계산을 하기가 어려워진다. 물론 최근에 OpenGL의 렌더링 파이프라인을 사용하여 기본적으로 제공되는 렌더링 기능 이외의 라이팅 계산을 수행하는 렌더링 기법들이 등장하고 있으나, '미리 고정된' 공식을 사용하기 때문에 그러한 계산을 하는데 있어 상당한 제한이 있다고 할 수 있다.

한편 종종 다른 렌더링 시스템에서는 라이팅 계산을 하는데 있어 '미리 고정된' 방식을 사용하는 것이 아니라, 프로그래머가 라이팅 계산을 원하는 방식으로 수행할 수 있도록 해주는 툴을 제공한다. 이러한 개념을 처음으로 도입한 렌더링 시스템은 바로 토이 스토리 등과 같은 영화를 제작하는데 사용된 Pixar의 RenderMan

소프트웨어인데[21], 여기서는 하나의 고정된 방식이 아니라 프로그래머가 쉐이딩 언어(Shading Language)라고 하는 C 스타일의 프로그래밍 언어로 쉐이더(shader)라는 프로시져를 작성하여 라이팅 과정 전반에 대하여 프로그래밍을 할 수가 있다. 따라서 한 개의 라이팅 계산 방식을 사용하는 OpenGL과는 달리 RenderMan 시스템에서는 매우 다양한 형태의 라이팅 계산을 할 수가 있다.

RenderMan에서는 다음과 같이 다섯 종류의 쉐이더를 지원한다.

- **표면 쉐이더(surface shader):** 이 쉐이더는 물체 표면에 입사하는 빛을 어떻게 반사시킬 것인가를 프로그래밍하는데 사용된다.

- **변위 쉐이더(displacement shader):** 이 쉐이더는 쉐이딩을 하려하는 물체의 지점에서의 기하 속성, 즉 위치나 법선 벡터 등을 변화시켜 물체 표면에 주름이나 울퉁불퉁한 범프(bump) 효과 등을 내는데 사용이 된다.

- **광원 쉐이더(light shader):** 이 쉐이더는 그 이름이 암시하듯이 광원의 성질, 즉 위치, 방향, 밝기, 색깔, 빛을 비추는 방식 등을 프로그래밍하는데 사용이 된다.

- **공간 쉐이더(volume shader):** 이 쉐이더는 빛이 광원에서 물체로 입사할 때나 물체에서 빛이 반사되어 카메라로 들어올 때 통과하는 공간이 빛의 흐름에 어떻게 영향을 미치는가를 기술하는데 사용이 되는데, 안개나 아지랭이와 같은 환경 효과(environmental effect)를 프로그래밍할 수가 있다.

- **이미징 쉐이더(imager shader):** 이 쉐이더는 빛이 카메라의 필름에 투영이 되었을 때, 그 결과 래스터 이미지를 출력하기 전에 각 픽셀 단위로 적절한 처

[21] S. Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics,* Addison-Wesley, 1990.

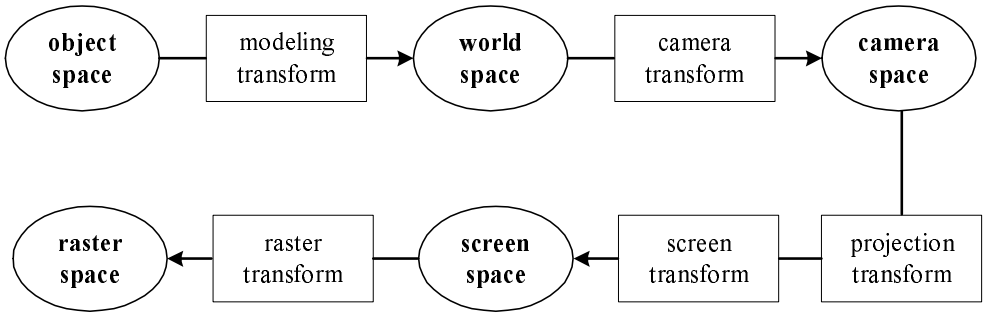



그림 3.23: RenderMan 시스템의 좌표계 및 기하 변환

리를 할 수 있도록 하는데 사용이 된다.

RenderMan 시스템에서는 각 쉐이더의 타입마다 몇 가지씩 표준 쉐이더를 제공하는데, 이 외에도 프로그래머가 원하는 방식으로 쉐이더를 프로그래밍하여 사용을 할 수가 있다. RenderMan 쉐이더에 대한 설명은 이 책의 범위를 벗어나지만, 프로그래밍을 통한 라이팅 계산은 컴퓨터 그래픽스 렌더링에 있어 매우 중요한 개념 중의 하나이기 때문에 간단한 예를 통하여 살펴보도록 하겠다. RenderMan 프로그램은 357쪽의 프로그램 예 3.4에 주어진 코드와 같이 RenderMan Interface(RMI)에서 제공하는 함수와 쉐이딩 언어를 사용하여 작성을 한다[22]. 라이팅 계산에 대하여 알아보기 전에 프로그램의 전체 골격에 해당하는 뷰잉 관련 부분을 살펴보면, RenderMan 시스템에서도 OpenGL과 마찬가지로 그리려고 하는 기하 물체 및 그에 관련된 정보들이 여러 좌표 공간을 거쳐가면서 렌더링 계산이 수행이 됨을 알 수 있다. RenderMan의 기하 파이프라인에서는 그림 3.23에 도시된 바와 같이 다섯 개의 좌표계를 사용한다. 물체 공간(object space)과 세상 공간(world space)은 각각 OpenGL에서의 물체 좌표계를 구성하는 모델링 좌표계와 세상 좌표계에 해당한다[23]. 카메라 공간(camera space)은 OpenGL의 눈 좌표계에 대응이 되며, 마지

[22] *The RenderMan Interface, Version 3.1*, Pixar, 1989.

[23] 여기서는 OpenGL 시스템을 설명할 때와는 달리 좌표계(coordinate system)대신에 공간(space)이라는 용어를 사용하겠다. 이는 단지 두 시스템의 좌표 공간간의 혼란을 막기 위한 것

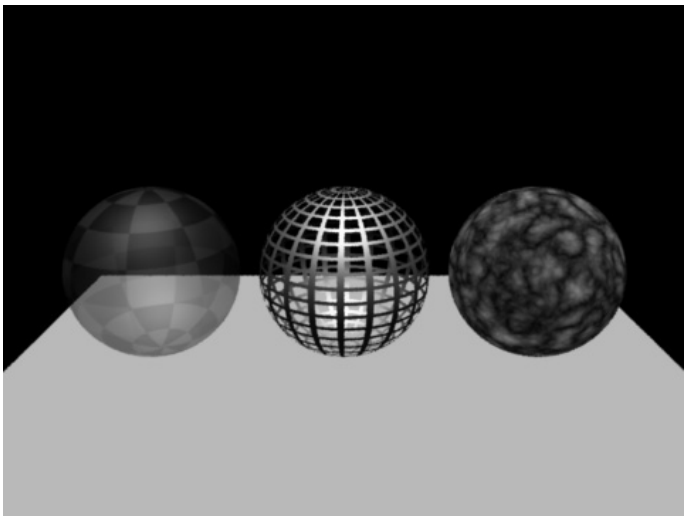


그림 3.24: RenderMan을 사용한 렌더링 예

막 래스터 공간(raster space)은 개념적으로 윈도우 좌표계에 해당이 된다고 생각하면 된다. 그 중간의 스크린 공간(screen space)은 아직 래스터 공간으로 래스터화가 되기 직전의 정규화된 2차원 화면에 해당하는 좌표계로서, 3차원으로 정규화된 OpenGL의 정규 디바이스 좌표계 상의 정규 윈도우에 해당한다고 생각하면 된다. RenderMan 시스템에서의 각 좌표 공간간의 기하 변환에 대한 이름이 그림 3.23에 도시되어 있는데, OpenGL에서의 기하 파이프라인을 생각해 보면 그 의미를 어렵지 않게 이해할 수 있을 것이다.

이제 프로그램 예 3.4의 코드를 살펴보자. RenderMan은 기본적으로 실시간 렌더링을 위한 렌더링 툴이 아니고, 비교적 사실적인 영상의 생성을 통하여 다양한 애니메이션을 제작하는데 그 목적이 있다. 따라서 보통 이미지를 생성한 후 화면의 윈도우 안에 도시하는 OpenGL과는 달리 RenderMan에서는 일반적으로 각 프레임을 화일에 저장을 한다. 이 프로그램에서는 단순히 그림 3.24에 주어진 한 장의 이미지를 생성을 한다. Line (b)의 `RiFrameBegin(1);` 문장은 1번 프레임을 생

으로서 본질적으로 어떤 용어를 사용하건 같은 개념을 의미한다.

성하는데 필요한 코드가 시작이 된다는 것을 의미한다. Line (a)의 `RiFormat(*)` 함수는 래스터 변환(raster transformation)에 영향을 미치는 함수로서, 생성하려는 이미지의 해상도를 결정한다. 실제로 RenderMan에서는 좀 더 자세하게 래스터 변환을 조절할 수 있는데, 여기서는 기하 변환에 대한 상세한 설명은 생략하도록 하겠다. 이 프로그램에서는 Line (b)의 다음 문장의 `RiDisplay(*)` 함수를 통하여 생성하고자 하는 이미지를 (`"rgba"`) 타입의 형태로 `"3spheres.tif"`라는 이름을 가지는 화일(`"file"`)로 저장을 함을 알 수 있다. 그 다음 문장의 `RiProjection(*)` 함수는 그 이름이 암시하듯이 사용할 투영 변환(projection transformation)을 결정한다. 다음 연달아 나오는 이동 변환과 회전 변환(Line (c)와 그 다음 문장)은 OpenGL에서의 뷰잉 변환에 해당함을 쉽게 추측할 수 있는데, RenderMan에서는 이를 카메라 변환(camera transformation)이라 부른다.

한 장의 이미지 프레임을 생성하기 위해서는 여러 가지의 렌더링 인자가 영향을 미친다. RenderMan에서는 한 프레임을 계산하는데 있어 고정이 되는 인자들을 옵션(option)이라 하는데, 여기에는 카메라의 성질을 결정하는데 사용이 되는 카메라 옵션, 그리고 최종적으로 생성될 래스터 이미지의 형식에 관련된 디스플레이 옵션, 그리고 이미지 전반에 걸쳐 영향을 미치는 인자(예를 들어 한 화소 당 몇 개의 샘플링을 할 것인가 등등) 등이 포함이 된다. 반면에 한 장의 이미지를 생성하는데 있어 동적으로 변하는 인자들을 속성(attribute)라 부르는데, 색깔, 불투명도, 라이팅 계산에 사용하는 쉐이더 이름 등과 같은 라이팅 인자, 방향성, 바운딩 박스, 세분 레벨(subdivision level) 등과 같이 각 기하 물체를 그리는데 영향을 미치는 인자, 그리고 모델링 변환을 저장하는 현재 변환 행렬 스택(current transformation matrix stack) 등이 포함된다.

Line (d)에서 `RiWorldBegin()` 함수를 호출하면, 카메라 변환을 포함한 모든 Ren-

derMan 옵션이 고정이 된 상태에서 모델링 변환(modeling transformation)을 통하여 기하 물체를 그리기 위한 준비를 마치게 된다. 그림 3.24를 보면 이 프로그램에서는 바닥에 해당하는 직사각형 한 개와 세 개의 구 등 모두 네 개의 기하 물체를 그리고 있는데, Line (g), Line (i), Line (l), 그리고 Line (n)에서의 `RiAttributeBegin()` 함수와 그에 대응되는 `RiAttributeEnd()` 함수에 의해 구획지어지는 네 개의 블럭 안에서 각 기하 물체를 그리고 있음을 알 수 있다. 이 두 함수의 용도는 OpenGL에서의 glBegin()과 glEnd() 함수를 생각해 보면 쉽게 유추할 수가 있다. 전자는 스택의 푸쉬 연산을 통하여 현재 설정되어 있는 RenderMan의 모든 속성을 보존을 해주고, 후자는 스택의 팝 연산을 통하여 직전에 푸쉬한 내용으로 복원을 해주는데, 물론 스택의 탑에 있는 '현재 속성'을 적절히 바꿔가면서 렌더링을 한다. 이 두 함수 사이의 코드를 보면 알 수 있듯이 각 기하 물체를 그리는데 필요한 속성을 동적으로 설정하면서 `RiPolygon(*)`과 `RiSphere(*)` 함수를 통하여 해당 물체를 그리고 있다. 또한 세 개의 구를 그릴 때마다 호출하는 `RiTranslate(*)` 함수는 바로 모델링 변환에 해당함은 쉽게 유추할 수가 있다.

**프로그램 예 3.4** RenderMan 프로그래밍의 예.

```
#include <stdio.h>
#include <math.h>
#include "ri.h"

void main(void) {
  RtFloat fov = 45, intensity1 = 0.08, intensity2 = 0.8;
  RtPoint from = {0.0, 1.0, 4.0}, to = {0.0, 0.0, 0.0};
  RtColor fcolor = {0.85, 0.85, 0.85};
  RtPoint floor[4] = {-5.0, -5.0, 0.0,   5.0, -5.0, 0.0,
         5.0, 5.0, 0.0,   -5.0, 5.0, 0.0};
  RtColor sphcolor1 = {1.0, 1.0, 1.0};
  RtColor sphcolor2 = {0.797, 0.498, 0.196};
```

```
RtColor sphcolor3 = {0.439, 0.859, 0.576};
RtFloat Ka1 = 1.0, Kd1 = 0.5, Ks1 = 1.0, roughness1 = 0.1;
RtFloat Ka2 = 1.0, Kd2 = 0.9;
RtFloat Ka3 = 1.0, Kd3 = 0.8, Ks3 = 1.0;
RtColor speccolor = {1.0, 1.0, 1.0};
RtColor opacity1 = {0.3, 0.3, 0.3};
char *txtname = "redchecker";


RiBegin(RI_NULL);
RiFormat(480, 360, 1); // Line (a)
RiPixelSamples(2, 2);
RiFrameBegin(1); // Line (b)
 RiDisplay("3spheres.tif", "file", "rgba", RI_NULL);
 RiProjection("perspective", "fov", &fov, RI_NULL);
 RiTranslate(0.0, -2.0, 8.0); // Line (c)
 RiRotate(-110.0, 1, 0, 0);
 RiWorldBegin(); // Line (d)
  RiLightSource("ambientlight", "intensity",
        &intensity1, RI_NULL); // Line (e)
  RiLightSource("distantlight", "intensity", &intensity2,
        "from", from, "to", to, RI_NULL); // Line (f)
  RiAttributeBegin(); // Line (g)
   RiColor(fcolor);
   RiSurface("matte", RI_NULL); // Line (h)
   RiPolygon(4, RI_P, (RtPointer) floor, RI_NULL);
  RiAttributeEnd();

  RiAttributeBegin(); // Line (i)
   RiTranslate(-2.25, 0.0, 2.0);
   RiColor(sphcolor1);
   RiOpacity(opacity1); // Line (j)
   RiMakeTexture("./texture/checker.tif", txtname,
      RI_PERIODIC, RI_PERIODIC,
      (RtFloatFunc) RiGaussianFilter, 2.0, 2.0, RI_NULL);
   RiShadingRate(0.25);
   RiShadingInterpolation("constant");
   RiDeclare("specularcolor", "varyingcolor");
   RiSurface("paintedplastic", "Ka", &Ka1, "Kd", &Kd1,
      "Ks", &Ks1, "roughness", &roughness1,
      "specularcolor", speccolor, "texturename",
```

```
            (RtPointer) &txtname, RI_NULL); // Line (k)
          RiSphere(1.0, -1.0, 1.0, 360.0, RI_NULL);
        RiAttributeEnd();

        RiAttributeBegin(); // Line (l)
          RiTranslate(0.0, 0.0, 2.0);
          RiColor(sphcolor2);
          RiShadingRate(0.25);
          RiShadingInterpolation("constant");
          RiSurface("screen", "Ka", &Ka2, "Kd", &Kd2); //
    Line (m)
          RiSphere(1.0, -1.0, 1.0, 360.0, RI_NULL);
        RiAttributeEnd();

        RiAttributeBegin(); // Line (n)
          RiTranslate(2.25, 0.0, 2.0);
          RiColor(sphcolor3);
          RiShadingRate(0.25);
          RiShadingInterpolation("constant");
          RiSurface("granite", "Ka", &Ka3, "Kd", &Kd3,
             "Ks", &Ks3, RI_NULL); // Line (o)
          RiSphere(1.0, -1.0, 1.0, 360.0, RI_NULL);
        RiAttributeEnd();
      RiWorldEnd();
    RiFrameEnd();
    RiEnd();
    }
```

지금까지 간략하게 RenderMan 프로그램의 뷰잉 구조에 대하여 살펴보았는데, 물론 상세한 부분에 있어 약간의 차이는 있으나 전체적인 구조는 OpenGL의 뷰잉과 거의 유사하다고 할 수 있다. 이제 쉐이더를 통한 라이팅 계산에 대하여 살펴보자. OpenGL에서는 OpenGL 함수의 호출을 통하여 라이팅 계산에 사용될 각 인자의 값들을 설정하였다. 반면에 RenderMan 시스템에서는 RenderMan 인터페이스 함수의 호출을 통하여 라이팅 계산 모델의 각 부분을 정의해주는 쉐이더 함수를 설정하고,

또한 그에 필요한 인자 값을 설정하면, 그렇게 설정된 쉐이더와 인자 값을 사용하여 라이팅 계산이 수행이 된다. 우선 라이팅 계산에 있어 가장 중요한 인자 중의 하나인 광원의 설정에 대하여 생각해 보면, Line (e)와 Line (f)의 `RiLightSource(*)` 함수는 라이팅 계산에서 사용할 광원에 대한 쉐이더를 지정을 한다. OpenGL에서와 마찬가지로 RenderMan에서도 현재 광원 리스트(current light source list)라는 상태 변수를 유지함으로써 여러 개의 광원을 동적으로 사용을 할 수가 있는데, 이 프로그램에서는 `ambientlight`와 `distantlight`라는 이름을 가지는 광원 쉐이더에 의해 정의되는 두 개의 광원을 사용한다.

이 광원 쉐이더들은 RenderMan에서 기본적으로 제공하는 표준 쉐이더들로서, 그 이름을 보면 이들 쉐이더의 목적은 분명하다. `ambientlight` 쉐이더는 다음과 같이 쉐이딩 언어를 사용하여 정의가 된다.

```
light ambientlight(float intensity = 1; color lightcolor =
1;) {
  Cl = intensity*lightcolor;
}
```

광원 쉐이더는 앞에서도 기술한 바와 같이 광원이 빛을 어떻게 비추는가에 관한 사항을 기술하는데 사용된다. 이 쉐이더는 입력 인자로 `intensity`와 `lightcolor`를 사용하고 있는데, 이들 인자는 `RiLightSource(*)`로 쉐이더를 설정할 때 원하는 값으로 설정을 할 수가 있고, 만약 이 함수에서 설정이 되지 않으면 쉐이더의 정의에서 기술된 디폴트 값을 사용한다. Line (e)를 보면 이 프로그램에서는 `"intensity"`와 `&intensity1`을 통하여 `intensity` 인자 값을 지정을 하고 있다. 재차 강조하면 광원 쉐이더의 근본 목적은 광원이 어떤 색깔의 빛을 어떤 방향으로 어떻게 비출 것인가를 설정하는 것인데, 이 쉐이더에서 계산하는 `Cl`은 쉐이더 변수(shader variable)로서, RenderMan에서는 쉐이더 변수라는 전역 변수를 통하여 렌더링 계산에 필요

한 인자 값을 저장한다. 여기서 Cl은 lightcolor과 마찬가지로 RGB 값을 가지는 color 타입의 값으로서 광원으로부터 물체 표면에 들어오는 빛의 색깔을 의미한다. ambientlight 쉐이더에서는 광원의 색깔 외에 다른 인자, 즉 광원의 위치나 방향에 대하여 어떠한 설정도 하지 않고 있는데, 이는 이 쉐이더가 방향성이 없는 광원, 다시 말해서 앰비언트 광원을 정의하고 있음을 의미한다.

다음 Line (f)에서 지정하는 distantlight 쉐이더는 평행 광원을 설정하기 위한 것인데 다음과 같이 정의된다.

```
light distantlight(float intensity = 1; color lightcolor =
1;
         point from = point "shader" (0,0,0);
         point to = point "shader" (0,0,1);) {
  solar(to-from, 0)
    Cl = intensity*lightcolor;
}
```

이 쉐이더의 두 인자 from과 to는 평행 광원의 방향을 결정하기 위한 것으로서, RiLightSource(*) 함수를 통하여 넘어온 인자 값을 사용하여 계산된 to-from 벡터가 빛이 들어오는 방향이 된다. 자세한 설명은 생략하겠지만, 여기서는 쉐이딩 언어에서 제공하는 solar 문장을 통하여 Cl의 색깔을 가지는 평행 광원을 정의하고 있다.

다음 Line (h), (k), (m), (o)에서는 RiSurface(*) 함수를 통하여 물체 표면에서 광원에서 들어오는 빛을 어떻게 반사시킬 것인가를 정의해주는 표면 쉐이더를 설정하고 있다. 우선 Line (h)에서는 그림 3.24의 바닥에 해당하는 직사각형에 대한 표면 쉐이더로서 matte를 지정하고 있다. 이 쉐이더도 RenderMan에서 기본으로 제공하는 표준 쉐이더 중의 하나인데, 다음과 같이 정의된다.

```
surface matte(float Ka = 1, Kd = 1;) {
  point Nf;
```

```
    Nf = faceforward(N, I);
    Oi = Os;
    Ci = Os*Cs*(Ka*ambient() + Kd*diffuse(Nf));
  }
```

이 함수의 내용을 보면 matte 쉐이더는 앰비언트 반사와 난반사만 하는 물질의 반사 성질을 정의해주는 쉐이더임을 쉽게 추측할 수가 있다. 여기서도 특별한 의미를 가지는 쉐이더 변수들이 사용이 된다. 표면 쉐이더의 궁극적인 목적은 광원에서 출발하여 물체 표면에서 반사가 되어 우리 눈으로 들어오는 빛의 색깔을 계산하는 것인데, 위의 쉐이더에서 계산하는 쉐이더 변수 Ci와 Oi가 그러한 빛의 색깔과 불투명도를 나타낸다. 이 쉐이더는 입력 인자로 들어오는 앰비언트 반사 계수 Ka와 난반사 계수 Kd, 그리고 각각 RiColor(*) 함수와 RiOpacity(*) 함수에 의해 설정되는 현재 색깔 Cs와 현재 불투명도 Os를 사용하여 반사 색깔을 계산한다. 여기서 사용되는 내장 함수 faceforward(N, I)를 보면, N은 쉐이딩에 사용이 되는 물체 표면에서의 법선 벡터이고, I는 눈에서 현재 쉐이딩을 하려하는 물체의 지점 P를 향한 벡터이다. faceforward(*) 함수의 목적은 N 벡터가 항상 눈을 향한 쪽이 되도록 해주는 것인데, 이 함수를 사용함으로써 양면 조명을 할 수가 있다. 또 다른 내장 함수 ambient()는 현재 광원 리스트에 있는 모든 앰비언트 광원의 색깔을 더해 리턴을 해주고, diffuse(*)는 인자로 들어온 법선 벡터를 사용하여 각 광원에 대하여 300쪽의 조명 공식 (3.1)에서의 $I_{l_i\lambda} \cdot (N \cdot L_i)$에 해당하는 값들을 모두 더해 리턴을 해주는데, 각 값에 물질 인자 Ka와 Kd을 곱해 원하는 반사 색깔을 구하게 된다.

다음 그림 3.24의 가장 왼쪽에 있는 구를 그리는데 사용하는 paintedplastic 쉐이더를 살펴보자(Line (k)).

```
surface paintedplastic(float Ka = 1, Kd = .5, Ks = .5,
        roughness = .1; color specularcolor = 1;
```

```
        string texturename = "";) {
  point Nf, V;
  color Ct;

  if (texturename != "") Ct = color texture(texturename);
  else Ct = 1;

  Nf = faceforward(normalize(N), I);
  V = -normalize(I);
  Oi = Os;
  Ci = Os*(Cs*Ct*(Ka*ambient() + Kd*diffuse(Nf)) +
        specularcolor*Ks*specular(Nf, V, roughness));
}
```

이 구는 Line (j)에서 설정된 바와 같이 0.3의 불투명도를 갖도록 "redchecker"라는 텍스춰를 사용하여 렌더링을 하였다. 여기서 사용된 내장 함수 specular(*)는 정반사 계산에 필요한 색깔 값을 계산해주는 내장 함수인데, 그 의미는 diffuse(*)를 생각해보면 쉽게 유추할 수가 있다. 그 내용을 보면 paintedplastic 쉐이더의 계산 과정을 어렵지 않게 이해할 수가 있는데, 쉐이딩 지점 P에 대응이 되는 텍스춰 좌표 s와 t를 이용하여 텍스춰 맵으로부터 텍스춰 색깔 Ct를 가져온 후, 이를 사용하여 물체 표면에서의 반사 색깔을 계산하고 있다.

다음 그림의 가운데와 오른쪽의 구를 렌더링하는데 사용하는 screen 쉐이더 (Line (m))와 granite 쉐이더(Line (o))는 RenderMan 쉐이더가 얼마나 유용한 툴인지를 보여주고 있다. 즉 RenderMan 시스템에서는 간단한 쉐이더 프로그래밍을 통하여 매우 다양한 렌더링 효과를 창출할 수가 있는데, OpenGL처럼 고정된 라이팅 계산을 사용하는 렌더링 시스템에서는 종종 그러한 효과를 내는 것이 상당히 복잡하거나 경우에 따라 불가능하고는 한다. 이 두 표면 쉐이더는 아래와 같이 정의된다.

```
surface screen(float Ka = 1, Kd = 0.75, Ks = 0.4,
```

```
            roughness = 0.1; color specularcolor = 1;
            float density = 0.25, frequency = 20;) {
  point Nf;

  if (mod(s*frequency, 1) < density ||
            mod(t*frequency, 1) < density) {
  Oi = 1;
  Nf = faceforward(normalize(N), I);
  Ci = Os*(Cs*(Ka*ambient() + Kd*diffuse(Nf)) +
       specularcolor*Ks*specular(Nf, -normalize(I),
roughness));
  }
  else {
    Oi = 0; Ci = 0;
  }
}


surface granite(float Kd = .8, Ka = .2, Ks = 1.5) {
  float sum = 0;
  float i, freq = 1.0;

  for (i = 0; i < 6; i = i + 1) {
    sum = sum + abs(.5 - noise(4*freq*I))/freq;
    freq *= 2;
  }
  Ci = Ks*Cs*sum*
       (Ka + Kd*diffuse(faceforward(normalize(N), I)));
}
```

screen 쉐이더에서는 쉐이더 변수 s와 t를 사용하여 반사되는 색깔 Ci와 불투명도 Oi를 계산하고 있다. s와 t는 현재 쉐이딩 계산을 하려고 하는 지점을 나타내는 쉐이더 변수 P에 매핑이 된 텍스춰 좌표를 나타낸다. 디폴트로 이 매핑은 일반적인 구와 정사각형에 대한 매핑을 사용하는데, 이 텍스춰 좌표와 모수 함수 mod(*)를 사용하여 쉐이딩하려는 지점의 위치가 골격에 해당하는 부분에 있으면 색깔을 계

산해주고, 아니면 완전히 투명하게 하여(Oi = 0; Ci = 0;), 마치 구에 뼈대만 있고 나머지 부분은 구멍이 뚫린 것처럼 보이게 해준다. 또한 granite 쉐이더는 쉐이더 변수 I와 내장 함수인 노이즈 함수 noise(*)를 사용하여 구의 표면에 화강암 효과를 내주고 있다. I는 카메라 시점에서 P를 향한 벡터를 의미하는데, 즉 현재 쉐이딩하려는 지점에 대한 방향에 대하여 랜덤 함수인 noise(*)를 사용하여 화강암 효과를 내고 있다. 한 가지 흥미로운 것은 가운데 구를 그릴 때 골격만 있는 구를 모델링하여 그에 대한 기하 물체를 그린 것이 아니라, 다른 구처럼 단순히 구를 그리면서, 단지 라이팅 계산을 할 때 적절한 처리를 통하여 마치 골격만 그린 것과 같은 효과를 내고 있다는 사실이다. 이 두 쉐이더에 대한 자세한 설명은 생략하려 하는데, 그 내용을 살펴보고 프로그램이 가능한 라이팅 계산이 고정된 라이팅 계산 구조에 비해 얼마나 강력한 기능을 제공할 수 있는지에 대한 느낌을 얻기 바란다. 참고로 영화 토이 스토리를 제작할 때 1300개 이상의 쉐이더를 작성하여 여러 가지 재미있는 효과를 냈다고 한다. 이러한 쉐이더 프로그래밍을 하는데 있어 프로그래머의 상당한 창조성을 필요로 하는데, 바로 이러한 프로그래밍이 컴퓨터 그래픽스 프로그래밍의 백미가 아닌가 한다.

## 6.7 렌더링 성능 제고를 위한 노력

이제 다시 OpenGL로 돌아와 라이팅 계산과 관련하여 몇 가지 프로그래밍 기법에 대하여 생각을 해보자. 일반적으로 실시간 렌더링 시스템에서는 라이팅 계산을 하기 위하여 그리고자 하는 기하 프리미티브들의 모든 꼭지점에 대하여 조명 모델 공식을 적용하므로, 장면이 약간만 복잡해도 상당히 많은 개수의 꼭지점에 대하여 방대한 양의 부동 소수점 연산을 수행하여야 한다. 따라서 경우에 따라 렌더링 파이프라인 전체에 있어 라이팅 계산이 큰 부담이 되기도 한다. OpenGL 시스템은 소

프트웨어만으로 구현을 하기도 하지만 렌더링 속도를 향상시키기 위하여 렌더링 파이프라인의 일부나 또는 전체를 하드웨어로 구현하여 가속을 하기도 한다. 저가의 그래픽스 하드웨어의 경우에는 점, 선분, 다각형 등의 기하 프리미티브들의 래스터화 과정과 같은 기본적인 연산은 하드웨어로 구현을 하고, 부동 소수점 연산을 사용하는 기하 변환, 조명 계산, 절단 등의 복잡한 렌더링 계산은 소프트웨어적으로 수행을 하고는 한다. 이러한 하드웨어를 사용하면 단순히 소프트웨어로 구현된 OpenGL 시스템을 사용할 때보다 단순한 기하 물체는 빠르게 렌더링이 되겠지만, 조명 계산이 필요할 경우 이 부분은 소프트웨어적으로 계산이 되므로 느려지게 된다. 따라서 저가의 OpenGL 가속기를 사용한다면 과다한 광원의 사용은 시스템 전체에 큰 부담이 될 수가 있다. 물론 고가의 그래픽스 하드웨어일수록 조명 계산을 비롯한 많은 부분이 하드웨어로 가속이 되므로 렌더링 계산이 빠르게 수행이 된다.

어떤 방식으로 구현된 OpenGL 시스템을 사용하느냐에 따라 그 효과가 달라질 수가 있겠지만, 프로그래머의 입장에서 보면 OpenGL에 기반을 둔 응용 소프트웨어를 개발할 때 렌더링 계산이 효율적으로 수행이 될 수 있도록 모든 노력을 경주해야 할 것이다. 조명 계산의 성능에 영향을 미칠 수 있는 몇 가지 사항에 대하여 살펴보자. 아래의 내용은 OpenGL의 라이팅 모델을 정확히 이해를 하였다면 쉽게 이해를 할 수 있을 것이다. 고가의 그래픽스 하드웨어의 경우에는 대부분의 렌더링 계산이 하드웨어로 수행이 되므로 덜 영향을 받겠지만, 저가의 가속기나 소프트웨어로 구현된 OpenGL을 사용한다면 상대적으로 많은 영향을 받게 될 것이다.

- 법선 벡터는 조명 계산에 있어 중요한 역할을 한다. 기하 물체의 각 꼭지점에 대하여 조명 공식을 적용할 때 법선 벡터의 길이는 1이어야 한다. OpenGL 시스템에는 임의의 길이를 가지는 법선 벡터를 단위 벡터로 변환하여 주는 기능이 있으나, 이는 적지 않은 부동 소수점 연산을 필요로 하므로 가능한 한 그러

한 기능의 사용은 피해야 한다. 따라서 미리 법선 벡터를 정규화하여 단위 벡터로 만들어 사용을 하고, 한편 사용하는 모델링 및 뷰잉 변환이 강체 변환이라면 변환 후에도 벡터의 길이가 보존이 되므로 그러한 기능을 사용하지 않는 것이 현명한 방법이라 하겠다. 반면 눈 좌표계까지의 기하 변환이 강체 변환이 아니면, 예를 들어, 크기 변환을 포함하면, 변환 후 벡터의 길이가 변하게 되므로 OpenGL 시스템에서 다시 길이를 1로 만들어 주도록 해야 한다. 만약 한 물체에 대한 모델링 변환이 벡터의 길이를 일정하게 변화시킨다면, 예를 들어, 항상 벡터의 길이를 두 배로 확장한다면, 물체의 모델링 단계에서 벡터의 길이를 모두 0.5가 되도록 하면, 이 모델에 대해서는 기하 변환 후 OpenGL 시스템에서 벡터의 길이를 정규화할 필요가 없다.

- 다음은 상대적으로 많은 계산량을 요구하기 때문에 가능한 한 피해야 한다: 점 광원, 스폿 광원, 양면 조명, 지역 관찰자, 과다한 광원의 설정. 대신에 평행 광원, 일면 조명, 무한 관찰자 등을 사용하도록 하는 것이 소프트웨어로 구현된 OpenGL이나 저가의 가속기를 사용할 때 성능 향상에 도움을 줄 것이다. 특히 평행 광원을 사용할 때 원근 투영을 하더라도 무한 관찰자를 사용하면 정반사 계산에 필요한 해프웨이 벡터가 상수 벡터가 되므로 장면 전체에 대하여 한 번만 계산을 하면 된다. 이렇게 하면 매 꼭지점마다 해당 벡터를 계산하여 길이를 정규화해주어야 하는 부담을 많이 줄일 수 있을 것이다. 또한 기하 물체를 그릴 때 문제가 없다면 뒷면 제거를 한다. 이렇게 함으로써 래스터화 단계에서의 부담을 줄일 수가 있다.

- OpenGL 시스템을 소프트웨어로 구현을 하건 하드웨어로 구현을 하건, 주어진 환경에서 렌더링 계산을 가장 효율적으로 할 수 있도록 개발을 할 것이다.

특히 렌더링 시스템을 개발할 때 상대적으로 비용이 많이 드는 부동 소수점 연산을 가능한 한 줄이도록 노력을 할 것이다. 조명 계산도 부동 소수점 연산이 많이 필요한 부분이므로, 이러한 과정을 어떻게 구현하였을까를 생각해 보는 것은 그러한 시스템을 효율적으로 사용하는데 도움이 될 것이다. 특히 많은 부분이 소프트웨어로 구현이 되어 있는 경우에는 더욱 그러할 것이다. 350쪽의 OpenGL 조명 공식을 다시 한번 살펴보자. 프로그래머가 조명 인자를 한 번 설정을 하면, 그러한 인자들은 물체를 구성하는 각 꼭지점에 대하여 조명 공식을 적용하는데 있어 반복적으로 사용될 가능성이 많다. 따라서 조명 인자들이 설정이 되면 $\mathbf{a}_{cm} * \mathbf{a}_{cli}$, $\mathbf{d}_{cm} * \mathbf{d}_{cli}$, $\mathbf{s}_{cm} * \mathbf{s}_{cli}$ 등을 미리 곱해 저장을 하고, $\mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs}$ 값을 계산해 놓으면, 각 꼭지점에 대하여 조명 계산을 할 때 부동 소수점 연산의 비용을 줄일 수 있다. 또한 정반사 지수 $s_{rm}$을 사용하여 거듭 제곱을 계산하여야 하는데, 이 또한 계산량이 적지 않은 부분이라 할 수 있다. 이러한 거듭 제곱 계산을 효율적으로 근사화할 수 있는 방법으로 아래와 같이 전처리를 통한 테이블 사용 기법을 들 수가 있다.

```
void compute_table(GLfloat *table, GLfloat shininess) {
 int i;

 table[0] = 0.0;
 if (shininess == 0)
  for (i = 1; i <= TABLE_SIZE; i++) table[i] = 1.0;
 else {
  for (i = 1; i <= TABLE_SIZE; i++) {
   double tmp = pow(i/(GLfloat) TABLE_SIZE,
shininess);
    table[i] = 0.0;
    if (tmp > 1e-10) table[i] = tmp;
  }
 }
```

```
}
```

정반사 계산을 할 때에는 항상 0과 1 사이의 값에 대하여 거듭 제곱을 계산하므로 0부터 1까지의 구간을 적절한 개수(TABBLE_SIZE)로 나눈 후, 새롭게 설정하려하는 정반사 지수(shininess)를 사용하여 각 구간의 끝점에서의 거듭 제곱 값을 미리 계산하여 테이블에 저장을 하고, 실제로 각 꼭지점에 대한 조명 계산을 수행할 때에는 가장 가까운 값에 대한 테이블 값을 사용한다면 많은 양의 부동 소수점 연산을 줄일 수 있을 것이다. 물론 이러한 근사 방법을 사용하면 오차가 발생하겠지만 테이블의 크기를 충분히 크게 설정하면 별 문제가 발생하지 않을 것이다. 이러한 전처리를 통한 테이블 사용 기법은 실시간 그래픽스와 같이 계산 시간이 매우 중요한 실시간 처리 분야에서 많이 쓰이는 방법이다. 만약 사용 OpenGL 시스템이 이러한 식으로 구현이 되어 있다면 물질의 정반사 지수는 가급적 자주 바꾸지 않는 것이 좋을 것이다.

- 조명 계산에 있어 광원을 많이 사용할 수록 계산량이 증가하리라는 것은 명확하다. 따라서 꼭 필요한 광원이 아니라면 사용하지 않는 것이 좋다. 특히 어떤 OpenGL 구현에서는 한 개의 광원만 사용할 경우에 대하여 조명 계산을 최적화하기도 하기 때문에, 이러한 시스템에서는 두 개 이상의 광원을 사용할 경우 한 개의 광원을 사용할 때 보다 많이 느려지게 된다.

- 플랫 쉐이딩과 스므드 쉐이딩을 비교하여 보면, 플랫 쉐이딩을 사용할 때에는 한 개의 꼭지점에 대해서만 조명 계산을 하면 되고, 스므드 쉐이딩의 경우에는 모든 꼭지점에 대하여 조명 계산을 하여야 한다. 따라서 일반적으로 말하면 플랫 쉐이딩을 할 때가 수행되는 조명 계산의 양이 훨씬 적다고 할 수 있을 것이다. 그러나 이는 상황에 따라서, 예를 들어, 삼각형 스트립처럼 다각형들

이 꼭지점들을 공유할 때, 반드시 훨씬 빠르다고만 할 수는 없을 것이다.

다음 조명 계산에 직접적으로 영향을 미치지는 않지만 렌더링 계산을 효율적으로 수행하는데 도움이 되는 몇 가지 사항에 대하여 살펴보자.

- 당연한 이야기이겠지만 불필요한 OpenGL 함수의 호출을 피하도록 한다. 예를 들어 조명 계산을 안 한다면 굳이 glNormal*(*) 함수를 사용하여 각 꼭지점마다 법선 벡터를 기술할 필요가 없다.

- 꼭지점의 성질을 기술하기 위하여 glVertex*(*), glNormal*(*), glColor*(*), glTexCoord*(*) 함수 등을 사용할 때 포인터 버전, 즉 함수 이름에 v가 붙은 것을 사용하는 것이 더 효율적일 가능성이 많다. 이러한 함수들을 사용함으로써 함수 호출에 있어 인자의 개수를 줄일 수 있고, 연속된 영역의 메모리를 액세스할 수 있게 된다. 이러한 기계 레벨의 최적화 기법은 사용하는 하드웨어나 운영 체제, 그리고 컴파일러 시스템 등에 따라 그 효과가 달라지나, 일반적으로 말해서 연달아 액세스하는 데이터는 메모리의 연속된 영역에 저장이 되도록 하는 것이 좋다. 이렇게 함으로써 효율적으로 캐쉬 메모리를 사용할 수가 있고 또한 시스템 버스를 통하여 움직이는 데이터의 양을 줄일 수 있을 것이다. 또한 과다한 전역 변수의 사용은 시스템이나 컴파일러에 따라 지역 변수만을 사용할 때보다 비효율적인 실행 코드를 산출할 가능성이 있으므로, 불필요한 전역 변수의 사용은 가급적 피하는 것이 좋다. 또한 과다한 포인터의 사용이나 함수 호출, goto 문장, 조건문 등은 캐쉬 메모리를 효율적으로 사용하는데 방해가 될 가능성이 많다. 컴퓨터 아키텍춰나 컴파일러 시스템 등에 지식이 있는 프로그래머라면 자신이 작성한 프로그램이 수행될 때의 캐쉬 메모리의 히트 비율(hit ratio)에 대하여 생각해 보는 것도 도움이 될 수 있

다. 최근의 컴파일러들은 실행 코드를 생성할 때 다양한 최적화 기법을 적용하여 상당히 효율적인 코드를 생성하기 때문에 반드시 이러한 프로그래머의 노력이 성능 향상에 큰 효과가 있으리라는 보장은 없다. 그러나 실시간 프로그래밍을 할 때, 사용 시스템의 프로세서 및 컴파일러의 매뉴얼을 통하여 작동 원리를 제대로 이해를 하고, 그러한 이해에 기반을 두고 프로그래밍을 한다면 한 단계 수준 높은 실시간 소프트웨어를 제작할 수 있을 것이다.

- 또한 컴파일러 개발에 있어 사용되는 여러 최적화 기법을 응용 프로그램 작성 시 직접 적용해보는 것도 좋은 생각이라 할 수 있다. 조명 계산을 비롯하여 여러 부류의 렌더링 계산을 할 때, 같은 형태의 계산이 방대한 수의 꼭지점에 대하여 반복적으로 수행이 되는 경우가 많다. 이러한 경우 컴파일러가 여러 번 반복되는 회전 루프에 대하여 최적화 노력을 하듯이, 루프에 대하여 정적인 데이터는 미리 한 번만 계산을 한다거나 약간의 프로그램 구조를 바꾸어 불필요한 계산을 줄인다거나하는 등의 방법을 적용해볼 수가 있을 것이다. 예를 들어 다음과 같은 코드를 생각해보자.

```
for (k = 0; k < n_vecs; k++) {
  for (i = 0; i < 3; i++) {
    tmp = 0;
    for (j = 0; j < 3; j++) {
      tmp += M[i][j]*ptr_vec_in[j];
    }
    ptr_vec_out++ = tmp;
  }
  ptr_vec_in += 3;
}
```

지금 n_vec개의 3차원 벡터가 포인터 변수 ptr_vec_in가 가리키는 지점에서 시작하여 연속하여 저장이 되어 있는데, 각 벡터를 3행 3열 행렬 M에 곱해 그

결과를 역시 포인터 변수 ptr_vec_out이 가리키는 장소에 계속해서 저장을 하려하고 있다. 이 코드의 내용은 쉽게 이해를 할 수 있는데, 아주 간결하게 잘 작성한 코드처럼 보이지만 여기에는 한 가지 고려를 해야할 점이 있다. 이 코드를 컴파일한 후 실행을 하면, 매번 회전을 할 때 안 쪽에 포함된 두 개의 회전문에 대한 인덱스 변수 i와 j를 항상 새롭게 고쳐주어야 하고, 또한 그 값들이 3보다 작은지에 대한 비교 연산을 해주어야 한다. 이는 별로 많은 연산량을 필요로 하는 것 같지는 않지만 매번 회전문 안에서 행렬과 벡터를 곱하기 위해 수행되는 계산량에 비해서 결코 무시할 수 없는 계산이라 할 수 있다. 또한 프로그램이 수행이 될 때는 이 코드처럼 이리저리 옮겨가면서 명령어를 수행하는 것보다는 순차적으로 저장된 명령어를 수행시키는 것이 더 효율적임은 널리 알려져 있다. 컴파일러의 최적화 방법 중에 회전문을 풀어버리는 기법(loop unrolling)이 있는데, 이는 간단한 회전문은 그냥 풀어버리는 방법으로서, 위의 코드는 약간 원시적으로 보이기는 하지만 아래와 같이 풀 수가 있다.

```
for (k = 0; k < n_vec; k++) {
  tmp1 = M[0][0]*prt_vec_in[0];
  tmp2 = M[1][0]*prt_vec_in[0];
  tmp3 = M[2][0]*prt_vec_in[0];
  tmp1 += M[0][1]*prt_vec_in[1];
  tmp2 += M[1][1]*prt_vec_in[1];
  tmp3 += M[2][1]*prt_vec_in[1];
  tmp1 += M[0][2]*prt_vec_in[2];
  tmp2 += M[1][2]*prt_vec_in[2];
  tmp3 += M[2][2]*prt_vec_in[2];
  ptr_vec_out++ = tmp1;
  ptr_vec_out++ = tmp2;
  ptr_vec_out++ = tmp3;
  ptr_vec_in += 3;
```

```
}
```

실제로 사용하는 컴파일러에 따라 그 효과가 다르겠지만, 이러한 노력은 종종 상당한 효과를 거두고는 한다[24].

다시 한번 강조하면 이러한 부류의 최적화 노력은 이미 컴파일러에 의하여 대부분 수행이 되나, 계산 시간이 OpenGL 렌더링을 하는데 있어 가장 중요한 자원중의 하나라는 점을 고려하면 프로그래머의 입장에서는 모든 가능한 노력을 경주하여야 할 것이다. 한 가지 더 예를 들어보자. glBegin(*)과 glEnd() 함수는 그 안에서 기하 물체에 대한 정보가 기술되었을 때, 그러한 정보들을 모아서 렌더링 파이프라인으로 보내는 역할을 한다. 따라서 렌더링 시스템 입장에서 보았을 때, 이 두 함수의 내부에서는 가능한 한 효율적으로 프로그래밍이 되어야만 기하 물체에 대한 데이터를 빠르게 파이프라인으로 보낼 수가 있을 것이다. 다음과 같은 코드를 생각해보자.

```
glBegin(GL_QUAD_STRIP);
  for (i = 0; i < nvertices; i++) {
    if (do_lighting) glNormal3fv(obj->normal[i]);
    glVertex3fv(obj->vertex[i]);
}
glEnd();
```

여기서는 조명 계산을 할 필요가 있을 때에만(`do_lighting == TRUE`) 법선 벡터를 기술하기 위하여 `if` 문장을 사용하였다. 이 코드를 아래의 코드와 비교하여 보자.

```
if (do_lighting) {
  glBegin(GL_QUAD_STRIP);
    for (i = 0; i < nvertices; i++) {
```

[24] C. Hecker. "PowerPC Compilers: Still Not So Hot," *Game Developer Magazine*, pp. 12-23, June/July 1996.

```
      glNormal3fv(obj->normal[i]);
      glVertex3fv(obj->vertex[i]);
    }
  glEnd();
}
else {
  glBegin(GL_QUAD_STRIP);
    for (i = 0; i < nvertices; i++)
glVertex3fv(obj->vertex[i]);
  glEnd();
}
```

역시 이 예에서도 앞의 코드가 간결하게 작성한 좋은 코드같이 보이나, 그래픽스 시스템을 구동하는 입장에서 보면 후자가 더 좋은 코드라 할 수 있다. 즉 전자의 경우 매번 `for` 루프 안의 코드가 수행이 될 때 각 꼭지점에 대하여 비교 연산을 하여야 하지만, 후자의 경우에는 `if` 문장을 바깥으로 뽑아 냄으로써 한 번의 비교 연산만 하면 되는 것이다. 여러 가지로 볼 때 후자의 코드가 훨씬 더 좋은 코드라 하겠다.

- 가능하다면 기하 물체를 기술할 때 GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUAD_STRIP과 같이 꼭지점을 공유하는 기하 프리미티브를 사용하는 것이 좋다. 이럴 경우 GL_LINES, GL_TRIANGLES, GL_QUADS, GL_POLYGON 등을 사용하여 물체를 구성하는 프리미티브들을 개별적으로 기술하는 것 보다 훨씬 적은 개수의 꼭지점을 사용하여 물체를 표현할 수 있다. 꼭지점의 개수가 적다는 것은 그 만큼 메모리를 적게 필요로 하고, 또한 기하 변환이나 조명 계산의 회수가 줄어들게 됨을 의미하게 된다. 또한 정확도에 문제가 없다면 굳이 GLfloat 타입 대신에 GLdouble 타입의 데이터를 사용할 이유가 없다.

- 반드시 스므드 쉐이딩을 사용해야하는 상황이 아니라면 플랫 쉐이딩을 사용하도록 한다. 래스터화 과정에서 스므드 쉐이딩의 경우에는 다각형의 꼭지점에 연관된 색깔들에 대하여 선형 보간법을 사용하여 다각형 내부에 해당하는 각 화소의 색깔을 계산을 하여야 한다. 플랫 쉐이딩을 하면 하나의 색깔을 모든 화소에 칠하게 되므로 계산량이 줄어 들 수가 있다. 물론 앞에서 언급한 바와 같이 이는 사용하는 그래픽스 하드웨어의 성능에 따라 달라 질 수가 있다. 즉 래스터 전 과정이 하드웨어로 구현이 되어 있다면 굳이 플랫 쉐이딩을 선호할 이유가 없다.

- 상황이 허락하면 정적인 기하 물체에 대해서는 OpenGL에서 제공하는 디스플레이 리스트(display list) 기능을 사용하도록 한다. 프로그래머가 OpenGL 함수의 호출을 통하여 기하 물체를 기술하면, 이는 OpenGL 렌더링 파이프라인이 이해할 수 있는 형태로 변환이 되어 파이프라인으로 흘러가게 된다. 따라서 매번 기하 물체를 그릴 때 그러한 변환 과정이 필요하다. 디스플레이 리스트 기능을 사용하면, 동일한 기하 물체를 여러 번 반복해서 렌더링을 해야 할 때, 첫 번째 그릴 때의 변환 계산 결과를 저장을 해놓고, 다음 번부터는 그러한 결과를 사용하여 렌더링 파이프라인을 구동하게 되므로 효율적인 렌더링을 할 수가 있다.

# 제 4 장

# 래스터화: 벡터 데이터에서 래스터 데이터로

## 제 1 절 래스터화 과정에 대한 고찰

### 1.1 조명 계산 이후의 렌더링 계산

지금까지 2장과 3장에서는 렌더링 파이프라인에서 기하 변환과 조명 계산에 관련된 부분에 대하여 자세하게 살펴보았다. 다음의 계산 과정으로 넘어가기 전에 107쪽의 그림 2.15와 266쪽의 그림 3.2, 그리고 그림 4.1을 통하여 OpenGL 렌더링 파이프라인에서의 현재의 위치를 조관해보자. 첫 번째 그림(그림 2.15)은 프로그래머가 기하 물체에 대한 속성을 기술해주는 물체 좌표계에서, 화면에 해당하는 윈도우 좌표계까지의 기하 변환에 관한 것이다. 반면 두 번째 그림(그림 3.2)은 눈 좌표계에서의 조명 계산과 프리미티브 조합 과정에 관한 것이며, 마지막으로 세 번째 그림(그림 4.1)은 그 이후의 OpenGL 렌더링 계산 과정을 보여주고 있다.

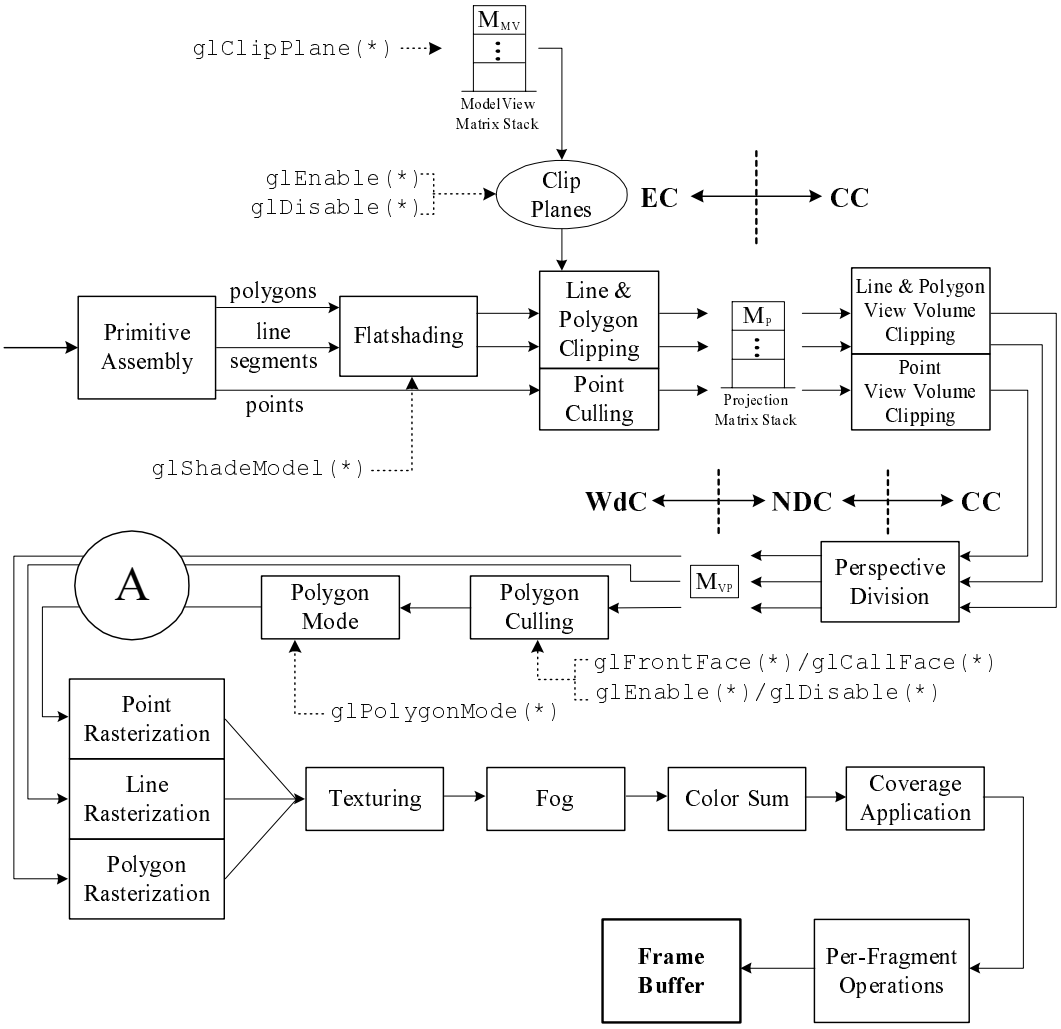



그림 4.1: 조명 계산 이후의 렌더링 계산

프리미티브의 조합 이전에 렌더링 파이프라인을 통하여 흘러가는 데이터의 기본 단위는 꼭지점이라고 설명을 하였다. 꼭지점의 좌표와 그에 연관된 데이터들이 적절한 처리를 통하여 프리미티브 조합 모듈에서 프리미티브 별로 조합이 되어, 그 순간부터 점, 선분, 다각형 등 기하 프리미티브들이 데이터의 기본 단위가 되어 파이프라인을 흘러가게 된다. 물론 앞에서 설명한 바와 같이 각 기하 프리미티브는 개념적으로 그것의 타입과 프리미티브들을 구성하는 꼭지점 좌표 및 그에 연관된 데이터(색깔, 텍스춰 좌표, 에지 플래그 등)로 구성이 된다.

그림 4.1에 주어진 것과 같이 일단 프리미티브 형태로 조합이 되면, 플랫 쉐이딩을 하는지의 여부에 따라 선분과 다각형에 대하여 적절한 처리를 한 후[1], 사용자 설정 절단 평면에 대한 절단 연산을 수행한다. 여기까지가 눈 좌표계에서의 계산인데, 이후 투영 행렬 $M_P$에 의하여 절단 좌표계로 변환이 된다. 이 좌표계에서는 뷰잉 볼륨에 대하여 절단을 하여 렌더링 결과에 영향을 미치지 않는 기하 프리미티브들을 제거한 후, 원근 나눗셈을 통하여 정규 디바이스 좌표계로 넘어 간다. 다음 마지막 기하 변환인 뷰폿 변환 행렬 $M_{VP}$에 의하여 OpenGL 파이프라인에서의 마지막 좌표계인 윈도우 좌표계에 이르게 된다. 이 좌표계에서는 다각형에 대하여 뒷면 제거를 한 후 다각형을 어떻게 그릴 것인가를 결정하는 다각형 모드에 따라 적절한 처리를 하게 된다. 바로 이 지점(그림 4.1에서 A로 표시된 곳)까지가 우리가 지금까지 자세하게 살펴 본 OpenGL에서의 렌더링 과정에 해당한다.

이 지점 이후의 OpenGL 렌더링 파이프라인을 살펴보면 각 타입의 기하 프리미티브들에 대하여 래스터화(rasterization) 계산이 수행이 됨을 알 수가 있다(래스터화(Rasterization) 모듈). 스캔 변환(scan conversion)이라고도 하는 이 과정의 주목적은 각 기하 프리미티브가 투영이 되는 화면상의 화소들을 찾아 적절한 색깔로 칠

[1] 꼭지점에 대해서는 스므드 쉐이딩과 플랫 쉐이딩에 대한 차이가 없다.

해주는 것이다. 바로 이러한 래스터화 과정이 본 4장의 주제인데, 이 지점에서 렌더링 파이프라인을 흘러가는 데이터의 형태에 큰 변화가 일어난다. 즉 래스터화 과정 이전에는 연속 공간인 3차원 좌표계를 기준으로 하여 기하 및 그에 연관된 데이터가 '연속적인 형태'로 표현이 되었는데, 이 계산 과정을 통하여 이산 공간인 래스터 공간에서의 '이산적인 형태'의 데이터로 변환이 된다. 이제 래스터화 과정과 그에 따른 영향에 관해서 하나씩 살펴보려 하는데, 이 과정을 통하여 지금까지와는 구조적으로 전혀 다른 그래픽스 데이터가 생성이 됨을 알 수가 있을 것이다.

이후에는 텍스춰(Texturing) 모듈에서부터 시작하여 네 개의 그래픽스 연산이 수행이 된 후, 마지막으로 프래그먼트별 연산(Per-Fragment Operations) 모듈에서 실제로 프레임 버퍼에 저장이 될 데이터를 결정하는 계산이 수행이 된다. 렌더링의 근본 목표는 이미지를 생성하는 것이고, 따라서 프레임 버퍼, 특히 색깔 버퍼에 원하는 이미지 데이터가 저장이 되도록 하는 것이므로, 이러한 과정을 끝으로 렌더링 계산이 종료하게 된다.

## 1.2 연속 공간과 이산 공간, 그리고 앨리어싱

그림 4.1의 A 지점에 다다른 기하 프리미티브의 각 꼭지점 좌표의 의미에 대하여 다시 한번 생각을 해보자. 윈도우 상에서의 좌표 $(x_{wd}, y_{wd})$와 그 꼭지점이 화면 안쪽으로 얼마나 깊게 들어가 있는가에 대한 깊이 정보인 $z_{wd}$값이 해당 꼭지점에 대한 3차원적인 기하 정보를 제공하고 있다. 이 값들은 OpenGL에서의 화면에 해당하는 윈도우 좌표계 공간에서의 편리한 계산을 위하여 이 좌표계를 기준으로 표현이 되어 있다. 따라서 물체 좌표계에서의 원래의 좌표로부터 전혀 다른 값으로 변환이 되었겠지만, 아직도 하나의 3차원 공간에서 연속적인 성질을 가지는 데이터로 표현이 되어 있다. 즉 우리가 기하 물체에 대한 정보를 기술할 때 3차원 연속 공

간(3D continuous space)인 물체 좌표계에서 벡터 데이터 형태로 표현을 하였는데, 윈도우 좌표계에서도 아직 그러한 성질이 유지가 되고 있는 것이다[2].

렌더링 계산의 결과 최종적으로 얻고자 하는 것은 벡터 형태의 이미지가 아닌 래스터 이미지, 즉 균일한 격자로 구성되는 래스터 공간(raster space)이라 부르는 하나의 이산 공간(discrete space)에서 정의되는 2차원 이미지이다. 따라서 연속 공간에서 벡터 데이터 형태로 출발한 기하 프리미티브들이 어느 시점에선가 이산 공간의 래스터 이미지 형태로 변환이 되어야 하는데, 바로 이 과정이 래스터화인 것이다. 사실 래스터화 과정은 우리가 사용하는 래스터 그래픽스 시스템에서 없어서는 안 될 가장 중요한 계산 과정이면서, 반면에 래스터 그래픽스에서 가장 골칫거리 중의 하나인 앨리어싱(aliasing)의 원인이 되는 요인이기도 하다. 래스터화 과정에서 그래픽스 프리미티브에 대하여 발생하는 가장 큰 문제가 무엇인가 생각을 해보면, 연속 공간의 정보가 이보다 훨씬 불충분한 표현 능력을 가지는 이산 공간의 정보로 변환이 되면서 오차가 발생하고, 그 결과 많은 정보가 유실이 되는 것이라 할 수 있다.

컴퓨터가 연속 공간인 실수 집합의 숫자 중 간단하게 보이는 0.1이라는 숫자조차도 정확하게 표현하지 못하는 것은, 유한 개의 비트를 사용하는 컴퓨터가 기본적으로 유한 개의 숫자만 표현이 가능한 이산 공간에서 정의되는 불충분한 숫자 체계를 사용하기 때문이다. 그러한 결과 컴퓨터를 사용하여 실수 또는 복소수와 같은 형태의 데이터를 표현할 때 수치 오차(numerical error)가 발생하고, 이러한 오차들은 부동 소수점 연산이 계속 됨에 따라 지속적으로 누적이 된다. 다시 말해서 컴퓨터는 연속 공간의 수인 실수를 정확하게 표현을 할 수도 없거니와, 가감승제와 같

[2]물론 컴퓨터에서는 연속 공간을 구성하는 실수의 표현을 위하여 유한 개의 비트를 사용하는 부동 소수점 숫자를 사용하므로, 엄밀히 말하면 물체 좌표계는 연속 공간은 아니다. 하지만 여기서는 물체 좌표계가 개념적으로 연속 공간이라고 가정을 하자.

은 기본적인 실수 연산도 정확하게 수행할 능력이 없다. 따라서 실수 또는 복소수를 주로 사용하는 계산에서 부동 소수점에 대한 처리를 제대로 하지 못하면, 종종 수치 오차로 인하여 계산의 정확도에 심각한 문제가 발생한다. 실제로 컴퓨터는 간단하게 보이는 이차 방정식도 정확하게 풀 수 없는 경우가 있다[3]. 컴퓨터학에서 수치 해석(numerical analysis)이라는 분야에서는 바로 이렇게 불완전한 컴퓨터를 사용하여 실수와 복소수와 같은 연속 공간의 데이터를 처리해야만 하는 자연과학 및 공학 분야에서의 제반 문제를 어떻게 하면 빠르고 정확하게 해결을 할 것인가에 관한 문제를 다룬다.

다시 이 책의 주제인 그래픽스 렌더링으로 돌아와서 보면, 래스터화 과정에서도 동일한 문제가 발생함을 쉽게 알 수 있다. 즉 무한한 표현 능력을 제공하는 연속 공간에서의 기하 정보를 불충분한 표현 도구라 할 수 있는 이산 공간에서의 래스터 이미지라는 격자 틀을 이용하여 표현을 하려고 하기 때문에 표현 오차가 발생하는데, 그러한 것이 렌더링 문제에서는 시각적인 형태의 오차로 나타난다. 이렇게 컴퓨터 그래픽스 분야에서 앨리어싱의 결과 발생하는 오차를 앨리어스(alias)라 한다. 예를 들어 다각형의 래스터화 과정을 단순한 관점에서 생각해보면, 연속 공간에 존재하는 다각형에 대하여 래스터 이미지 격자의 기본 단위인 화소라는 조그만 창문의 중심에 대하여 샘플링을 하는 과정이라 할 수 있다(그림 4.2의 왼쪽). 다시 말해서 각 화소의 중심을 지나도록 직선을 뻗어(즉 화소의 중심에서 샘플링을 하여) 다각형과 만나면 다각형의 색깔로 그 화소를 칠하고 아니면, 배경 색깔로 칠하는 과

[3] 참고로 이차방정식 $f(x) = x^2 - 100000.000001x + 1 = 0$에 대하여 근의 방정식을 사용하여 근을 구하는 간단한 프로그램을 작성하여 보기 바란다. 정확한 두 근은 $10^5$과 $10^{-5}$인데, 고급 SGI 워크스테이션에서 사용되는 MIPS R12000 CPU상에서 `double` 타입의 부동 소수점 숫자를 사용하여 계산한 결과 $10^5$과 $0.9999996 \cdot 10^{-5}$이 구해진다. 두 번째 근에 오차가 개입이 되었음을 알 수가 있는데, `double` 타입이 십진수 15-16 자리 정도의 정확도를 제공한다는 사실을 고려하면 이는 결코 적은 양이 아니며, 응용 문제에 따라 심각한 결과를 초래할 수가 있다. 실제로 이렇게 수치적으로 불안정한(numerically unstable) 경우는 여러 부류의 문제에서 쉽게 발견할 수 있다.

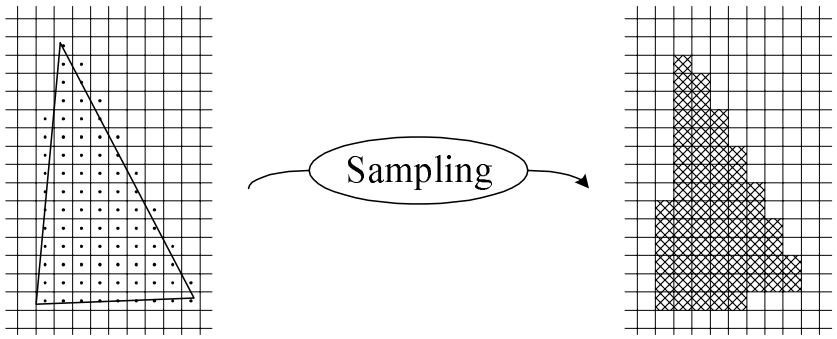



그림 4.2: 샘플링과 앨리어스

정이라 할 수 있다. 이렇게 래스터화가 되면 그 이후에는 기하 프리미티브에 대하여 연속 공간에서 표현된 정보는 모두 없어지고, 그림 4.2의 오른쪽과 같은 불충분한 형태의 이산적인 정보만 남게 된다. 물론 이 그림은 약간 과장이 되기는 하였으나 아무리 이미지의 해상도를 높여도 이산 공간으로의 샘플링이라는 과정을 통하여 많은 정보가 손실이 되고, 따라서 종종 시각적인 오차를 포함하는, 눈에 거슬리는, 이미지가 생성이 된다.

그림 4.3은 선분을 조금씩 회전시켜 가면서 렌더링을 한 모습을 보여주고 있는데, 선분이 거의 수평으로 눕거나 거의 수직으로 섰을 때 직선이 심하게 찢어져 보이는 것과 같은 모습을 쉽게 관찰할 수 있을 것이다. 이렇게 선분이 계단이 져서 보이는 현상을 톱니 모양의 선분(jagged edge)이라 하는데, 컴퓨터 그래픽스에서의 대표적인 앨리어스 중의 하나이다. 이러한 문제를 극복하기 위하여 여러 가지 방식의 시도를 해왔는데, 그러한 기법들을 앤티앨리어싱 기법(anti-aliasing technique)이라 부른다.

여기서 이러한 문제를 길게 설명하는 이유는 래스터화 과정에서 발생하는 골치 아픈 문제들에 대하여 정확한 이해가 필요하기 때문이다. 불행하게도 컴퓨터 그래픽스 분야에서 사용하는 대부분의 앤티앨리어싱 기법은 문제를 근본적으로 해결하지를 못한다. 더욱이 이러한 기법들은 추가적인 계산 시간을 요구하기 때문에 실시

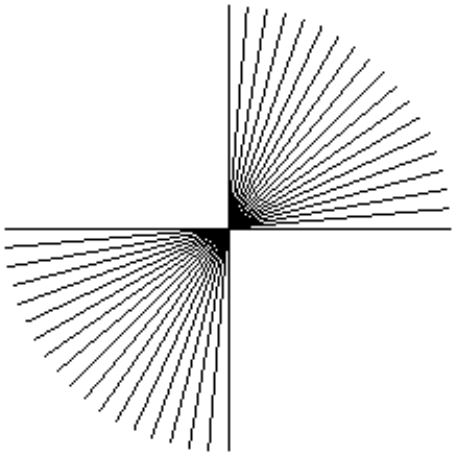


그림 4.3: 톱니 모양의 선분

간 그래픽스 프로그래밍에 대하여 더더욱 적용하기가 쉽지가 않다. 하지만 렌더링 소프트웨어를 제작하는 프로그래머 입장에서는 최소한 여기서의 문제와 그에 대한 대응책을 정확하게 파악을 해서, 경우에 따라 조금이라도 문제의 심각성을 축소시킬 수 있도록 여러 가지 노력을 기울여야 할 것이다.

## 제 2 절 선형 보간법과 점진적 계산

래스터화 과정에 대하여 본격적으로 살펴보기 전에 이 과정에서 핵심적인 역할을 하는 두 가지 부류의 계산 방법에 대하여 알아보자. 하나는 복잡한 함수의 효과적인 근사화를 위한 선형 보간법이라는 기법이고, 다른 하나는 효율적인 계산을 위한 점진적 계산이라는 기법이다.

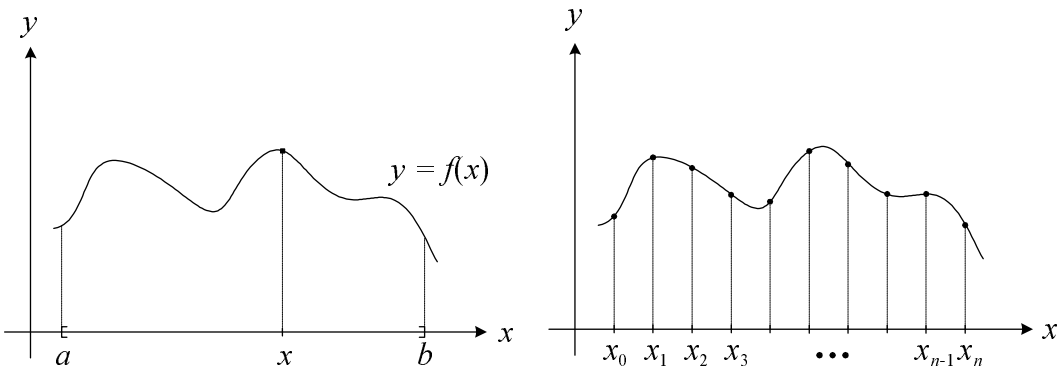
## 2.1 선형 보간법

### 2.1.1 보간법의 소개

보간법(interpolation)은 수치 해석 분야에서 가장 기본이 되는 방법 중의 하나로서, 수학적으로 복잡하거나, 또는 수학적으로 나타내는 것이 불가능한 함수를 효과적으로 표현하기 위한 수단으로 널리 쓰이고 있다. 이 방법은 실시간 계산만을 위하여 고안된 방법은 아니고, 오랜 기간 동안 복잡한 함수를 효과적으로 근사하기 위하여 사용되어 온 방법인데, 여기서는 이 책의 중요한 주제인 실시간 계산의 문맥에서 보간법에 대하여 간략하게 알아보도록 하자.

자연과학, 공학, 경영학 등 여러 분야에서 주어진 자연 현상을 수학적인 도구를 사용하여 모델링을 해야 하는 경우가 많이 있다. 예를 들어 렌더링 문제의 문맥에서 이를 살펴보면, 렌더링 계산을 통하여 영상을 생성한다는 것은 카메라의 필름에 해당하는 이미지 윈도우를 통하여 보이는 내용을 래스터 이미지 형태로 계산을 한다는 것을 의미한다. 이 경우는 3차원 공간에서 정의되는 2차원 평면 상의 사각형 영역을 향하여 물리학적 대상인 빛이 들어오는 것에 해당하는데, 임의의 지점 $(x, y)$를 향해 들어오는 빛의 색깔을 $C = f(x, y)$와 같이 수학적인 함수로 정확하게 표현을 할 수 있다면 이는 렌더링이라는 문제를 가장 완벽하게 풀었다고 할 수 있을 것이다. 그러나 대부분의 경우 이러한 해결이 불가능하기 때문에, 다른 방법을 통하여 근사적인 해법을 찾아야 한다.

근사적인 해결 방법으로 널리 쓰이는 것 중의 하나는 샘플링 과정을 통하여 주어진 현상으로부터 샘플 데이터를 수집하고, 이를 통하여 그러한 현상을 근사적으로 모델링을 하는 것이다. 그림 4.4(a)에서 도시된 바와 같이 가장 단순한 형태인, 1차원 구간에서 정의되는 임의의 함수 $y = f(x)$에 대하여 생각해보자. 이러한 함수의




(a) 미지의 함수

(b) 샘플링

| $x$ | $x_0$ | $x_1$ | $x_2$ | $\cdots$ | $x_{n-1}$ | $x_n$ |
|---|---|---|---|---|---|---|
| $y$ | $y_0$ | $y_1$ | $y_2$ | $\cdots$ | $y_{n-1}$ | $y_n$ |

(c) 샘플 데이터

그림 4.4: 함수의 샘플링

함수 값은 응용 문제에 따라 다양하게 나타날 수가 있는데, 렌더링의 경우에는 깊이 정보, 색깔, 텍스춰 좌표 등의 데이터가 사용될 수가 있다.

지금 우리가 관심이 있는 구간 $[a,b]$의 임의의 $x$에 대하여 미지의 함수 값 $f(x)$를 구하려 한다. 이 때의 함수를 효율적으로 근사화 하기 위하여 우선 구간 $[a,b]$를 $n$개의 구간으로 나누어, 각 구간의 끝 점 $x_i (i = 0, 1, \cdots, n)$에서 함수 값 $y_i = f(x_i)$을 실험이나 전처리를 통하여 얻어낸다(그림 4.4(b)). 다시 말해서 $n+1$개의 $x$ 값에 대하여 함수를 샘플링을 하는데, 샘플링 구간은 문제에 따라서 그 폭이 서로 다르기도 하지만, 여기서는 편의상 균일한 구간에 대하여 샘플링을 한다고 가정하자. 그 결과 그림 4.4(c)와 같이 구간 길이 $h = \frac{b-a}{n}$에 대하여 $n+1$개의 샘플링 데이터를 얻는데, 이후부터는 이러한 정보만을 통하여 원래의 함수를 가장 효과적으로 모델링 해줄 수 있는 방법을 사용하게 된다.

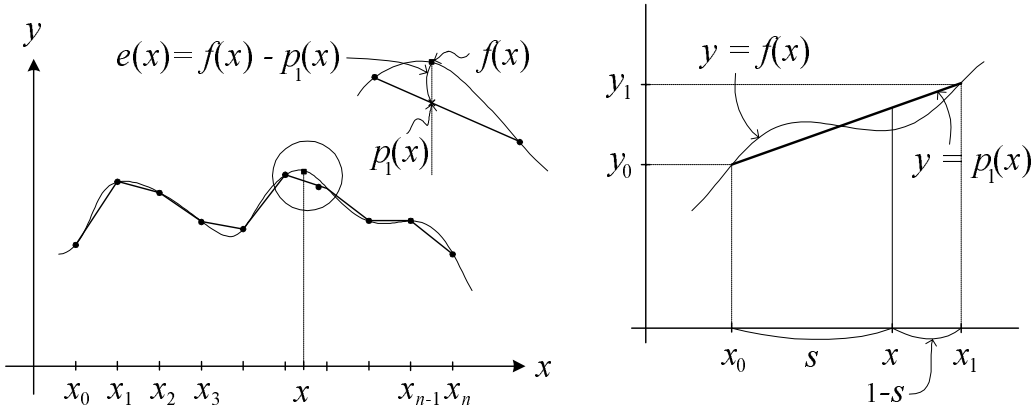
문제는 이러한 정보가 주어졌을 때 원래의 함수를 어떻게 근사화할 것인가 하는 것이다. 그러한 방법 중의 하나가 바로 보간법인데, 이것은 비교적 수학적으로 단순한 형태를 가지는 함수들을 사용하여 샘플링을 통해 획득한 정보를 정확하게 만족시켜주는 새로운 함수를 구하는 방법이다. 일반적으로 가장 많이 사용되는 방법은 효율적인 계산이 가능한 다항식(polynomial)을 기반으로 하는 것이다. $n+1$개의 데이터가 주어졌을 때, $n$차 다항식 $y = p_n(x) \equiv a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$을 사용하면 임의의 $n+1$개의 함수 값 조건을 만족시켜주는 다항식을 구할 수 있다. 이렇게 구한 함수의 그래프는 모든 샘플링 점을 지나가는데, 이렇게 샘플링을 통해 구한 모든 요구 사항을 정확히 만족시켜주는 것을 데이터를 보간한다고 한다. 이 때 구간 $[a,b]$ 사이의 임의의 $x$에 대하여 $p_n(x)$ 값을 함수 $f(x)$의 근사 값으로 사용을 하게 된다. 다항식은 함수의 표현을 위하여 사용되는 수학적인 도구 중 비교적 어느 정도의 함수 표현 능력을 가지면서도 다른 어떤 함수 형태보다도 적은 비용으

로 사용이 가능하기 때문에, 복잡한 함수를 근사화하는데 있어 널리 쓰이고 있다.

하지만 이러한 방식의 보간은 데이터의 양이 많아질수록, 즉 $n$이 커질수록 많은 문제가 발생하게 된다. 우선 주어진 $x$에 대하여 $n$차 다항식을 계산을 하기 위해서는 최소한 $n$번의 곱셈과 $n$번의 덧셈/뺄셈이 필요하기 때문에, $n$이 클 경우 실시간 계산에서는 이러한 비용도 큰 부담으로 작용을 한다. 그러나 더 큰 문제는 $n$이 커질수록 다항식의 그래프가 요동을 치기 때문에, 주어진 $n+1$개의 $x$ 값 주변에서는 근사 값이 비교적 정확하지만 다른 영역에서는 그 정확도가 매우 불안정해진다. 이러한 문제를 해결하기 위한 방법으로서, 데이터가 많을 경우 한 개의 다항식을 사용하여 전체를 보간하는 것이 아니라, 전체 구간을 여러 개의 부분 구간으로 나누어 각 구간에 대하여 서로 다른 다항식을 통하여 보간을 한다. 이렇게 하면 다항식의 차수를 낮출 수가 있는데, 이러한 방식을 토막 다항식(piecewise polynomial)을 사용한 보간이라 한다.

### 2.1.2 선형 보간법

토막 다항식을 사용한다고 할 때, 각 부분 구간 안의 데이터의 개수가 사용할 다항식의 차수와 밀접한 연관이 있으므로 이에 대한 결정을 신중히 내려야 한다. 실시간 렌더링 분야에서와 같이 함수를 근사화하는데 필요한 시간이 중요한 응용 문제에 대해서는 $y = p_1(x) \equiv a_1 x + a_0$ 형태의 1차 다항식을 많이 사용한다. 1차 다항식은 직선을 나타내므로 두 개의 조건에 의하여 결정이 된다. 따라서 위의 데이터를 토막 1차 다항식(piecewise linear polynomial)을 사용하여 근사화하려면, 각 구간마다 한 개씩의 다항식을 사용하여야 한다. 따라서 그림 4.5(a)에서와 같이 각 구간에 대하여 양 끝점을 있는 직선들을 사용하여 그 구간에서의 함수를 근사화하게 된다. 구간 안에서의 임의의 $x$ 값에 대해서는 해당 직선에 대한 함수 값 $p_1(x)$로 원래의




(a) 샘플 데이터의 선형 보간 (b) 매개 변수를 통한 선형 보간

그림 4.5: 토막 다항식을 사용한 선형 보간

함수 $f(x)$를 근사화할 수 있다.

이 그림에서 $i$ 번째 구간 $[x_i, x_{i+1}]$에 대한 두 개의 조건 $y_i = f(x_i)$와 $y_{i+1} = f(x_{i+1})$를 만족시켜주는 1차 다항식 $y = p_1^i(x)$는 다음과 같이 표현할 수 있다.

$$\begin{aligned} f(x) \approx p_1^i(x) &= f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i) \\ &= y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i), \quad x \in [x_i, x_{i+1}] \end{aligned}$$

각 구간에 대하여 동일한 방식으로 계산을 하므로, 편의상 첫 번째 구간 $[x_0, x_1]$에 대한 근사를 생각하도록 하자. 이 때의 다항식 $p_1^0(x)$는 다음과 같이 또 다른 방식으로 표현이 가능하다.

$$\begin{aligned} f(x) \approx p_1^0(x) &= y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) \\ &= \frac{x_1 - x}{x_1 - x_0} y_0 + \frac{x - x_0}{x_1 - x_0} y_1 \end{aligned}$$

이 때 $s = \frac{x - x_0}{x_1 - x_0}$라 하면 이 함수를 다음과 같이 구간 $[x_0, x_1]$에 대하여 '지역적으

로(locally)' 표현을 할 수가 있다.

$$\begin{aligned} p_1^0(x) = y(s) &\equiv (1-s)\cdot y_0 + s\cdot y_1,\ \ \delta_y \equiv y_1 - y_0 \\ &= y_0 + s\cdot \delta_y,\ \ 0 \le s \le 1 \end{aligned}$$

여기서의 새로운 변수 $s$를 매개 변수(parameter)라 하는데, 그림 4.5(b)에 도시된 바와 같이 관심 구간 $[x_0, x_1]$의 길이에 대한 $x$ 값까지의 길이의 비율에 해당한다. 이 변수는 해당 구간의 점에 대해서 0과 1사이의 값을 가지는데, 이러한 매개 변수를 통하여 처리하고자 하는 문제를 해당 구간의 영역으로 지역화를 할 수 있다. 어떠한 방식으로 다항식을 표현을 하건 바로 이러한 것이 컴퓨터 그래픽스 분야에서 가장 널리 쓰이는 보간 방식으로서, 1차 즉 선형의 다항식을 사용하므로 선형 보간법(linear interpolation)이라 한다[4]. 선형 보간은 그 계산량이 매우 적기 때문에, 많은 그래픽스 계산에서 데이터를 빠른 시간 안에 보간을 하여야 할 때 거의 대부분이 방법을 사용한다.

### 2.1.3 선형보간시의 오차

지금까지 보간법, 특히 선형 보간법에 대하여 간략히 살펴보았는데, 이 방법은 원래의 함수를 정확히 표현을 하는 것이 아니므로 수치적인 오차가 발생한다. 선형 보간의 경우 그림 4.5(a)에서와 같이 $f(x)$ 대신에 $p_1(x)$를 사용할 때 $e(x) = f(x) - p_1(x)$ 만큼의 오차가 생기는데, 종종 이에 대하여 정확한 분석을 해야 할 필요가 있다. 보간법 이론에 의하면 오차는 $e(x) = \frac{f^{(2)}(\xi)}{2}(x-x_0)(x-x_1)$와 같이 나타낼 수가 있는데, 구간 $[x_0, x_1]$의 안의 $x$ 값만을 고려한다면 $\xi$는 이 구간에 존재하는 어떤 값임을

[4] 컴퓨터 그래픽스 분야에서는 선형 보간법을 lerp이라고 간결하게 부르기도 한다.

보일 수가 있다.

**잠깐 4.1** 어떻게 보면 오차에 대한 공식은 별로 유용한 정보를 제공하는 것처럼 보이지 않는다. 우선 $\xi$는 구간 $[x_0, x_1]$ 사이에 존재한다고 알려진 미지의 값이고, 오차 공식을 계산하기 위해서는 함수의 2차 미분 값을 구해야 한다. 함수 $f(x)$의 함수 값을 계산하는 것이 매우 어려워 단순한 다항식으로 근사화를 하려 하는 것인데, 일반적으로 함수 값 보다 계산하기가 더 어려운 2차 미분 값을 구해야 한다는 것은 실제적으로 이러한 오차 공식이 유용하게 쓰일 수가 있는지 의심이 간다. 하지만 상황에 따라 이 공식은 선형 보간법의 정확도 측정에 있어 효과적으로 사용할 수가 있다.

예를 들어 0도와 90도 사이의 구간에서 코사인 값 $f(x) = \cos x$를 구하는 문제에 대하여 생각을 해보자. 요즈음은 프로세서의 속도가 워낙 빠르기 때문에, 코사인 값을 직접 계산해도 빠른 속도로 그 값을 정확하게 구할 수가 있다. 하지만 프로세서의 속도가 느려 가능한 한 그에 필요한 계산량을 줄이고 싶을 때[5], 한 가지 적용할 수 있는 방법은 선형 보간법을 사용하는 것이다.

이 때 보간법을 사용하여 구하는 근사 값의 오차가 $10^{-6}$과 같거나 작아야 하는 조건이 있다고 가정하자. 이는 최소한 소수점 이하 5-6 번째 자리까지 정확하게 계산을 하고자 하는 상황이다. 오차의 크기는 구간의 간격을 작게 할 수록, 즉 미리 구하는 샘플 데이터의 개수가 커질수록 작아짐은 쉽게 알 수가 있다. 구간 $[0, \frac{\pi}{2}]$를 $n$개의 구간으로 나누면, 구간의 크기는 $h = \frac{\pi}{2n}$이 된다. 이 때 각 구간에 대하여 선형 보간을 적용하면, $i$ 번째 구간에 대하여 오차에 대한 절대값 $|e_i(x)|$는 다음과 같이 정리를 할 수가 있다.

[5] 간단한 예로 70년대의 소형 계산기를 생각해보자.

$$
\begin{aligned}
|\, e(x)\,| &= |\, \frac{\cos^{(2)}\xi_i}{2}(x-x_i)(x-x_{i+1})\,| \\
&= 0.5\cdot|\,\cos\xi_i\,|\cdot|\,(x-x_i)(x-x_{i+1})\,|,\ \ x,\,\xi_i\in[x_i,x_{i+1}] \\
&= 0.5\cdot|\,\cos\xi_i\,|\cdot|\,x(x-h)\,|,\ \ x\in[0,h],\,\xi_i\in[x_i,x_{i+1}]
\end{aligned}
$$

여기서 $\xi_i$의 정확한 값은 모르지만 $|\cos\xi_i|$ 값은 1보다 클 수가 없고, 또한 구간 $[0,h]$에서의 $|x(x-h)|$의 최대 값은 $x=\frac{h}{2}$일 때 $\frac{h^2}{4}$이므로, $|e_i(x)|\le\frac{h^2}{8}$과 같은 부등식이 성립한다. 이에 대하여 $\frac{h^2}{8}\le 10^{-6}$의 조건을 가하면, 보간의 결과 발생하는 오차를 $10^{-6}$보다 작게 할 수 있다. 이를 정리하면 $h\le 2.82843\cdot 10^{-3}$이 되고, $h=\frac{\pi}{2n}$의 관계를 사용하면 $n=\frac{\pi}{2h}\ge 555.36$과 같이 $n$에 대한 조건을 얻을 수가 있다. 따라서 0과 90도 사이의 구간을 556개 이상의 구간으로 나누어, 선형 보간법을 적용하면 원하는 오차 조건을 만족시킬 수가 있다.

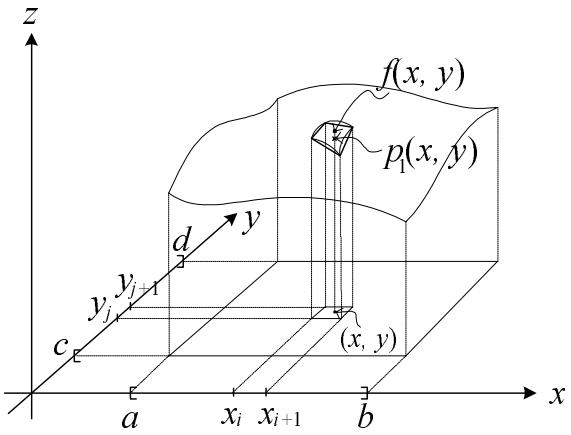
이렇게 구한 $n$ 값에 대하여 샘플 데이터 값을 테이블 형태로 미리 저장을 해놓고, 주어진 $x$ 값에 대하여 해당 구간에서 선형 보간법을 사용하면 실시간적으로 원하는 정확도를 가지는 코사인 값을 구할 수가 있다. 이러한 전처리를 통한 테이블 사용 기법은 과거의 방법처럼 보일지도 모르지만, 계산량을 줄여 최적화된 계산을 수행해야 하는 실시간 계산 분야에서는 아직도 중요하게 사용이 되고는 한다. 앞에서 살펴본 바와 같이, 조명 계산에서 정반사 부분을 계산할 때 멱제곱을 빠르게 계산하기 위하여 테이블을 사용하는 기법이나 5장에서 살펴볼 실시간 텍스춰 매핑을 위한 밉매핑 방법도 그러한 예라 할 수 있다.

### 2.1.4 이선형 보간

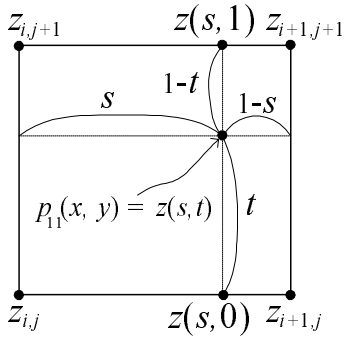
선형 보간은 그 단순성에 기인하여 널리 쓰이고 있는데, 앞 절에서 살펴본 선형 보간 방법은 임의의 개수의 변수를 가지는 함수에 대하여 자연스럽게 확장을 할 수가 있다. 특히 컴퓨터 그래픽스 분야에서는 2차원 공간에서 정의가 되는 래스터 이미지와 3차원 공간에서 정의되는 가상의 세상을 다루어야 하기 때문에, 2차원 및 3차원 공간에서의 선형 보간이 자주 쓰인다. 그림 4.6(a)는 $z = f(x, y)$와 같이 두 개의 변수에 대하여 정의가 되는 함수 $f$를 $[a, b] \times [c, d]$ 영역 안에서 균일하게 샘플링을 하는 모습을 보여주고 있다. 지금 변수 $x$에 대한 구간 $[a, b]$를 $n$개의 부분 구간으로, 그리고 변수 $y$에 대한 구간 $[c, d]$를 $m$개의 부분 구간으로 나누어 샘플링을 하고 있다. 그 결과 2차원 형태의 격자 점 $(x_i, y_j)$ $(i = 0, 1, \cdots, n,\ j = 0, 1, \cdots, m)$들에 대한 함수 값 $z_{ij} = f(x_i, y_j)$를 얻는데, 바로 이러한 값들을 사용하여 주어진 직사각형 영역에서의 함수를 근사화하게 된다.

이 경우에도 앞에서와 마찬가지로 토막 다항식을 사용하여 $[a, b] \times [c, d]$ 영역 안의 각 사각형에 대한 근사 함수를 계산한다. 임의의 $(x, y) \in [a, b] \times [c, d]$가 주어졌을 때, 우선 그 점이 포함되는 영역 $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$을 찾아, 그 안에서의 거리에 대한 비율을 사용하여 선형 보간을 한다(그림 4.6(b)). 이 경우에는 $x$축과 $y$축 등 두 개의 방향이 존재하기 때문에, 각 방향에 대한 비율 $s = \frac{x - x_i}{x_{i+1} - x_i}$와 $t = \frac{y - y_j}{y_{j+1} - y_j}$를 구해야 한다. 이 때, $f(x, y)$에 대한 근사 값 $p_{11}(x, y)$는 다음과 같이 선형 보간을 $x$축 방향에 대하여 두 번 수행을 한 후,

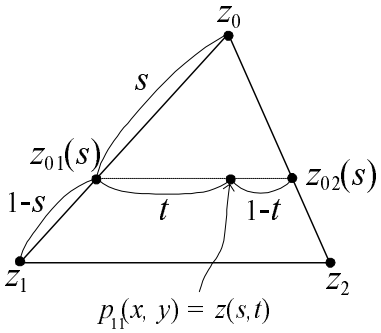
$$
\begin{aligned}
z(s, 0) &= s \cdot z_{i+1,j} + (1 - s) \cdot z_{ij} \\
z(s, 1) &= s \cdot z_{i+1,j+1} + (1 - s) \cdot z_{i,j+1}
\end{aligned}
$$




(a) 이변수 함수의 샘플링




(b) 이선형 보간 1




(c) 이선형 보간 2

그림 4.6: 이선형 보간의 원리

두 보간 값을 $y$축 방향에 대하여 다시 한 번 선형 보간을 하면 원하는 근사값을 구할 수가 있다.

$$p_{11}(x,y) = z(s,t) \equiv t \cdot z(s,1) + (1-t) \cdot z(s,0), \quad 0 \le s \le 1,\ 0 \le t \le 1$$

변수가 두 개일 경우에는 '2차원적으로' 선형 보간을 하는데, 이러한 방식의 보간을 이선형 보간(bilinear interpolation)이라 부른다. 앞에서의 선형 보간 함수 $y = p_1(x) = y(s)$는 해당 구간에서 선형적으로, 다시 말해서 직선의 모양으로 변하는데, 과연 $z = p_{11}(x,y) = z(s,t)$ 함수의 그래프는 어떤 모양을 가질까? 이 함수는 해당 사각형 영역의 네 모서리 점을 보간해야하므로, 그 그래프는 일반적으로 평면일 수가 없음을 쉽게 알 수가 있다. 위의 보간 식들을 사용하여 이선형 보간 함수를 다음과 같이 정리할 수가 있는데,

$$\begin{aligned} z &= p_{11}(x,y) \\ &= z(s,t) = (z_{i+1,j+1} - z_{i+1,j} - z_{i,j+1} + z_{ij})st + (z_{i+1,j} - z_{ij})s \\ &\quad + (z_{i,j+1} - z_{ij})t + z_{ij} \end{aligned}$$

이 함수는 $s$와 $t$에 대한 이차식으로 표현이 되는 이차 곡면임을 알 수 있다[6]. 좀 더 구체적으로 말하면 이 함수의 그래프는 여러 종류의 이차 곡면 중 쌍곡포물면(hyperbolic paraboloid) 형태의 모양을 가진다. 따라서 이선형 보간은 다른 관점에서 보면 주어진 함수 $z = f(x,y)$를 쌍곡포물면으로 근사화해주는 방법이라 할 수가 있다.

[6] $s$와 $t$는 $x$와 $y$에 대하여 각각 선형적으로 표현이 되므로, 이 함수는 또한 $x$와 $y$에 대한 이차 함수이다.

한 가지 재미있는 사실은 이 함수가 $s$와 $t$에 대해서는 2차식이지만, 각 변수에 대해서는 1차식이라는 점이다. 즉 $s$와 $t$ 각 매개 변수에 대하여 정리하면, 1차식으로 표현이 되는데, 이것이 의미하는 바는 무엇일까? 이는, 예를 들어, $t$값을 고정하고, $s$ 값을 0에서 1로 변화시키면서 이선형 보간을 해보면, 그 결과 함수 값은 쌍곡포물면 상에서 직선을 따라 움직인다는 것을 의미한다. 쌍곡포물면은 무한개의 직선을 원기둥의 둘레에 배치한 후 이를 '비스듬히 찌그러뜨려' 만든 2차 곡면으로서, 이선형 보간과 재미있게 연결이 됨을 관찰할 수 있다. 요약을 하면, 이선형 보간은 $s$와 $t$ 각 매개 변수에 대하여 선형 보간을 하는 것으로서, 그 결과 형성되는 함수는 곡면의 형태를 가지지만 그 안에는 엄연히 선형성이 내재되어 있다.

지금 살펴본 이선형 보간은 직사각형 형태의 구역에 대한 보간이었는데, 컴퓨터 그래픽스 분야에서는 그림 4.6(c)에서와 같이 삼각형 영역에 대한 선형 보간이 자주 사용된다. 한 가지 보간 방법은 보간을 하려는 점 $(x, y)$가 주어졌을 때, 삼각형의 두 변에 대하여 선형 보간을 하여 $z_{01}(s)$와 $z_{02}(s)$를 구한 후, 이들을 수평 방향으로 선형 보간을 하는 것이다. 이는 이 장의 주제인 래스터화 과정에서 스캔 라인을 따라 계산이 일어날 때 자연스럽게 사용이 되는 방법으로서, 그 식은 다음과 같이 정리할 수가 있다.

$$\begin{aligned} z_{01}(s) &= s \cdot z_1 + (1-s) \cdot z_0 \\ z_{02}(s) &= s \cdot z_2 + (1-s) \cdot z_0 \\ p_{11}(x,y) &= z(s,t) \equiv t \cdot z_{02}(s) + (1-t) \cdot z_{01}(s) \end{aligned}$$

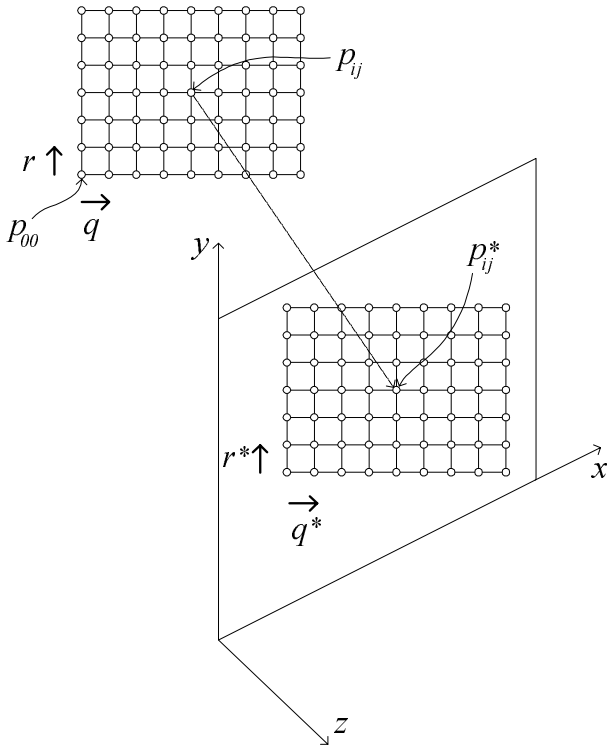
삼각형에 대한, 즉 세 점에 대한 선형 보간은 중심자리 좌표(Barycentric coordinates)를 사용하여 더 자연스럽게 표현할 수 있는데, 이에 대한 설명은 생략하도록

하겠다. 어떠한 방식을 사용하건, 삼각형 영역에 대한 선형 보간의 결과 생성되는 근사 함수는 평면이 됨을 명심하기 바란다. 선형 보간은 세 개의 변수에 의하여 정의되는 함수에 대한 삼선형 보간(trilinear interpolation)으로 자연스럽게 확장을 할 수가 있는데, 어떠한 방식으로 수행이 되는지 생각해보기 바란다. 지금까지 선형 보간에 대하여 이론적인 측면을 간략하게 살펴보았는데, 한 가지 더 고려를 해야할 사항은 이러한 방법을 실제로 적용할 때 어떻게 하면 효율적으로 계산을 할 수 있을 것인가 하는 점이다. 무엇보다도 이선형 보간을 하기 위해, 또는 삼선형 보간을 하기 위해 최소한으로 필요한 부동 소수점 연산의 회수는 얼마인지 살펴보기 바란다. 또한 실제 구현에 있어 상황에 따라 다음 절에 설명할 점진적 계산 방법을 적용할 수 있는지에 대해서도 항상 생각을 해봐야 할 것이다.

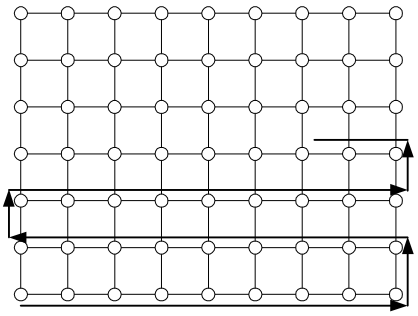
## 2.2 점진적 계산

여러 번 강조한 바와 같이 OpenGL 프로그램과 같은 실시간 소프트웨어를 제작할 때에는 가능한 한 불필요한 계산을 제거하도록 갖은 노력을 기울여야 한다. 이러한 노력의 한 예로서 본 절에서는 어느 정도의 규칙성을 가지면서 비슷한 계산이 반복적으로 수행이 되는 문제에서, 그에 대한 계산 구조를 잘 이용함으로써 계산량을 최적화할 수 있는 간단한 기법에 대하여 알아보겠다. 이를 위하여 그림 4.7(a)에서와 같은 평행 투영 문제를 생각해보자. 여기에는 3차원 공간의 임의의 평면 상에 가로 $m$개, 세로 $n$개의 점으로 이루어진 $mn$개의 격자 점이 있다. 왼쪽 아래 모서리의 점 $p_{00}$의 좌표는 $(x_{00}\ y_{00}\ z_{00})^t$이고, 오른쪽으로 한 칸씩 움직일 때마다 $q = (q_x\ q_y\ q_z)^t$ 만큼, 그리고 위로 한 칸씩 움직일 때마다 $r = (r_x\ r_y\ r_z)^t$ 만큼 좌표 값이 변한다고 하자.

지금 이 격자 점들을 $xy$ 평면으로 평행 투영을 한 투영 점의 좌표 값을 계산




(a) 격자 점의 평행 투영



(b) 점진적 계산 순서

그림 4.7: 점진적 계산의 예

하려고 한다. 계산량에 대한 고려가 없이 단순하게 계산을 하면 다음과 같다. 우선 $i$행 $j$열에 해당하는 점의 좌표는 $p_{ij} \equiv (x_{ij}\ y_{ij}\ z_{ij})^t = p_{00} + i \cdot q + j \cdot r = (x_{00} + i \cdot q_x + j \cdot r_x\ \ y_{00} + i \cdot q_y + j \cdot r_y\ \ z_{00} + i \cdot q_z + j \cdot r_z)^t$가 되고, 평행 투영은 아핀 변환이므로 적절한 상수 $a, b, \cdots, h$에 대하여 투영 점 $p^*_{ij} = (x^*_{ij}\ y^*_{ij})^t$을 다음과 같이 구할 수가 있다.

$$\begin{aligned} x^*_{ij} &= a \cdot x_{ij} + b \cdot y_{ij} + c \cdot z_{ij} + d \\ y^*_{ij} &= e \cdot x_{ij} + f \cdot y_{ij} + g \cdot z_{ij} + h \end{aligned}$$

따라서 모든 격자 점에 대하여 이러한 방식으로 투영 점을 계산하면 되지만, 이는 매우 비효율적임은 분명하다.

이 문제의 성격을 살펴보면, 일정한 순서로 격자 점을 따라가면서 같은 형태의 계산이 반복이 되고 있으며, 또한 한 격자 점의 계산 결과를 다음 격자 점을 계산하는데 유용하게 사용할 수 있음을 어렵지 않게 알 수가 있다. $p_{ij}$에 대한 투영 점 $p^*_{ij}$의 계산한 후 다음 $p_{i+1j}$에 대한 투영 점 $p^*_{i+1j}$의 값을 계산하려 한다고 하자. $p_{ij}$와 $p_{i+1j}$ 사이에는 $p_{i+1j} = p_{ij} + q = (x_{ij} + q_x\ \ y_{ij} + q_y\ \ z_{ij} + q_z)^t$와 같은 관계가 있기 때문에 이 두 점의 투영 점의 좌표 사이에도 다음과 같은 유사한 관계가 존재한다.

$$\begin{aligned} p^*_{i+1j} &= \begin{pmatrix} a \cdot x_{i+1j} + b \cdot y_{i+1j} + c \cdot z_{i+1j} + d \\ e \cdot x_{i+1j} + f \cdot y_{i+1j} + g \cdot z_{i+1j} + h \end{pmatrix} \\ &= \begin{pmatrix} a \cdot x_{ij} + b \cdot y_{ij} + c \cdot z_{ij} + d \\ e \cdot x_{ij} + f \cdot y_{ij} + g \cdot z_{ij} + h \end{pmatrix} + \begin{pmatrix} a \cdot q_x + b \cdot q_y + c \cdot q_z \\ e \cdot q_x + f \cdot q_y + g \cdot q_z \end{pmatrix} \end{aligned}$$

따라서 $q^* = (a \cdot q_x + b \cdot q_y + c \cdot q_z\ \ e \cdot q_x + f \cdot q_y + g \cdot q_z)^t$라 하면, 바로 오른쪽의 점

을 투영을 할 때에는 바로 직전에 구한 좌표 값에 $q^*$만 더하면 된다. 여기서 이 벡터는 처음에 한 번만 계산을 하면 되므로 덧셈 두 번만 수행하면 오른쪽의 투영 점을 계산할 수 있다. 물론 바로 왼쪽의 투영 점을 계산하려 하면 이 벡터만큼 빼주면 된다. 같은 이유로 상하로 움직일 때는 $r^* = (a \cdot r_x + b \cdot r_y + c \cdot r_z \;\; e \cdot r_x + f \cdot r_y + g \cdot r_z)^t$ 벡터를 사용하면 된다. 따라서 $mn$개의 격자 점을 투영하려할 때, 우선 최초 격자점 $p_{00}$에 대한 투영 계산을 통하여 투영 점을 구하고, 나머지 $mn-1$개의 점은 그림 4.7(b)에서와 같이 규칙적으로 스캔을 하면서 각 점 당 덧셈/뺄셈 두 번을 통하여 효율적으로 투영 점을 계산할 수 있다.

이 단순한 예에서와 같이 직전의 계산 결과를 최대한으로 사용하여 다음 계산을 수행하는 것을 점진적 계산(incremental computation)이라 한다. 이러한 계산 구조는 마치 귀납법을 통한 증명 방식을 연상시키는데, 이전의 계산 결과를 최대한으로 이용함으로써 계산의 효율을 높일 수가 있다. 물론 반복성을 가지는 모든 계산을 이와 같이 점진적으로 수행을 할 수 있는 것은 아니다. 하지만 이 예에서와 같이 규칙적인 계산 과정에서 점진적인 구조를 이용할 수 있을 경우 많은 계산을 줄일 수가 있다. 렌더링 파이프라인 과정에서는 바로 래스터화 문제에 이러한 점진적인 구조가 존재하고, 따라서 이러한 성질을 최대한을 이용을 하여야 하는데 이에 대해서는 뒤에서 다시 살펴보겠다.

# 제 3 절 OpenGL에서의 래스터화

## 3.1 래스터화와 프래그먼트

이제 다시 원래의 문제로 돌아와 OpenGL 시스템에서의 래스터화 과정에 대하여 살펴보자. 378쪽의 그림 4.1은 OpenGL 렌더링 파이프라인에서의 래스터화 과정과 그 이후의 렌더링 계산을 도시하고 있다. 넓은 의미로는 래스터화 이후 프레임 버퍼까지의 모든 과정을 단순히 래스터화라고 하기도 한다. 반면 OpenGL에서는 좁은 의미로 래스터화라는 용어를 사용하는데, 특히 다음의 두 가지 계산을 수행하는 것을 주목적으로 한다. 첫째는 윈도우 좌표계로 변환된 각 기하 프리미티브들이 화면상에서 차지하는 화소들의 위치를 찾는 것이다. 한편 기하 프리미티브들의 기본 요소인 꼭지점에는 여러 가지 종류의 데이터가 연관되어 있다고 하였는데, 벡터 형태의 데이터인 기하 프리미티브들이 래스터들로 변환이 되면서, 각 연관 데이터들을 기하 프리미티브들의 내부에 해당하는 화소들에게 붙여줄 적절한 값을 계산하여야 하는데, 바로 이것이 두 번째 목적이다.

그림 4.8은 래스터화 과정에 관한 것인데, 이 과정의 입력은 기하 프리미티브들의 리스트이고, 출력은 그것들이 투영되는 화소들의 리스트이다. OpenGL에서는 래스터화의 결과 생성되는 화소와 그에 연관된 정보들을 프래그먼트(fragment)라 부른다. 각 프래그먼트는 정확히 말하면 1. 윈도우 좌표계 공간에서의 화소의 위치인 $(x_{wd}, y_{wd})$와 그에 연관된 데이터(associated data), 즉 2. 그 화소를 통하여 보이는 기하 프리미티브의 해당 지점까지의 깊이 정보인 $z_{wd}$, 3. 그 지점의 색깔 $(r, g, b, a)$, 그리고 4. 그 지점에 매핑이 되는 텍스춰 좌표 $(s, t, r, q)$로 구성이 된다. 일단 이러한 프래그먼트로 변환이 되면, 그 화소에 해당하는 기하 프리미티브의 연속적인 정보는 사라지고, 래스터화 과정 이후에는 모든 계산이 프래그먼트 단위로 일어나게

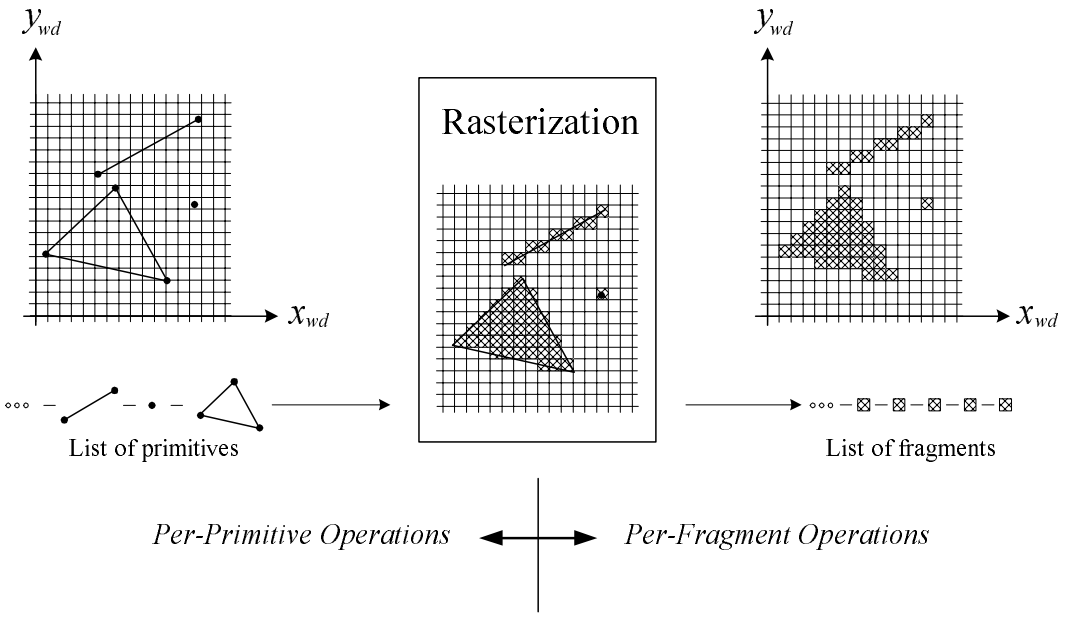



그림 4.8: 래스터화 계산의 결과

된다.

따라서 래스터화 계산의 결과 렌더링 파이프라인을 따라 흘러가는 데이터의 형태에 큰 변화가 일어나는데, OpenGL에서는 기하 프리미티브 조합 이후부터 래스터화 직전까지의 계산을 프리미티브별 연산(per-primitive operations)이라 하고, 래스터화 이후는 프래그먼트별 연산(per-fragment operations)이라 한다. 문맥에 따라서 이 두 가지 연산을 각각 꼭지점별 연산(per-vertex operations)과 화소별 연산(per-pixel operations)이라 하기도 한다.

## 3.2 화소의 정의와 연산의 종류

래스터 계산을 위하여 한 가지 명확히 결정해야 할 것은 OpenGL이 어떤 형태의 윈도우 좌표계를 사용하는가 하는 것이다. 앞에서도 언급하였듯이 OpenGL에서는 $x$축은 오른쪽, $y$축은 위쪽으로 값이 증가를 한다. 그렇다면 정수 $x$, $y$에 대해 $(x, y)$라는 값이 나타내는 화소의 위치는 정확히 어디일까? 그림 4.9가 나타내듯이 OpenGL

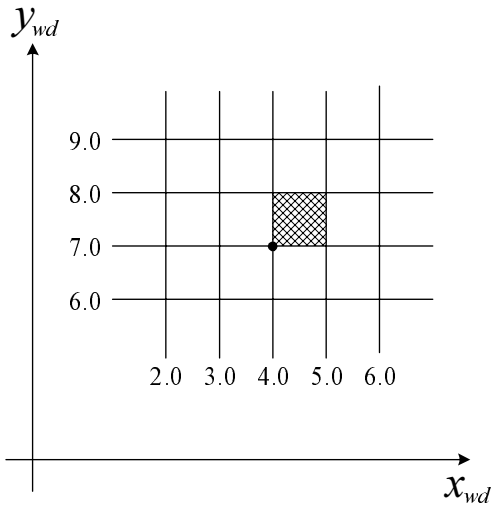



그림 4.9: OpenGL에서의 화소

에서 $(x, y)$라는 화소는 윈도우 좌표계에서 정수 격자 점 $(x, y)$를 왼쪽 아래 모서리로 하는 각 변의 길이가 1인 사각형을 의미한다. 따라서 화소 $(x, y)$의 중심의 정확한 위치는 $(x + 0.5, y + 0.5)$가 되는데, 이 그림에는 화소 (4, 7)이 화면에서 차지하는 영역이 도시되어 있다. 어떤 래스터 시스템에서는 정수 좌표 $(x, y)$가 화소의 왼쪽 아래 모서리가 아니라 화소의 중심을 나타내기도 한다. 절대적으로 어떤 방식이 더 좋다고는 말할 수는 없고, 단지 두 방법의 차이는 래스터 이미지가 각 축 방향으로 0.5 화소만큼 씩 밀려 있다는 점인데, 한 래스터 시스템을 기준으로 개발된 래스터 알고리즘은 다른 시스템으로 쉽게 변환을 할 수 있다.

래스터화 계산에 관하여 한 가지 더 결정을 해야할 중요한 사항중의 하나는 어떤 정확도로 이 과정을 구현할 것인가 이다. 다른 계산 과정도 마찬가지겠지만 특히 래스터화 계산은 빠르게 수행이 되어야 한다. 장면이 조금만 복잡해도 한 장의 이미지를 렌더링하는데 있어 수만 또는 수십만 개의 기하 프리미티브들에 대하여 그것들이 투영이 되는 화소들을 찾아주어야 한다는 점을 생각해보면 래스터화 계산의 방대함을 추측할 수 있다. 또한 래스터화 계산은 최종 화질에 결정적인 영향

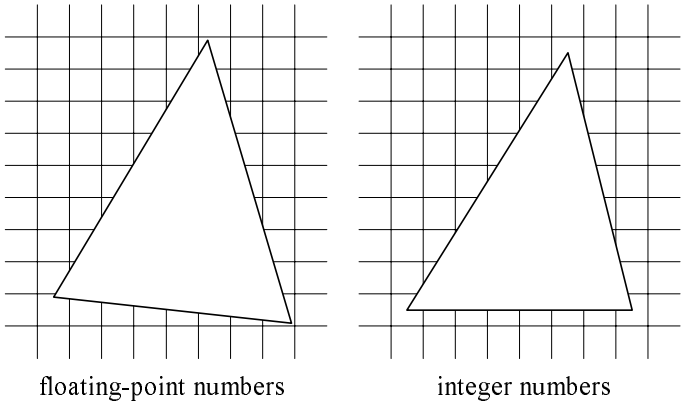



그림 4.10: 래스터화 계산의 정확도

을 미치므로 가능한 한 정확하게 계산을 수행해야 한다[7]. 따라서 많은 렌더링 시스템에서 래스터화 계산을 빠르고 정확하게 수행하기 위하여 이 부분을 하드웨어로 구현하거나 그러지 못할 경우 코드를 최적화하기 위하여 많은 노력을 기울인다.

OpenGL의 렌더링 파이프라인에서 일반적으로 래스터화 직전까지의 모든 계산은 부동 소수점 숫자를 사용한다. 문제는 래스터화 과정에서도 계속해서 부동 소수점 숫자를 사용을 할 것인가 인데, 이는 상황에 따라 적절하게 결정을 해야한다. 부동 소수점 연산은 정확한 반면 상대적으로 계산 비용이 높다. 반면 정수 연산은 빠르게 래스터화 과정을 구현할 수 있지만 정확도에 문제가 발생할 수가 있다. 과거에는 실시간 렌더링이나 3차원 게임과 같이 속도가 가장 중요한 척도중의 하나인 응용 소프트웨어나 하드웨어를 개발할 때에는 `float`나 `double` 타입의 부동 소수점 연산은 가능한 한 피하고 정수 연산을 사용하도록 하였다. 즉 부동 소수점 연산은 정수 연산에 비하여 상대적으로 많이 느렸기 때문에, 부동 소수점 숫자를 사용하여야 하는 경우에도 이를 정수로 바꾸어 계산을 한 것이다. 그러한 예로 윈도우 좌표계에서의 부동 소수점 숫자를 정수로 반올림하여 정수 연산만을 사용하여 속도를

[7] 같은 장면에 대하여 서로 다른 OpenGL 가속기를 사용하여 렌더링한 이미지들을 비교해보면 그 정확도나 전체적인 이미지에 대한 느낌의 차이를 발견할 수 있다. 이러한 사실은 종종 그래픽스 가속기의 성능 평가에 있어 하나의 척도로 사용이 된다.

높이는 방법을 들 수가 있다. 하지만 이 방법의 경우 그림 4.10에서와 같이 윈도우 좌표계로 투영된 기하 프리미티브의 모양이 어느 정도 변형이 되는 것을 피할 수 없다. 특히 이러한 상황에서 기하 물체에 대하여 애니메이션을 해보면 물체가 부드럽게 움직이지를 않고 정수 격자 점을 따라 급격히 움직이는 것을 쉽게 관찰할 수가 있는데, 이는 매우 바람직하지 못한 상황이라 하겠다.

이렇게 단순히 정수 타입의 숫자로 변환을 하는 것은 별로 바람직하지 않은데, 정수 연산과 부동 소수점 연산의 대안으로 정수 연산을 사용하면서도 어느 정도의 정확도를 제공하는 고정 소수점 연산(fixed-point operations)을 고려할 수 있다. 래스터화 계산을 수행할 때 단순히 윈도우 상에서의 좌표 값인 $x_{wd}$와 $y_{wd}$ 값뿐만 아니라, 꼭지점에 연관된 깊이 정보, 색깔, 텍스춰 좌표 값에 대해서도 효과적인 처리를 해주어야 한다. 이러한 계산은 기본적으로 부동 소수점 연산을 사용해야 하지만, 이것이 상황에 따라 불가능하다면 고정 소수점 연산을 사용할 수 있는데, 종종 만족할 만한 결과를 산출한다.

여기서 OpenGL의 공개 구현 코드인 Mesa에서 사용하는 방식을 통하여 고정 소수점 연산에 대하여 살펴보자[8]. 우선 고정 소수점 숫자를 표현하기 위하여 32비트의 정수를 사용하는데, 이 때 소수점이 기존의 정수 타입의 숫자처럼 가장 오른쪽에 있는 것이 아니라 오른쪽에서 11번째 자리에 있다고 가정을 한다. 따라서 이 숫자 체계에서는 정수 1은 16진수로 0x00000800 값으로 저장이 된다. 이 경우 정수와 고정 소수점 숫자는 다음과 같은 매크로를 사용하여 서로간에 변환을 할 수 있다.

```
#define FIXED_SHIFT 11
#define IntToFixed(X) ((X) << FIXED_SHIFT)
#define FixedToInt(X) ((X) >> FIXED_SHIFT)
```

물론 RGBA 색깔의 각 채널과 같이 무부호 문자로 표현된 데이터가 있다면 다음

[8] B. Paul. *The Mesa 3D Graphics Library*, http://www.mesa3d.org, 1999.

과 같이 고정 소수점 숫자에서 원래의 타입으로 돌아올 수가 있다.

```
#define FixedToUns(X) (((unsigned int)(X)) >> 11)
```

또한 텍스춰 좌표와 같이 부동 소수점 숫자가 있다면, 이 또한 다음과 같은 매크로에 의하여 고정 소수점 숫자와 서로 변환이 가능하다.

```
#define FIXED_SCALE 2048.0f
#define ONE_OVER_FIXED_SCALE 0.00048828125f
#define FloatToFixed(X) ((int) ((X)*FIXED_SCLAE))
#define FixedToFloat(X) ((X)*ONE_OVER_FIXED_SCALE)
```

일단 고정 소수점 숫자로 변환을 하면, 정수 연산만 사용하여 마치 부동 소수점 연산을 하는 것과 같은 효과를 얻을 수 있다. 물론 계산이 다 끝난 후에는 다시 원래의 데이터의 타입으로 변환을 하면 된다. 한 가지 주의를 해야할 것은 고정 소수점 숫자는 32비트의 이진수 안에서 소수점의 위치가 고정이 되어 있으므로, 계산 중에 이 숫자가 허용하는 값의 범위를 벗어나지 않도록 해야 한다는 점이다. 위의 예에서는 2진수로 소수점 이하 11자리까지 표현할 수 있으므로, $2^{-11} \approx 0.00048828$정도의 정확도를 제공하는데, 이 정도라면 일반적으로 래스터화 과정에서는 충분하다고 할 수 있다. 이렇게 값비싼 부동 소수점 연산을 정수 연산으로 대치를 할 수 있는데, 이러한 기법은 단지 그래픽스 렌더링에서뿐만 아니라 어느 정도 부동 소수점 연산의 정확도의 손실이 허용이 되는 여러 분야의 응용 문제에서 계산 비용을 줄이기 위하여 유용하게 쓰인다[9].

다음 절로 넘어가기 전에 사용할 연산의 타입을 선택하는데 있어 영향을 미치는 요인에 대하여 한 가지 더 생각을 해보자. 앞에서도 말했듯이 부동 소수점 연산은 정수 연산에 비하여 상당히 느린 연산이라고 알려져 왔는데, CPU의 종류에 따라

[9]여기서 한 가지 고려를 해야할 것이 있다. 고정 소수점 연산을 사용하려면 부동 소수점 숫자와의 변환이 일어나야 하는데, CPU에서의 연산을 생각을 해보면 타입 변환 비용 또한 만만치가 않다. 따라서 수행할 연산의 회수가 필요한 변환의 회수보다 훨씬 많다면 결과적으로 비용을 줄일 수가 있지만, 그렇지 않다면 고정 소수점의 사용을 재고해야 한다.

| | 정수 타입 | | | 부동 소수점 타입 | | |
|---|---|---|---|---|---|---|
| | ± | × | ÷ | ± | × | ÷ |
| Intel 386/387 | 2 | 9-38 | 43 | 23-34 | 27-35 | 89 |
| Intel i486 | 1 | 12-42 | 43 | 10 | 11 | 62(35) |
| Intel Pentium | 1 | 10 | 46 | 3 | 3 | 33(19) |
| PowerPC 604 | 1 | 4 | 20 | 3 | 3 | 31(18) |
| MIPS R4x00 | 1 | 10 | 69 | 4 | 8(7) | 36(23) |

표 4.1: CPU에 따른 연산 비용의 비교(괄호안은 단일 정밀도)

다르기는 하지만 사실 그 간격이 많이 좁혀져 왔다. 표 4.1은 각 종류의 연산을 수행하는데 필요로 하는 CPU 사이클의 회수를 보여주고 있다[10]. 부동 소수점 연산의 상대적인 비용이 과거에 비해 줄기는 했지만, 아직도 부동 소수점 연산, 특히 나눗셈은 정수 연산에 비해 상당히 비용이 높음을 알 수 있다. 물론 CPU가 연산을 반복적으로 수행할 때 사용하는 숫자의 타입과 연산의 종류에 따라 연산의 수행이 서로 중첩(overlap)되는 정도가 다르고, 또한 연산이 정수 타입인지 부동 소수점 타입인지에 따라 캐쉬에 로드하여야 하는 데이터의 양이 다르다. 따라서 여기서의 사이클 회수가 실제의 계산 속도라고는 할 수는 없지만 프로그래머 입장에서는 CPU를 비롯하여 사용하는 시스템의 제반 성능과 또한 컴파일러의 최적화 방식을 고려를 하여 최적의 프로그램을 작성하려는 노력이 필요하다고 할 것이다.

## 3.3 점의 래스터화

이제 OpenGL의 문맥에서 점, 선분, 다각형 등 각 기하 프리미티브에 대한 래스터화 과정에 대하여 살펴보자. 우선 점은 그 좌표가 주어졌을 때 쉽게 래스터화를 할 수 있다. 단 이 과정에 영향을 미치는 몇 가지 요인이 있는데, 그 중 하나는 점을 얼마나 크게 그릴 것인가 하는 것이고, 다른 하나는 앤티앨리어싱 기법을 적용할 것

[10] C. Hecker. "More Compiler Results and What To Do About it," *Game Developer Magazine*, pp. 14-21, August/September 1996.

인가 이다. 특히 후자는 래스터화 방식에 큰 영향을 미치는데 여기서는 앤티앨리어싱 기법을 적용하지 않는다고 가정하자.

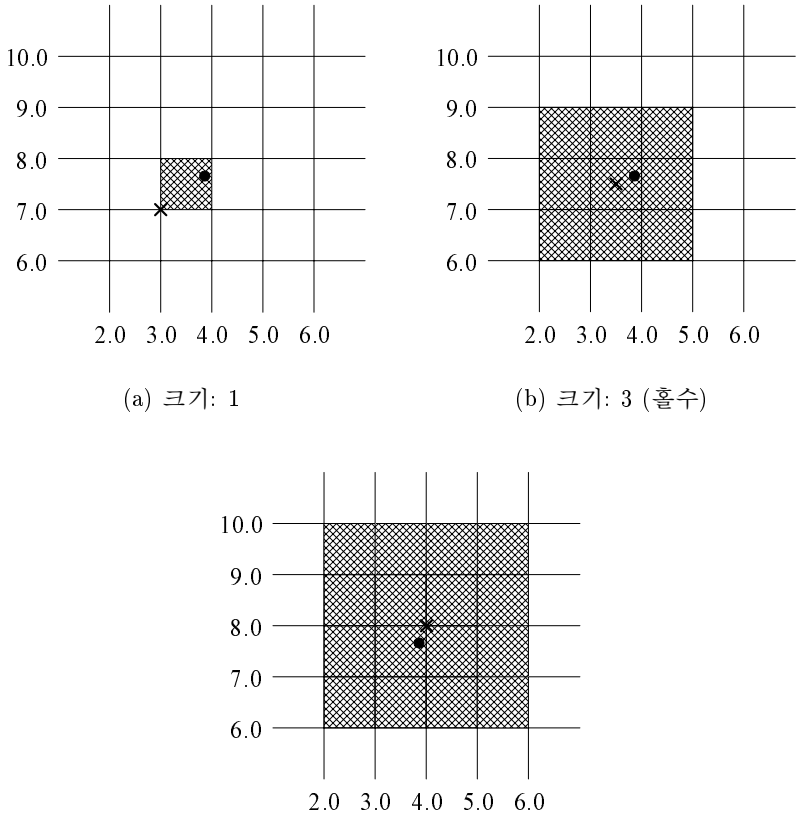
점의 크기는 void glPointSize(GLfloat size); 함수를 통하여 설정할 수 있다. 여기서 size는 그리고자 하는 점의 크기를 나타내는데, OpenGL의 디폴트 상태에서는 점의 크기가 1.0으로 초기화되어 있다. 이 경우 윈도우 좌표 $(x_{wd}, y_{wd})$가 주어졌을 때, 이 점에 해당하는 화소의 위치는 자연스럽게 $(x, y) = (\lfloor x_{wd} \rfloor, \lfloor y_{wd} \rfloor)$가 된다(그림 4.11(a)). 만약 크기가 1.0이 아니면 앤티앨리어싱을 하지 않는 상태에서는 이 값을 가장 가까운 정수로 반올림을 하는데, 그 값이 0이 되면 1로 변환한다. 또한 OpenGL의 구현에 따라 다르기는 하나 최대로 허용되는 점의 크기에 제한이 있어 만약 반올림한 값이 최대 값보다 크면 최대 값으로 바뀌게 된다. $(x_{wd}, y_{wd})$에 대하여 점의 크기가 홀수이면 $(x, y) = (\lfloor x_{wd} \rfloor + \frac{1}{2}, \lfloor y_{wd} \rfloor + \frac{1}{2})$이, 만약 짝수라면 $(x, y) = (\lfloor x_{wd} + \frac{1}{2} \rfloor, \lfloor y_{wd} + \frac{1}{2} \rfloor)$이 점을 그리는데 있어 중심이 된다(그림 4.11(b)와 (c)).

앞에서 강조를 하였듯이 래스터화 계산의 결과 기하 프리미티브는 프래그먼트 형태로 변환이 된다. 점의 경우 상황에 따라 한 개 또는 그 이상의 화소가 선택이 되는데, 각 화소들이 프래그먼트들로 변환이 된다. 이 때 원래의 점에 연관된 데이터들이 그대로 각 프래그먼트들을 구성하는데, 텍스춰 좌표의 경우에만 $(s, t, r, q)$가 $(\frac{s}{q}, \frac{t}{q}, \frac{r}{q})$로 바뀌게 된다.

## 3.4 선분의 래스터화

### 3.4.1 'diamond-exit' 규칙

선분은 $p_0 = (x_0, y_0)$과 $p_1 = (x_1, y_1)$ 등 두 개의 윈도우 좌표에 의해 정의가 되는데, 그에 대한 래스터화 과정은 점보다 훨씬 복잡해진다. 여기서의 좌표 값은 윈도




(a) 크기: 1

(b) 크기: 3 (홀수)

(c) 크기: 4 (짝수)

그림 4.11: 점 $(3.86, 7.69)$의 래스터화

우 좌표계에서 정의되는 부동 소수점 숫자인데 이 절에서는 편의상 $_{wd}$라는 첨자를 사용하지 않겠다. 선분의 속성은 두 개의 함수, 즉 void glLineWidth(GLfloat width); 함수와 glLineStipple(GLint factor, GLushort pattern); 함수를 사용하여 설정할 수 있다. 전자는 선분의 두께를 결정하고, 후자는 실선으로 그릴지, 점선으로 그릴지 등의 패턴을 결정한다. 물론 디폴트는 선분의 두께는 1이고 패턴은 실선인데, 여기서는 위에서와 같이 앤티앨리어싱을 하지 않는 디폴트 상태를 고려하겠다.

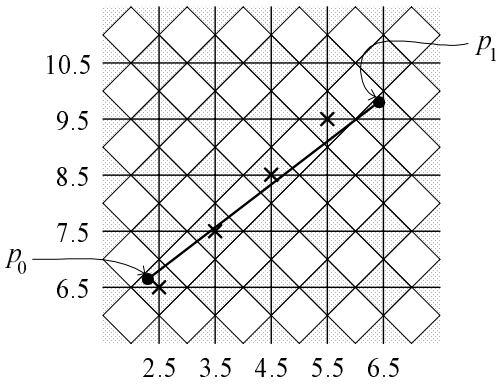



그림 4.12: 'diamond-exit' 규칙

선분의 래스터화 계산의 가장 기본적인 요소는 물론 선분이 지나가는 영역의 화소를 찾는 것이다. 여기서 단순히 '선분이 지나가는 영역'이라고 하였는데, 주어진 선분에 대하여 어떻게 화소를 선택할 지에 대한 정확한 기준이 설정이 되어야 일관된 렌더링 결과를 얻을 수가 있다. OpenGL에서는 'diamond-exit' 규칙('diamond-exit' rule)이라는 규칙을 적용하는데, 그림 4.12를 보면 이 규칙을 쉽게 이해할 수 있다. 이 그림에서 편의상 격자 점은 그림 4.10에서와 같이 OpenGL 화소의 왼쪽 아래 모서리가 아니라 화소의 중심으로 되어 있다. 즉 격자 점은 화소의 중심을 나타내는데, 각 프래그먼트의 중점 $(x_f, y_f)$에 대하여 다음과 같은 다이아몬드 형태의 영역을 생각하자.

$$R_f = \{\,(x,y)\,|\,|x - x_f| + |y - y_f| < \frac{1}{2}\,\}$$

그림에서 어둡게 칠한 부분이 바로 각 화소, 즉 프래그먼트에 대하여 지정된 영역

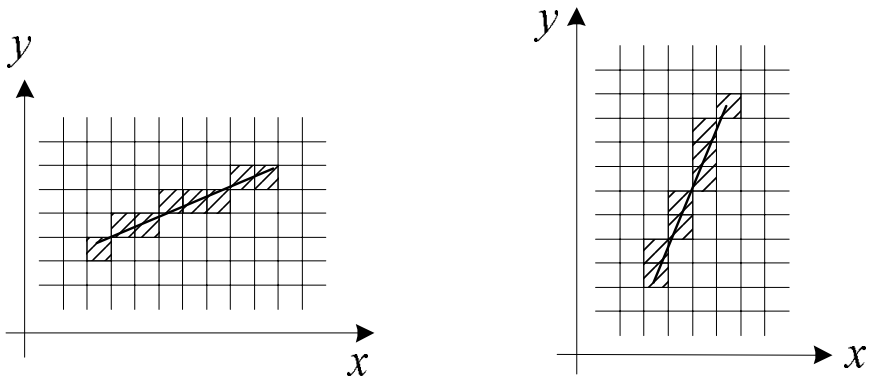
인데, 이는 각 화소의 고유 영역이라고 할 수 있다. 'diamond-exit' 규칙에서는 선분이 화면상에서 $p_0$부터 시작하여 $p_1$로 가면서 지나갈 때 거쳐가는 다이아몬드 영역에 해당하는 모든 화소를 선택을 하도록 되어 있다. 단 끝점 $p_1$이 어떤 다이아몬드 영역에 들어가면 그에 해당하는 화소는 제외를 해야 한다. 따라서 이 그림에서는 래스터화의 결과 가위표로 표시된 네 개의 화소가 선택이 된다.

이 방법을 정확하게 구현을 하기 위해서는 비교적 많은 양의 계산을 해야한다. OpenGL에서는 구체적으로 어떤 방법을 적용하도록 규정이 되어 있지 않다. 따라서 어떤 방법을 사용하건 이 규칙을 사용하여 래스터화를 하였을 때의 정확한 결과와 크게 다르지 않으면 허용이 되는데[11], 다음과 같은 약간의 추가적인 규칙을 따라야 한다.

래스터화 과정에서 2차원 평면 상의 선분은 기울기에 따라 두 가지 부류로 나누어진다(그림 4.13). 기울기가 -1과 1 사이에 들어오는 선분을 $x$-우선 선분($x$-major line)이라 하고, 그렇지 않으면 $y$-우선 선분($y$-major line)이라 한다. 즉 $x$-우선 선분은 수평에 가깝게 누운 선분이고, $y$-우선 선분은 수직에 가깝게 선 선분이다. OpenGL에서는 $x$-우선 선분에 대한 래스터화 계산을 수행할 때, 화소 격자의 각 열에 대하여 두 개 이상의 화소를 선택하지 못하도록 되어 있다. 이는 일반적으로 통용되는 제약으로서 이러한 조건을 만족시키려면 래스터화를 할 때 $x$축으로 한 칸씩 이동하면서 적절한 화소를 한 개씩 찾도록 하면 된다. 마찬가지로 선분이 $y$-우선이라면 화소 격자의 같은 줄에 있는 화소가 두 개 이상 선택이 되면 안 된다.

또 한 가지 OpenGL에서 요구하는 중요한 규칙은 두 개의 선분이 한 점에서 만날 때, 래스터화 알고리즘은 두 선분이 만나는 지점에 대하여 정확하게 한 개의 화소를 선택해야 한다는 것이다. 이 화소가 어떤 선분에 대한 것이라는 사실은 중요

[11] 예를 들어, 많아야 한 개의 화소가 차이가 나는 정도라면




(a) $x$-우선 선분 (b) $y$-우선 선분

그림 4.13: 기울기에 따른 선분의 분류

하지 않고, 다만 두 선분에 대하여 모두 선택이 되거나 둘 모두에 의해 선택이 되지 않는 경우를 피해야 한다. 위에서 설명한 바와 같이 'diamond-exit' 규칙에 의하면 선분이 래스터화 되었을 때 반쪽이 열린(half-open) 상태이어야 하는데, 바로 이러한 제약을 만족시키기 위한 것이다. 따라서 그림 4.12의 점 $p_1$에서 다른 선분이 뻗어 나가면 이 점이 있는 화소는 두 번째 선분에 의하여 선택이 된다. 이 규칙은 기하 프리미티브들을 화면에 일관되게 그려주기 위한 것이다. 만약 같은 화소가 서로 연결된 두 개의 선분에 의하여 동시에 선택이 되면, 예를 들어, 애니메이션을 할 때 그 지점의 색깔이 일관되지 않고 번갈아 가며 두 선분의 색깔이 나타날 수 있는데 그 결과 눈에 거슬리는 문제가 발생한다. 또한 만약 화소가 두 개의 선분 어느 것에도 선택이 되지 않으면, 그 결과 마치 두 선분이 떨어져 있는 것처럼 보이게 되므로, 이 또한 피해야할 상황이다. 지금까지 선분을 래스터화하는데 적용이 되는 규칙에 대하여 살펴보았는데, 이러한 기본적인 원리는 다각형을 래스터화할 때에도 적용이 된다.

### 3.4.2 브레즌햄 알고리즘

이 절에서는 가장 대표적인 선분의 래스터화 방법인 브레즌햄 알고리즘(Bresenham's algorithm)에 대하여 구체적으로 살펴보도록 하겠다. 여기서는 설명의 편의상 다음과 같은 가정을 하겠다. 첫째로 OpenGL에서와는 달리 정수 격자점은 화소의 왼쪽 아래 모서리가 아니라, 화소의 중앙을 나타낸다. 또한 선분의 양 끝점 $p_0 = (x_0, y_0)$과 $p_1 = (x_1, y_1)$는 정수 격자 점으로, 이는 주어진 선분이 두 화소의 중앙을 연결함을 의미한다. 또한 기울기가 0과 1 사이의 값을 가지고, $x_0 < x_1$이고 $y_0 < y_1$인 $x$-우선 선분만을 고려하려하는데, 다른 경우의 선분도 여기서 설명할 알고리즘을 약간만 고치면 쉽게 사용할 수가 있다.

정수 좌표를 가지는 선분에 대하여 $dx \equiv x_1 - x_0$, 그리고 $dy \equiv y_1 - y_0$라 하면, 이 두 점에 의하여 정의되는 직선 L은 절편 값 $b$에 대하여 $y = \frac{dy}{dx} \cdot x + b$와 같이 표현할 수 있다(그림 4.14).

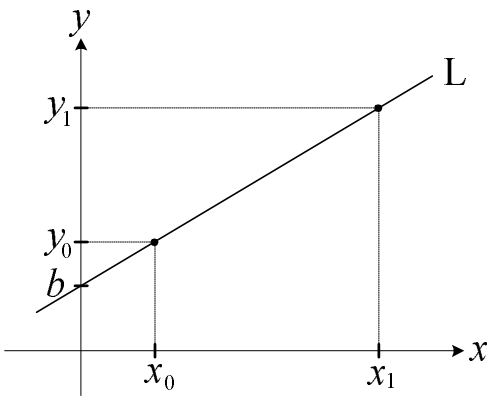



그림 4.14: 선분의 정의

사실 선분을 래스터화 하는 것은 개념적으로는 단순한데, 아래에 주어진 함수를 사용하면 직선 L을 쉽게 래스터화를 할 수 있다.

```
void draw_line_DDA(int x0, int y0, int x1, int y1) {
  float y = y0;
  float slope = (y1 - y0)/(float) (x1 - x0);

  int x;
  for (x = x0; x < x1; x++) {
    generate_fragment(x, round(y));
```

```
      y += slope;
    }
  }
```

이 방법의 수행 과정을 살펴보면, 처리하려는 선분이 $x$-우선이므로($dx \geq dy$), $x = x_0$부터 시작하여 한 칸씩 오른쪽으로 이동하면서 현재의 $y$ 값을 구하여(`y += slope;`), 가까운 정수 값으로 반올림하면 해당 $x$에 대한 화소를 선택할 수 있다. 이 때 $x$ 값이 $x_1 - 1$까지만 증가하는 것은 선분의 한 쪽 끝을 열린 상태로 해주기 위한 것이다.

이러한 방법을 DDA(Digital Differential Analyzer) 알고리즘(DDA algorithm)이라 부른다. 이 방법은 상당히 간결한데, 문제는 정수 연산이 아니라 `float` 타입의 부동 소수점 연산을 사용한다는 점이다. 앞에서도 간략하게 설명한 바와 같이 부동 소수점 연산은 아직도 정수 덧셈/뺄셈 연산과 비교하여 비용이 높은 연산이다. 따라서 정수 덧셈/뺄셈 연산에 기반한 알고리즘을 사용하여 래스터화를 수행할 수 있다면 결과적으로 렌더링 비용을 줄일 수 있다. 물론 위 함수의 회전문 안의 코드를 정수 덧셈/뺄셈만을 사용하는 코드로 대치를 해도, 그 코드가 한두 번만 수행이 된다면 별 차이가 없다. 그러나 래스터화 코드는 방대한 회수로 반복적으로 수행이 되므로, 이는 래스터화 과정의 성능에 큰 차이를 낳게 될 것이다.

브레즌햄 알고리즘의 가장 큰 특징은 정수 덧셈/뺄셈만 사용하여 선분을 래스터화 한다는 점이다. 특히 이 알고리즘이 개발이 되었던 60년대에는 부동 소수점 연산이 정수 연산에 비하여 지금보다 훨씬 더 부담이 되었기 때문에, 이 방법은 당시 매우 중요한 방법으로 사용이 되었으며, 또한 지금도 이에 기반한 알고리즘들이 널리 쓰이고 있다.

사실 두 정수 격자 점을 연결하는 직선 $y = \frac{dy}{dx}x + b$의 기울기와 절편은 유리수로 표현이 가능하기 때문에, 실수의 일부분인 유리수 연산만 사용하여 래스터화를 진

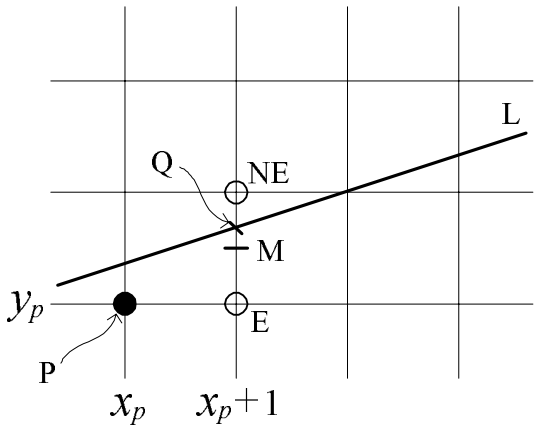



그림 4.15: 브레즌햄 방법

행할 수 있다. 물론 일반적인 CPU는 유리수만 효율적으로 다룰 수 있는 연산을 제공하지는 않는데, 유리수란 한 정수를 0이 아닌 또 다른 정수로 나누어 만든 숫자이기 때문에 정수 연산만을 사용하여 유리수를 효과적으로 다룰 수 있다. 곧 알게 되겠지만 이 직선의 식에서 절편 $b$는 크게 고려를 할 필요가 없고, 유리수인 기울기 값을 정수로 바꾸어 주기 위하여 양변에 $dx$를 곱하여 정리하면 이 직선은 다음과 같이 표현할 수 있다.

$$f(x,y) \equiv dy \cdot x - dx \cdot y + dx \cdot b = 0$$

이는 직선을 표현하는 또 다른 방식으로서, 앞의 방법을 양함수(explicit function) 형태로, 그리고 이 방법을 음함수(implicit function) 형태로 표현을 하였다고 한다. 후자의 장점은 2차원 공간의 임의의 점이 주어졌을 때, 직선에 대한 점의 상대적인 위치를 쉽게 결정할 수 있다는 점이다. 점의 좌표 $(x,y)$를 $f(x,y)$에 대입을 하였을 때, 그 값이 0이면 이 점은 바로 직선 상에 위치하게 된다. 한편 이 직선의 경우 그 값이 양수이면 그 점은 직선의 아래쪽에, 반대로 음수이면 위쪽에 위치함을 의미한다.

그림 4.15는 선분 L을 래스터화하고 있는 과정을 보여주고 있는데, 지금 막 $x = x_p$인 열에 대하여 화소 $(x_p, y_p)$를 선택하였다고 가정하자. 다음 $x = x_p + 1$인 열에 대하여 화소를 결정해야 하는데, 이 그림에서 보듯이 E(East)라는 이름을 붙인 화소와 NE(Northeast)라고 이름을 붙인 화소 두 개중 하나를 선택하려 한다. 여기서는 직선 L이 과연 $y$값이 $y_p$와 $y_p + 1$의 중간에 해당하는 점 M의 위를 지나가는지, 아니면 아래로 지나가는 지만 알면 쉽게 선택을 할 수가 있다. 즉 직선 L과 $x = x_p + 1$의 교점 Q가 M의 위에 있으면 화소 NE를 선택하고 아니면 화소 E를 선택하면 된다. 이를 위하여 M의 좌표 $(x_p + 1, y_p + \frac{1}{2})$을 음함수 형태의 직선의 식에 대입하면 다음과 같다.

$$d_{cur} \equiv f(x_p + 1, y_p + \frac{1}{2}) = dy \cdot (x_p + 1) - dx \cdot (y_p + \frac{1}{2}) + dx \cdot b$$

여기서의 $d_{cur}$를 선택 변수(decision variable)라고 하는데, 앞에서도 말했듯이 다음 화소에 대하여 $d_{cur}$가 양수이면 화소 NE를 선택하고 음수이면 화소 E를 선택하는데, 만약 이 값이 0이면 둘 중 어느 화소를 선택해도 무방하다. 그런데 여기서의 문제는 매번 $x$값이 1씩 증가할 때마다 어떻게 하면 정수 덧셈과 뺄셈만을 사용하여 선택 변수의 값을 계산할 것인가 인데, 바로 이 점이 브레즌햄 방법의 핵심이라 할 수 있다.

만약 지금 화소 E가 선택이 되었다고 가정하면, 화소 $(x_p + 1, y_p)$가 선택이 된 것이므로 다음 선택 변수 $d_{next}$는 점 $(x_p + 2, y_p + \frac{1}{2})$에 대하여 계산을 해야 하는데, 그 값은 다음과 같이 정리할 수 있다.

$$d_{next} \quad = \quad f(x_p + 2, y_p + \frac{1}{2}) = dy \cdot (x_p + 2) - dx \cdot (y_p + \frac{1}{2}) + dx \cdot b$$

$$
\begin{aligned}
&= dy \cdot (x_p + 1) - dx \cdot (y_p + \frac{1}{2}) + dx \cdot b + dy \\
&= d_{cur} + dy
\end{aligned}
$$

즉 현재의 선택 변수에 $dy$만 더하면 다음 선택 변수를 계산할 수 있다. 또한 만약 화소 NE가 선택이 되었다면 같은 방식으로 새로운 선택 변수는 다음과 같이 구할 수 있음을 알 수 있다.

$$
d_{next} = f(x_p + 2, y_p + \frac{3}{2}) = d_{cur} + (dy - dx)
$$

따라서 $\delta_E \equiv dy$, 그리고 $\delta_{NE} \equiv dy - dx$라 정의하면, 다음 선택 변수는 해당하는 경우에 따라 이들 중 하나를 선택하여 더하기만 하면 된다. 이 두 값은 모두 정수 값이므로 미리 그 값을 구해 놓으면, 매번 정수 덧셈 한 번만으로 다음 선택 변수 값을 구할 수 있다.

$x$ 값이 1씩 증가함에 따라 선택 변수는 점진적으로 계산을 하지만, 선택 변수의 초기 값, 즉 화소 $(x_0, y_0)$에 대한 선택 변수를 명시적으로 계산해주어야 하는데 이는 다음과 같이 구할 수 있다.

$$
\begin{aligned}
d_{start} &= f(x_0 + 1, y_0 + \frac{1}{2}) = dy \cdot (x_0 + 1) - dx \cdot (y_0 + \frac{1}{2}) + dx \cdot b \\
&= dy \cdot x_0 - dx \cdot y_0 + dx \cdot b + dy - \frac{dx}{2} \\
&= f(x_0, y_0) + dy - \frac{dx}{2} \\
&= dy - \frac{dx}{2}
\end{aligned}
$$

따라서 최초의 선택 변수는 $d_{start} \equiv dy - \frac{dx}{2}$가 되는데 여기에는 약간 문제가 있어

보인다. 지금의 최대 관심사는 정수 덧셈/뺄셈만을 사용하는 것인데, 이 값을 계산하기 위해서는 정수 나눗셈 연산을 수행해야 한다. 나눗셈은 네 가지의 기본 연산 중 가장 피해야 하는 연산이다. 물론 2의 제곱 값으로 어떤 정수를 나누는 것은 비교적 쉽게 할 수 있지만, 선분을 표현하는 방법을 조금만 바꾸면 최초 선택 변수를 계산할 때의 나눗셈을 피할 수 있다.

앞에서 유리수인 기울기 $\frac{dy}{dx}$에 대하여 정수 연산만 사용하기 위하여 직선의 식의 양변에 $dx$를 곱했었다. 여기서는 분모에 나타나는 2를 없애려면 다시 직선의 식 $f(x,y) = 0$의 양변에 2를 곱하기만 하면 된다. 따라서 원래의 직선의 식 대신에 $F(x,y) \equiv 2 \cdot f(x,y) = 2dy \cdot x - 2dx \cdot y + 2dx \cdot b = 0$을 사용하면 최초 선택 변수를 비롯한 전 선택 변수를 정수 덧셈/뺄셈만 사용하여 계산할 수 있다. 단 사용하는 모든 값들에 대하여 두 배를 해야하므로, $d_{start} = 2dy - dx$, $\delta_E = 2dy$, 그리고 $\delta_{NE} = 2(dy - dx)$를 사용하면 된다. 지금까지 설명한 브레즌햄 알고리즘을 요약하면 다음과 같은데, 이 방법은 유리수 연산을 정수 연산만을 사용하여 효과적으로 수행하는 좋은 예라 할 수 있다.

```
void draw_line_Bresenham(int x0, int y0, int x1, int y1) {
  int x, y = y0, dx = x1 - x0, dy = y1 - y0;
  int d, de, dne;

  de = dy + dy; d = de - dx; dne = d - dx;
  for (x = x0; x < x1; x++) {
    generate_fragment(x, y);
    if (d <= 0) { d += de; }
    else { y++; d += dne; }
  }
}
```

### 3.4.3 연관된 데이터의 선형 보간

다시 한번 강조를 하면 래스터화 과정에서는 두 가지의 중요한 계산이 수행이 되어야 하는데, 하나는 기하 프리미티브가 투영되는 지역의 화소의 위치 $(x_{wd}, y_{wd})$를 찾는 것이고, 다른 하나는 꼭지점에 연관되어 있는 깊이 정보 $z_{wd}$, 색깔 $(r, g, b, a)$, 그리고 텍스춰 좌표 $(s, t, r, q)$로부터 각 화소에 해당하는 데이터를 계산하여 프래그먼트를 형성하는 것이다. 앞에서 살펴본 DDA 알고리즘이나 브레즌햄 알고리즘은 주어진 선분에 대하여 화소의 위치를 찾는 방법들인데, 그러한 알고리즘을 조금만 수정하면 선형 보간을 통하여 화소에 연관시킬 데이터를 계산할 수 있다.

DDA 알고리즘을 보면, 이 방법은 $x$축 방향으로 균일한 거리만큼 한 화소씩 움직이면서 증분 `slope`를 더함으로써, 점진적으로 화소의 위치를 찾고 있다. 즉 DDA 알고리즘은 점진적인 계산에 기반한 전형적인 알고리즘이라 하겠다. 또한 브레즌햄 알고리즘도 변형된 형태의 점진적 계산 방법이라 할 수가 있다. 래스터화 과정에서 어떤 알고리즘을 사용하건 꼭지점에 연관된 데이터에 대해서도 점진적인 계산에 기반한 선형 보간을 통하여, 선분 내부의 화소에 대한 데이터 값을 계산할 수가 있다.

OpenGL에서는 기본적으로 앞에서 알아본 선형 보간을 사용하여 프래그먼트를 구성하는 데이터를 구하도록 되어 있다. OpenGL에서 규정하는 방식을 보면, 양 끝점의 윈도우 좌표를 각각 $p_0$과 $p_1$이라 하고 현재 처리하려는 프래그먼트의 화소 중심의 좌표를 $p$라 하면, 매개 변수 $t$는 다음과 같이 계산을 한다.

$$t = \frac{(p - p_0) \cdot (p_1 - p_0)}{|| p_1 - p_0 ||^2}$$

다음 양 끝점에 연관된 데이터를 $f_0$과 $f_1$이라고 하면 $f = (1 - t) \cdot f_0 + t \cdot f_1$과 같은

식을 사용하여 선형 보간을 할 수가 있다. 물론 이 방법은 이론적인 측면에서의 선형 보간이고 래스터화 코드는 점진적 계산을 통하여 효율적으로 구현해야 한다. 아래의 코드는 여러 데이터 중 무부호 문자 타입의 R 채널 값 $r$에 대하여, 브레즌햄 알고리즘을 확장하는 예를 보여주고 있다.

```
void draw_line_Bresenham(int x0, int y0, int x, int y1,
                         unsigned char r0, unsigned char r1) {
  int x, y = y0, dx = x1 - x0, dy = y1 - y0;
  int d, de, dne;
  int rr0 = IntToFixed(r0);
  int dr = IntToFixed(r1) - rr0;


  de = dy + dy; d = de - dx; dne = d - dx; dr = dr/dx;
  for (x = x0; x < x1; x++) {
    generate_fragment(x, y, FixedToUns(rr0));
    if (d <= 0) { d += de; }
    else { y++; d += dne; }
    rr0 += dr;
  }
}
```

여기서 한 가지 주의해야 할 것은 점진적인 방식으로 각 화소에 대한 선형 보간을 할 때, 일반적으로 화소의 중심이 양 끝점을 연결하는 직선 상에 있지 않기 때문에 이에 대한 고려를 해주어야 한다는 사실이다. 일반적인 선형 보간에서는 보간 값을 계산하려는 지점은 데이터가 주어진 양 끝점과 동일한 직선 상에 존재한다고 가정한다. 하지만 래스터화는 이산 공간인 래스터 공간에서 수행이 되기 때문에 그러한 조건이 항상 만족이 되지는 않는다. 이러한 문제를 해결하는 방법 중의 하나는 화소의 중심과 실제로 정확한 보간 지점인 선분 상의 점간의 거리를 계산하여 선형 보간을 한 값에 대하여 적절한 보정을 해주는 것이다. 물론 이러한 계산을 하려면 부동 소수점 연산을 해주어야 하므로 계산 비용이 증가한다. 이러한 문제는 다각형

을 래스터화 할 때 더 복잡하게 나타나는데, 3.6.4절 끝 부분에서 다시 한번 간략하게 언급을 하도록 하겠다.

## 3.5 다각형의 래스터화

### 3.5.1 점진적인 삼각형의 래스터화

마지막으로 다각형의 래스터화에 대하여 살펴보자. 만약 오목 다각형을 포함한 임의의 형태의 다각형을 래스터화 하려 한다면 여러 가지 상황을 고려하여야하기 때문에 알고리즘이 약간 복잡해진다. OpenGL에서는 마이크로 소프트의 윈도우스나 X 윈도우스 시스템과는 달리 다각형은 볼록하다고 가정을 하므로 상대적으로 간단한 래스터화 알고리즘을 사용할 수가 있다. 임의의 볼록 다각형은 여러 개의 삼각형으로 분해를 할 수 있기 때문에, 다각형 중 가장 단순한 형태를 가지는 삼각형에 대한 래스터화 알고리즘만 효과적으로 구현을 하면 된다. 이 절에서는 삼각형의 래스터화에 대하여 간략하게 살펴보도록 하겠다.

다각형의 래스터화 알고리즘들은 공통적으로 스캔 라인 변환(scan line conversion) 형태의 방식을 취한다. 즉 다각형이 투영되는 영역과 교차하는 스캔 라인들을 위에서 아래로(또는 아래서 위로) 하나씩 '스캔'하면서 각 스캔 라인과 다각형의 변과의 교차 지점을 구하여, 현재의 스캔 라인상의 다각형 내부에 해당하는 화소를 찾는다. 한 스캔 라인상의 다각형 내부 영역을 수평 스팬(horizontal span)이라 하는데, 삼각형의 경우 현재 처리하고 있는 스캔 라인이 다각형의 변 중 어느 변과 교차하는지를 결정하는데 있어 고려해야 할 경우의 수가 적기 때문에 래스터화 알고리즘이 비교적 단순하다.

그림 4.16은 현재 스캔 라인 $l$에 대한 수평 스팬을 처리하고 있는 모습을 보여주고 있는데, 여기서도 편의상 화면의 정수 격자 점이 화소의 중앙을 나타낸다고 가

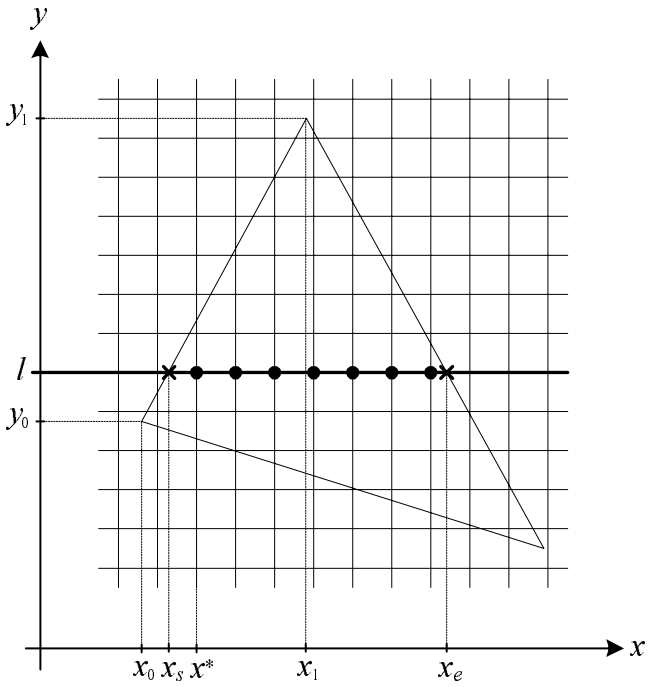



그림 4.16: 수평 스팬의 계산

정하자. 각 스캔 라인에 대하여 삼각형의 두 변과의 교차점의 $x$ 좌표인 $x_s$와 $x_e$를 계산하는 것이 삼각형 래스터화 계산의 핵심이라 할 수 있는데, 이는 $y$축 방향에 대한 점진적인 계산을 통하여 효율적으로 수행을 할 수가 있다. 예를 들어 왼쪽 변의 경우, 양 끝점의 좌표 $(x_0, y_0)$와 $(x_1, y_1)$에 대하여 증분 $\delta_x = -\frac{x_1 - x_0}{y_1 - y_0}$를 미리 계산을 해놓고, $y$축을 따라 위에서 아래로 한 스캔 라인씩 내려올 때마다 바로 직전의 스캔 라인의 시작 위치 $x_s$에 증분을 더하면 현재 스캔 라인에서의 시작 위치를 구할 수 있다. 이런 식으로 현재 처리하려하는 수평 스팬의 양 끝점에 대한 $x_s$와 $x_e$를 구했을 때 바로 그 안에 있는 정수 격자 점들이 삼각형 내부의 화소가 된다.

선분의 경우와 마찬가지로 인접한 삼각형들을 화면에 그려줄 때 적절하게 래스터화를 하지 못할 경우 상당히 눈에 거슬리는 현상이 나타날 수 있다. 예를 들어 사용하는 래스터화 알고리즘에 따라 경계 부분의 한 화소가 두 삼각형 모두에 의하여 선택이 되거나 반대로 두 삼각형 모두에 의하여 선택이 되지 않을 수가 있다. 그 결

과 물체 표면에 흠집이 난 것처럼 보이기도 하고, 또한 텍스춰 매핑을 할 때 텍스춰가 자연스럽게 연결이 안 되어 보이기도 한다. 한두 개의 화소라 무시할 수 있는 것이라고 생각할 수도 있겠지만, 사실은 그러한 결과가 의외로 심각한 현상을 초래하고는 한다. 특히 인접하는 두 삼각형의 변과 스캔 라인이 정수 격자 점에서 교차할 때, 이 화소가 어떤 삼각형에 속하게 할 것인가 하는 것이 중요한 문제인데, 어떤 방식을 사용을 하건 경계 지역의 화소는 일관성 있게 한 개의 삼각형에만 속하도록 해주어야 한다.

수평 스팬의 양 끝 좌표 $x_s$와 $x_e$를 계산을 하였을 때 스팬 내부의 화소를 결정하기 위하여 보편적으로 쓰이는 방법을 소개하면 다음과 같다.

- 왼쪽 변에 대해서는 $x$ 좌표가 $x_s$보다 같거나 큰 화소들이 선택된다. 다시 말해서 $\lceil x_s \rceil$가 시작 화소의 $x$ 좌표가 된다.
- 오른쪽 변에 대해서는 $x$ 좌표가 $x_e$보다 작은 화소들이 선택된다. 다시 말해서 $x_e$가 정수가 아니라면 $\lfloor x_e \rfloor$, 그리고 정수라면 $x_e - 1$이 마지막 화소의 $x$ 좌표가 된다.
- 만약 시작 화소의 위치가 마지막 화소의 위치보다 크다면 해당 스캔 라인에 대해서는 어떤 화소도 선택이 되지 않는다.
- 삼각형이 주어졌을 때 처리할 스캔 라인들에 대한 범위를 결정할 때에도 $y$축 방향에 대하여 비슷한 선택 기준을 적용한다. 즉, 삼각형의 꼭지점 중 가장 위쪽에 있는 꼭지점의 $y$ 좌표보다 같거나 작은 $y$값에 대한 스캔 라인들을 선택하고, 가장 아래쪽에 있는 꼭지점의 $y$ 좌표보다 큰 $y$값에 대한 스캔 라인들을 선택한다.

일반적으로 다각형의 변과 스캔 라인의 교차 위치를 계산하기 위해서는 부동 소

수점 숫자를 사용해야 하기 때문에, 다각형을 래스터화 할 때는 브레즌햄 알고리즘과 같이 정수 연산만으로는 만족할 만한 결과를 얻기가 힘들다. 다만 앞에서 설명한 고정 소수점 연산을 사용한다면 비교적 좋은 정확도와 빠른 래스터화 계산을 수행할 수 있는 여지가 있는데, 어떠한 연산 체계를 사용할 지는 시스템의 개발자가 적절하게 결정을 해야 할 것이다.

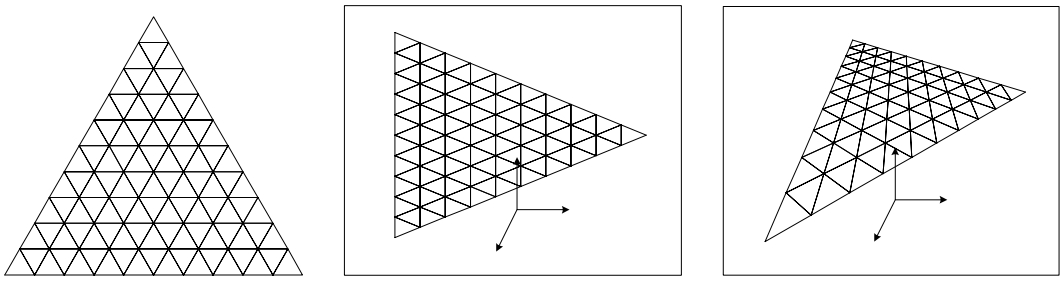
삼각형의 경우에도 선분과 마찬가지로 꼭지점에 연관된 각 데이터들을 사용하여 내부의 화소에 대하여 선형 보간을 해주어야 한다. 기본적으로 수평 스팬을 기준으로 그림 4.6(c)에 도시된 바와 같은 방식을 취하는데, 선분의 래스터화 시 설명한 점진적인 방법을 쉽게 확장할 수가 있다. 이에 대해서는 3.6.4절에서 원근 교정을 통한 삼각형의 래스터화 과정에 대하여 설명할 때 다시 한번 고려를 하겠다.

## 3.6 선형 보간과 원근 교정

### 3.6.1 원근 투영시의 선형 보간의 문제점

지금까지 래스터화 과정에서 꼭지점에 연관된 데이터를 보간할 때 사용하는 단순한 형태의 점진적 선형 보간 방법에 대하여 알아보았다. 사실 이와 같은 선형 보간은 기하 변환을 위하여 평행 투영을 사용할 때에는 아무런 문제가 없으나, 원근 투영을 사용할 경우에 심각한 문제가 발생하게 된다.

그림 4.17(a)는 정삼각형의 내부를 균일한 간격으로 잘게 나눈 모양을 보여주는데, 이는 삼각형 내부에서 점이 균일한 속도로 변하는 모습이라 생각할 수 있다. 그림 4.17(b)와 (c)는 이 정삼각형을 각각 평행 투영인 직교 투영과 원근 투영을 사용하여 렌더링한 모습을 보여주고 있다. 평행 투영을 한 경우에는 윈도우 좌표계에서도 삼각형이 균일한 간격으로 나누어져 있음을 알 수 있다. 반면 원근 투영을 한 결과를 보면 간격의 균일성이 유지가 되지 않음을 알 수 있다. 즉 윈도우 좌표계로 투



(a) 균일하게 세분한 정삼각형 (b) 직교 투영 결과 (c) 원근 투영 결과

그림 4.17: 직교 투영과 원근 투영의 차이

영된 삼각형 내부에서는 점이 더 이상 균일한 속도로 변하지 않게 된다.

그러한 이유는 다음과 같다. 평행 투영을 사용할 때에는 물체 좌표계에서 윈도우 좌표계까지의 모든 기하 변환은 아핀 변환이다. 앞에서도 강조한 바와 같이 아핀 변환의 가장 큰 특징은 평행성의 유지와 간격 비율의 유지인데, 그러한 성질에 의하여 물체 좌표계에서 점의 변화 속도가 일정할 경우 윈도우 좌표계에서도 일정한 속도를 가지게 된다. 거꾸로 말하면 윈도우 좌표계에서 균일한 속도로 움직이면, 물체 좌표계에서 대응이 되는 점도 균일한 속도로 움직이게 된다. 반면 원근 투영을 사용하면 전체 기하 변환이 비아핀 변환이 되므로, 점의 변화 속도가 삼각형 내부의 위치에 따라 변하게 되는 것이다.

그러면 이러한 관계가 꼭지점에 연관된 데이터의 보간에 어떠한 영향을 미칠까? 사실 기하 프리미티브의 꼭지점에는 여러 종류의 데이터를 연관시킬 수가 있지만, OpenGL에서는 깊이 정보, 색깔, 텍스춰 좌표가 연관된다고 하였다. 깊이 정보는 약간 다르게 취급이 될 수 있기 때문에 색깔과 텍스춰 좌표에 대하여 알아보자. 윈도우 좌표계에서 색깔에 대하여 선형 보간을 하는 것은 스므드 쉐이딩을 하기 위한 것이다. 조명 계산은 눈 좌표계에서 수행이 된다고 하였는데, 계산의 효율을 위

하여 기하 물체의 꼭지점에 대해서만 쉐이딩이 된 색깔을 계산하고, 다각형 내부의 점에 대해서는 선형 보간을 하여 얻은 색깔을 사용한다. 여기서의 선형 보간은 물체가 존재하는 물체 좌표계나 물체 좌표계로부터 아핀 변환을 통하여 얻어지는, 따라서 크기와 비율이 유지가 되는, 눈 좌표계를 기준으로 수행하는 것이 자연스럽다고 할 수 있다. 즉 이 경우 색깔 데이터는 삼각형 내부에서 물체 좌표계나 눈 좌표계의 좌표에 대하여 선형적으로 변한다. 만약 눈 좌표계에서 삼각형 내부의 점이 균일한 속도로 변할 때, 그에 대응이 되는 윈도우 좌표계에서의 점 또한 균일한 속도로 변한다면, 꼭지점에 연관된 데이터를 윈도우 좌표계상에서 선형 보간을 해도 결과는 같다. 하지만 평행 투영과는 달리 원근 투영은 이러한 성질을 만족시키지 못하기 때문에 윈도우 좌표계에서 선형 보간을 한다면 삼각형 내부에서 잘못된, 즉 공간이 왜곡된 결과를 얻게 된다.

그림 4.18: 정확하게 텍스춰 매핑이 된 모습

텍스춰 좌표의 경우 5장에서 살펴보겠지만 물체 좌표계인 모델링 좌표계나 세상 좌표계상의 꼭지점에 텍스춰 좌표를 붙여주고, 선형 보간을 통하여 삼각형 내부의 점의 텍스춰 좌표를 구한다. 일반적으로 모델링 좌표계, 세상 좌표계, 그리고 눈 좌표계간의 변환은 아핀 변환이기 때문에 꼭지점에 연관된 텍스춰 좌표를 어느 좌표계에서 보간을 하건 그 결과는 같다. 반면에 원근 투영을 하면 눈 좌표계에서 정규 디바이스 좌표계로의 변환이 아핀 변환이 아니므로, 그 이후의 좌표계, 특히 윈도우 좌표계에서 선형 보간을 할 경우 역시 문제가 발생하게 된다. 그림 4.18은 윈도

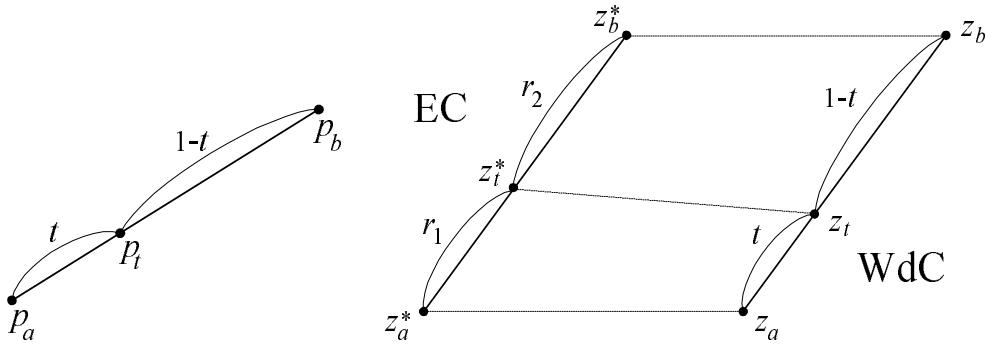
우 좌표계가 아니라 모델링 좌표계에서의 선형 보간을 통하여 텍스춰 좌표를 구한 후 렌더링을 한 예를 보여주는데, 정규적인 패턴을 가지는 체커 보드 텍스춰가 정확하고 원근감이 있게 붙여진 것을 알 수가 있다. 만약 윈도우 좌표계에서 텍스춰 좌표를 보간하였다면 텍스춰 패턴이 균일하게 나타나 원근감을 상실한 부자연스러운 모습을 보이게 될 것이다.

대부분의 실시간 렌더링 시스템에서와 같이 OpenGL에서도 윈도우 좌표계에서의 래스터화 과정에서 선형 보간 계산을 수행한다. 즉 계산의 효율을 위하여 일단 윈도우 좌표계까지 온 후 래스터화를 하면서 동시에 데이터를 보간하는데, 원근 투영을 사용할 때에는 적절한 처리를 해주어야만 자연스러운 렌더링 결과를 얻을 수 있다. 즉 이 경우 윈도우 좌표계가 아니라 물체 좌표계 또는 눈 좌표계상에서 선형 보간이 되도록 해주어야 하는데, 이를 위해서 래스터화 과정에서 변형된 형태의 선형 보간을 사용해야 한다. 이렇게 래스터화 과정에서 선형 보간 시 원근 투영의 문제를 처리해주는 것을 원근 교정(perspective correction)이라 한다.

보간을 하려하는 데이터인 깊이 정보, 색깔, 텍스춰 좌표 중 원근 교정을 하지 않을 경우 가장 심각한 문제가 발생하는 것은 텍스춰 좌표이다. 텍스춰 매핑은 가상현실이나 3차원 게임과 같은 실시간 그래픽스 소프트웨어를 개발하는데 있어 핵심적인 역할을 한다. 또한 이들은 가상의 세상을 렌더링 하기 위한 것이기 때문에 현실감을 생성하기 위하여 반드시 원근 투영을 사용하여야 한다. 이러한 이유로 원근 교정의 정확한 이해와 구현은 3차원 게임 엔진 등과 같은 실시간 렌더러를 개발하는데 있어 가장 중요한 요소 중의 하나가 됨은 분명하다.

### 3.6.2 원근 교정 식의 유도

원근 투영을 할 때의 문제는 근본적으로 원근 변환을 통하여 눈 좌표계에서 정규




(a) 윈도우 좌표계에서의 보간

(b) $z$ 좌표의 대응

그림 4.19: 원근 교정을 통한 선형 보간

디바이스 좌표계로 변환이 되면서 파생되는 공간의 왜곡으로부터 출발한다(157쪽의 그림 2.35 참조). 눈 좌표계에서 한 점이 카메라의 앞쪽으로, 즉 음의 $z_e$축 방향으로, 일정한 속도로 움직인다고 할 때, 정규 디바이스 좌표계에서 이 점에 대응되는 점의 속도는 공간의 찌그러짐에 따라 변하게 된다. 정규 디바이스 좌표계와 윈도우 좌표계간의 변환인 뷰폿 변환은 아핀 변환이므로 역시 윈도우 좌표계에서도 점의 변화 속도가 같은 방식으로 변하게 된다.

래스터화 과정에서는 윈도우 좌표계 상에서의 비율로 선형 보간을 하지만, 원근 투영을 고려하여 정확하게 보간을 하려면 눈 좌표계에서의 비율로 선형 보간을 해야 한다. 그림 4.19(a)에 도시된 바와 같이 윈도우 좌표계에서 양 끝점의 좌표가 $p_a = (x_a\ y_a\ z_a)^t$와 $p_b = (x_b\ y_b\ z_b)^t$인 선분을 생각하자. 윈도우 좌표계는 2차원 좌표계이고 $z_{nd}$ 값은 연관된 데이터로 간주가 되지만, $x_{nd}$, $y_{nd}$ 값들과 함께 3차원 공간에서의 점의 좌표를 구성한다고 볼 수 있는데, 이 공간에서는 $x_{nd}$, $y_{nd}$, $z_{nd}$ 값들이 선분을 따라 서로 선형적으로 변하게 된다. 이 두 점에 연관된 데이터를 $f_a$와 $f_b$라 할 때 선분상의 한 점 $p_t = (x_t\ y_t\ z_t)^t$에 연관될 데이터를 계산하려 한다. 우선 편의상 $z_a \neq z_b$라고 가정하자(그림 4.19(b)). 이는 눈 좌표계에서 카메라를 기준으로 하

여 이 선분을 따라 깊이가 변함을 의미한다. $p_t$는 이 선분을 두 개의 선분으로 나누는데, 윈도우 좌표계에서의 그 길이의 비율은 다음과 같이 된다.

$$\frac{z_b - z_t}{z_t - z_a} = \frac{1-t}{t} \tag{4.1}$$

여기서 명심해야 할 것은 올바른 선형 보간을 원한다면 이 비율 $t : 1-t$로 $f_a$와 $f_b$를 선형 보간을 하는 것이 아니라, 이 세 점에 대응되는 눈 좌표계 공간에서의 점들에 의해 결정되는 두 선분 간의 비율 $r_1 : r_2$를 사용하여 연관된 데이터를 보간해야 한다는 점이다. 따라서 이 비율을 구하기 위하여 눈 좌표계에서 정규 디바이스 좌표계를 거쳐 윈도우 좌표계까지 가는 기하 변환에 대하여 다시 생각해보자. 152쪽에서 유도한 바와 같이 눈 좌표계와 정규 디바이스 좌표계의 $z$ 좌표간에는 다음과 같은 관계가 성립한다.

$$z_{nd} = \alpha + \frac{\beta}{z_e}$$

한편 윈도우 좌표계상의 $z_{wd}$는 $z_{wd} = \gamma + \delta \cdot z_{nd}$와 같이 정규 디바이스 좌표계의 $z_{nd}$ 좌표에 대한 일차식으로 표현이 되므로, $z_{wd}$와 $z_e$간에는 상수 $\alpha'$와 $\beta'$에 대하여 다음과 같은 매핑이 존재하게 된다.

$$z_{wd} = \alpha' + \frac{\beta'}{z_e}$$

$z_a^*$, $z_b^*$, $z_t^*$를 눈 좌표계에서 $p_a$, $p_b$, $p_t$에 대응되는 점의 $z_e$ 좌표라 하면, 위의 관계를

적용하여 다음과 같은 식을 얻을 수 있다.

$$\begin{aligned} z_t - z_a &= (\alpha' + \frac{\beta'}{z_t^*}) - (\alpha' + \frac{\beta'}{z_a^*}) = \beta' \frac{z_a^* - z_t^*}{z_t^* \cdot z_a^*} \\ z_b - z_t &= (\alpha' + \frac{\beta'}{z_b^*}) - (\alpha' + \frac{\beta'}{z_t^*}) = \beta' \frac{z_t^* - z_b^*}{z_b^* \cdot z_t^*} \end{aligned}$$

이 두 식을 식 (4.1)에 대입하여 정리하면 다음과 같은 결과를 얻는다.

$$\frac{z_t^* - z_b^*}{z_a^* - z_t^*} = \frac{(1-t) \cdot z_b^*}{t \cdot z_a^*} \tag{4.2}$$

여기서 눈 좌표계에서 정규 디바이스 좌표계로의 투영 변환 $M_P$에 대하여 다시 생각해보면, 동차 좌표계인 절단 좌표계에서 대응되는 점의 $w_c$ 좌표는 바로 눈 좌표계에서의 $z_e$ 값의 부호를 바꾼 값임을 알 수 있다(156쪽의 식 (2.8) 참조). 따라서 양 끝점의 $w_c$ 좌표를 각각 $w_a$, $w_b$라 하면 $w_a = -z_a^*$, $w_b = -z_b^*$와 같은 관계가 존재하고, 이를 식 (4.2)에 대입하면 다음과 같은 식을 얻는다.

$$\frac{z_t^* - z_b^*}{z_a^* - z_t^*} = \frac{\frac{1-t}{w_a}}{\frac{t}{w_b}} \tag{4.3}$$

눈 좌표계에서 $z_e$ 좌표는 선분을 따라 일정한 속도로, 즉 선형적으로, 변하므로 이 좌표계에서의 두 선분간의 비율 $r_1 : r_2$는 $\frac{t}{w_b} : \frac{1-t}{w_a}$가 됨을 알 수 있다. 따라서 래스터화 과정에서 윈도우 좌표계상에서의 비율 $t : 1-t$가 주어졌을 때 원근 투영을 고려한 선형 보간은 다음과 같이 수행할 수 있다.

$$f_t = \frac{\frac{1-t}{w_a}}{\frac{1-t}{w_a} + \frac{t}{w_b}} \cdot f_a + \frac{\frac{t}{w_b}}{\frac{1-t}{w_a} + \frac{t}{w_b}} \cdot f_b$$

$$= \frac{\frac{1-t}{w_a} \cdot f_a + \frac{t}{w_b} \cdot f_b}{\frac{1-t}{w_a} + \frac{t}{w_b}}$$

지금까지는 $z_a \neq z_b$인 경우에 대해서만 고려를 하였다. 만약 $z_a = z_b$이라면 이는 위에서 고려한 세 점이 모두 눈 좌표계에서 $z_e$축에 수직인 동일한 평면 위에 있음을 의미한다. 앞에서의 원근 변환 행렬의 유도 과정을 다시 보면, 같은 평면 위에 있는 점들은 공간의 찌그러짐에 의한 영향을 받지 않음을 알 수 있다. 원근 나눗셈을 할 때 동일한 평면 위에 있는 점들에 대해서는 단지 $w_c$값에 따라 $x_c$와 $y_c$ 좌표에 대하여 균일하게 크기 변환만 되므로, 윈도우 좌표계에서와 같은 비율이 유지가 된다. 따라서 이 때는 단순히 $t : 1 - t$ 비율로 선형 보간을 해도 되는데, 두 점 $p_a$와 $p_b$에 대한 $w_c$값 $w_a$와 $w_b$가 서로 같음($w_a = w_b$)을 생각하면 역시 이 경우에도 위의 식이 성립함을 알 수 있다.

지금까지 유도한 방식의 원근 교정을 통하여 올바르게 선형 보간을 하려면 기하 물체들을 구성하는 점들이 윈도우 좌표계로 변환이 될 때 절단 좌표계에서의 $w_c$ 좌표 값을 기억을 해야 한다. 물론 평행 투영을 할 경우에는 $w_c$값이 1이므로 위의 식은 두 종류의 투영 변환 모두에 대하여 정확하게 선형 보간을 하게 된다. 이 식은 다음과 같이 표현할 수 있는데, 이를 통하여 원근 교정 계산의 직관적인 의미와 효율적인 계산 방법에 대하여 살펴보도록 하자.

$$f_t = \frac{(1-t) \cdot \frac{f_a}{w_a} + t \cdot \frac{f_b}{w_b}}{(1-t) \cdot \frac{1}{w_a} + t \cdot \frac{1}{w_b}} \tag{4.4}$$

### 3.6.3 원근 교정 식의 직관적인 의미

앞 절에서는 원근 교정을 통한 선형 보간에 관한 수식을 유도하였는데, 이 절에서는 다른 관점에서 이에 대한 직관적인 의미를 살펴보자. 앞에서도 언급한 바와 같

이 원근 투영을 사용할 때의 문제는 눈 좌표계에서 선형적이었던 정보가 원근 투영 후 윈도우 좌표계에서는 더 이상 선형적이지 않다는데 있다. 선형성(linearity)이라는 성질에 대하여 다시 고려하여 보자. $p$와 $q$라는 변수 사이에 선형 관계가 존재한다는 것은 두 변수를 상수 $a$와 $b$에 대하여 $q = a \cdot p + b$와 같이 표현할 수 있음을 의미한다. 따라서 두 변수의 관계를 직선 형태의 그래프로 표현할 수 있는데, 선형적이라는 것은 곧 직선적인, 다시 말해서 평면적인(planar) 관계를 의미한다. 두 변수간에 선형적인 관계가 존재할 때의 가장 큰 특징중의 하나는 한 변수를 다른 변수로 미분을 할 경우 미분 값이 상수가 된다는 점인데, 이는 한 변수가 일정한 속도로 변하면 다른 변수도 역시 일정한(일반적으로 속도는 다르겠지만) 속도로 변함을 의미한다.

같은 방식으로 생각하면 $n$개의 변수 $p_1, p_2, \cdots, p_n$과 $m$개의 변수 $q_1, q_2, \cdots, q_m$사이에 선형적인 관계가 존재한다는 것은 두 집단의 변수간에 관계를 $m$행 $n$렬의 행렬 $A$와 $m$차원 벡터 $b$를 사용하여 다음과 같이 표현할 수 있음을 뜻한다.

$$\begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_m \end{pmatrix} = A \cdot \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix} + b$$

이와 같은 다변수의 경우에도 한 그룹의 변수를 다른 그룹의 변수에 대하여 편미분을 할 경우 그 미분 값은 역시 상수가 된다. 컴퓨터 그래픽스에서 가장 많이 쓰이는 아핀 변환도 바로 이렇게 변환 전과 후의 좌표 값 사이에 선형적인 관계가 있는 단순한 변환인 것이다. 1차 미분 값, 즉 기울기 정보가 상수 값을 가지면, 한 그룹의 변수 값이 단위 길이만큼씩 점진적으로 변할 때, 이에 대응되는 다른 그룹의 변수

값은 단순히 매번 해당 기울기 값을 더해가면서 효율적으로 구할 수가 있다.

래스터화 과정에서 선형 보간 계산이 수행될 때의 정보는 크게 두 가지 그룹으로 나눌 수가 있다. 우선 한 그룹에는 윈도우 좌표계의 좌표 값 $x_{wd}$와 $y_{wd}$가 속하고, 다른 하나에는 보간을 하려는 연관된 데이터, 즉 깊이 정보 $z_{wd}$, 색깔 $(r, g, b, a)$, 텍스춰 좌표 $(s, t, r, q)$ 등이 속하게 된다. 앞에서 보간 계산은 윈도우 좌표계상의 화소를 기준으로 하여 수행이 된다고 하였다. 즉 $y_{wd}$축 방향으로 한 화소씩 움직이면서 각 스캔 라인 상에서의 수평 스팬의 양 끝점에서의 데이터 값을 보간하여, 이 값들에 대하여 $x_{wd}$축 방향으로 한 화소씩 움직이면서 점진적으로 선형 보간을 하게 된다. 따라서 윈도우 좌표계에서의 래스터화 계산 과정을 보면, 이 좌표계에서는 $x_{wd}$축, $y_{wd}$축 각 방향으로 일정한 속도로(각 방향으로 값을 1씩 증가시키면서) 점진적으로 움직이고 있음을 알 수 있다. 만약 위의 두 그룹의 값들 사이에 선형적인 관계가 존재한다면 윈도우 좌표계 상에서 일정한 속도로 움직이면서 화소를 스캔할 때 단순히 기울기 정보를 사용하여 연관된 데이터를 보간할 수 있다. 그러나 평행 투영의 경우와는 달리 원근 투영의 경우에는 두 그룹의 변수사이에 선형적인 관계가 존재하지 않기 때문에 원근 교정을 해주어야 한다.

원근 투영 시의 문제는 위의 두 그룹의 값들 사이에 비선형적인 관계가 존재하기 때문에, 지금 설명한 바와 같은 선형적인 값들간의 단순한 계산 방식을 적용할 수가 없다는 점이다. 과연 원근 투영의 경우 선형 관계의 장점을 이용할 수 있는 방법은 없을까? 원근 투영에서 사용되는 기하 변환을 잘 살펴보면 다음과 같은 관계가 존재함을 발견할 수 있다.

1. 절단 좌표계의 네 번째 좌표 값 $w_c$에 대하여, $\frac{1}{w_c}$과 윈도우 좌표계의 좌표 값 $x_{wd}$와 $y_{wd}$사이에는 각각 선형적인 관계가 존재한다.

2. 보간을 하려하는 두 번째 그룹의 임의의 값을 $f$라 하면[12], $\frac{f}{w_c}$와, $x_{wd}$와 $y_{wd}$사이에 각각 선형적인 관계가 존재한다.

이러한 관계 중 $\frac{1}{w_c}$와 $x_{wd}$사이에 선형적인 관계가 있음을 간략하게 증명해 보자. 154쪽에서 설명한 바와 같이 상수 $\alpha$에 대하여 $x_{nd} = \frac{\alpha \cdot x_e}{-z_e}$와 같은 관계가 존재한다. $x_{nd}$와 $x_{wd}$사이에 상수 $\beta$와 $\gamma$에 대하여 $x_{nd} = \beta \cdot x_{wd} + \gamma$와 같은 선형적인 관계가 있고(164쪽의 식 (2.10) 참조), $w_c = -z_e$인 점을 고려하면 다음과 같은 식이 성립한다.

$$\beta \cdot x_{wd} + \gamma = \frac{\alpha \cdot x_e}{w_c}$$

또한 기하 물체를 표현하기 위하여 선형적인 다면체 모델을 사용하기 때문에 각 다각형 내부에서의 눈 좌표계상의 좌표 값 $x_e$, $y_e$, $z_e$들간에는 서로 선형적인 관계가 존재한다. 다시 말해서 다각형의 점들은 동일한 평면 위에 있고, 평면 상의 점 $(x\ y\ z)^t$의 좌표 값간에는 $a \cdot x + b \cdot y + c \cdot z + d = 0$과 같은 관계가 존재하므로, 각 좌표 값 사이에는 선형적인 관계가 존재한다. 특히 $x_e$와 $z_e$사이에 선형적인 관계가 존재하고 $w_c = -z_e$이므로, $x_e$와 $w_c$사이에는 상수 $\beta'$과 $\gamma'$에 대하여 $x_e = \beta' \cdot w_c + \gamma'$과 같이 표현이 가능하고, 따라서 이를 위의 식에 대입하면 다음과 같이 정리할 수 있다.

$$x_{wd} = \frac{\alpha \cdot \gamma'}{\beta} \cdot \frac{1}{w_c} + \frac{\alpha \cdot \beta' - \gamma}{\beta}$$

[12] 즉 $f$는 $z_{wd}$, $r$, $g$, $b$, $a$, $s$, $t$, $r$, $q$ 등을 의미한다. $z_{wd}$는 렌더링 파이프라인의 뒷 부분에서 깊이 버퍼링에 사용이 되는데, 이 때 중요한 것은 점들간의 전후 관계이므로 이에 대해서는 굳이 원근 교정을 할 필요가 없다. 또한 원근 투영시의 선형 보간으로 인한 문제는 색깔을 보간할 때보다 텍스춰 좌표를 보간할 때 더 눈에 뜨이게 된다.

따라서 $x_{wd}$와 $\frac{1}{w_c}$사이에 선형적인 관계가 성립을 함을 알 수 있는데, 나머지 관계들도 비슷한 방식으로 증명할 수 있다. □

이제 이러한 두 사실을 고려하면 $\frac{1}{w_c}$과 $\frac{f}{w_c}$는 $x_{wd}$와 $y_{wd}$의 윈도우 좌표계 공간에서 선형적으로 변하게 됨을 알 수 있고, 따라서 이 값들에 대해서는 그러한 선형성을 바탕으로 하여 점진적인 선형 보간이 가능해진다. 래스터화 과정에서 선분을 따라 선형 보간을 할 때, $t : 1-t$의 비율에 대하여 두 꼭지점 $p_a$와 $p_b$에 연관된 $f_a$와 $f_b$ 값을 사용하여 직접 보간을 하는 것이 아니라, 1. 두 꼭지점에 해당하는 $\frac{1}{w_a}$와 $\frac{1}{w_b}$, 그리고 $\frac{f_a}{w_a}$와 $\frac{f_b}{w_b}$를 각각 계산하여, 2. 윈도우 좌표계상에서의 비율 $t : 1-t$를 사용하여 이 값들을 점진적으로 선형 보간을 한 후, 3. $\frac{f_t}{w_t}$에 해당하는 값 $(1-t)\cdot\frac{f_a}{w_a} + t\cdot\frac{f_b}{w_b}$를 $\frac{1}{w_t}$에 해당하는 값 $(1-t)\cdot\frac{1}{w_a} + t\cdot\frac{1}{w_b}$ 으로 나누면 $f_t = \frac{\frac{f_t}{w_t}}{\frac{1}{w_t}}$과 같이 원근 교정을 한 보간 값을 구할 수 있다. 바로 이것이 식 (4.4)에 대한 하나의 직관적인 해석이라 할 수 있다.

### 3.6.4 원근 교정의 효율적인 계산

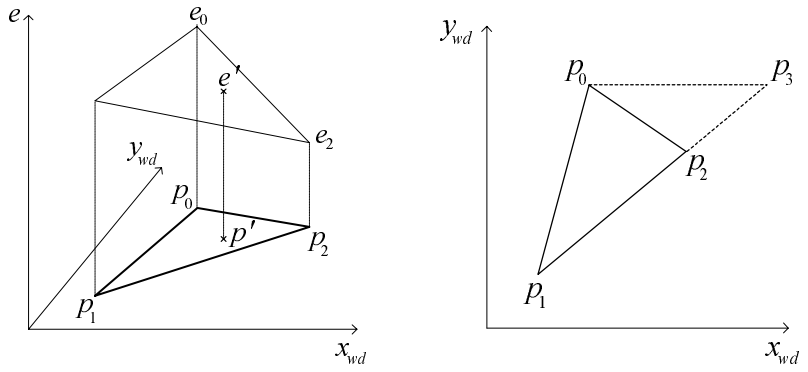
이제 문제는 어떻게 하면 위의 보간 식을 래스터화 과정에서의 점진적인 계산 틀에 끼워 넣을 것인가 하는 것이다. 앞에서 설명한 삼각형의 래스터화 과정을 다시 생각해보자. 삼각형이 걸치는 스캔 라인들을 아래에서 위로 하나씩 순서대로 따라가면서, 각 수평 스캔의 끝점에서의 보간 값을 구하기 위하여 삼각형의 해당 변을 따라 꼭지점에 연관된 데이터에 대하여 선형 보간을 하고, 다음 이 값들을 사용하여 스캔 라인상의 화소를 따라 같은 계산을 점진적으로 반복하면 올바른 선형 보간을 할 수가 있다. 문제는 원근 교정에 기반한 래스터화 알고리즘을 구현할 때 주의를 하지 않으면 매우 비효율적인 결과를 낳을 수가 있다.

한 가지 다행스러운 것은 $\frac{1}{w_c}$와 $\frac{f}{w_c}$가 윈도우 좌표계 상의 삼각형 내부에서 선형

적으로 변한다는 성질을 잘 이용하면 래스터화 과정의 계산량을 많이 줄일 수가 있다는 점이다. 어떤 데이터 $e$가 삼각형 내부에서 선형적으로 변한다고 하면[13], 이 값은 그림 4.20(a)에서와 같이 윈도우 좌표계 공간의 변수 $x_{wd}$와 $y_{wd}$에 대한 1차식의 함수로 표현할 수 있다. 즉 삼각형 내부에서 변하는 데이터 $e$는 $e = e(x_{wd}, y_{wd})$와 같이 윈도우 좌표계 상에서의 1차 함수로 표현이 가능한데[14], 데이터가 선형적으로 변할 때는 이 그림에서와 같이 함수의 그래프는 평면의 형태를 가지고, 이 때 '기울기'는 일정하게 된다. 즉 함수 $e(x, y)$의 그래디언트 $\nabla e(x, y) = (\frac{\partial e}{\partial x}, \frac{\partial e}{\partial y})$가 상수 값을 가지게 되고, 그 결과 윈도우 좌표계에서 $x$, $y$축 각 방향으로 일정한 속도로 움직일 때 데이터 값도 각각 일정한 속도로 변하게 된다. 따라서 래스터화 계산을 할 때 매 스캔 라인에 대하여 증분을 매번 계산할 필요가 없이 주어진 삼각형에 대하여 보간을 하려하는 데이터에 대한 그래디언트 값을 한 번만 계산을 한 뒤 필요할 때마다 사용을 하면 된다.

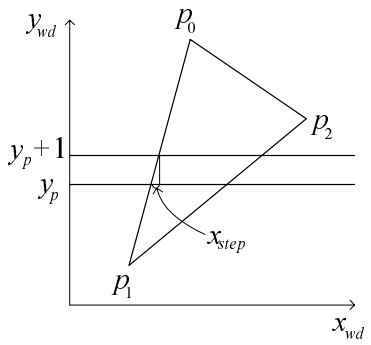
그러면 과연 그래디언트는 어떻게 계산을 할까? $x$축 방향의 미분 값 $\frac{\partial e}{\partial x}$를 구하는 방법을 이해하기 위하여 그림 4.20(b)를 보자. 윈도우 좌표계에서의 삼각형의 세 개의 꼭지점 $p_i = (x_i, y_i)$ $(i = 0, 1, 2)$에는 선형 보간을 하려고 하는 데이터 값 $e_i$가 연관되어 있다. 여기에 또 하나의 점 $p_3 = (x_3, y_3)$이 있는데, $p_3$는 $p_1$과 $p_2$를 연결한 직선과 직선 $y = y_0$가 만나는 점을 나타낸다. 선형 보간을 할 때에는 데이터 $e$가 윈도우 좌표계에서 선형적으로 변한다고 가정을 하는 것이기 때문에, 이러한 상황에서는 $p_3$에 대응되는 데이터 값 $e_3$도 유일하게 정의가 된다. 간단한 비례 관계를 사

[13] 여기서 $e$는 $\frac{1}{w_c}$이거나 보간을 하려하는 $f$에 대한 $\frac{f}{w_c}$ 값이다.
[14] 편의상 이 시점 이후의 변수 $x$와 $y$의 첨자 $wd$를 생략하자.




(a) 선형적인 데이터 $e$

(b) 그래디언트의 계산




(c) 증분의 계산

그림 4.20: 효율적인 래스터화

용하면 $p_3$의 $x$ 좌표 $x_3$는 다음과 같은 식으로 표현할 수 있다.

$$\frac{x_2 - x_1}{x_3 - x_1} = \frac{y_2 - y_1}{y_3 - y_1}$$

또한 $e$값이 선형적으로 변하므로 $e_3$값은 다음과 같은 방식으로 표현이 가능하다.

$$\frac{e_2 - e_1}{e_3 - e_1} = \frac{y_2 - y_1}{y_3 - y_1}$$

여기서 $y_0 = y_3$라는 사실을 이용하면 $x_3$와 $e_3$는 다음과 같이 구할 수가 있다.

$$\begin{aligned} x_3 &= x_1 + \frac{y_0 - y_1}{y_2 - y_1} \cdot (x_2 - x_1) \\ e_3 &= e_1 + \frac{y_0 - y_1}{y_2 - y_1} \cdot (e_2 - e_1) \end{aligned}$$

$x$축 방향으로의 미분 값은 상수로서 $e_3 - e_0$를 $x_3 - x_0$으로 나눈 값이 $x$에 대한 편미분 값 $\frac{\partial e}{\partial x}$이 되는데, 그 값은 다음과 같다.

$$\frac{\partial e}{\partial x} = \frac{(y_0 - y_1)(e_2 - e_1) + (y_2 - y_1)(e_1 - e_0)}{(y_0 - y_1)(x_2 - x_1) + (y_2 - y_1)(x_1 - x_0)} \tag{4.5}$$

같은 방법으로 $y$축 방향으로의 미분 값 $\frac{\partial e}{\partial y}$을 구하면 다음과 같다.

$$\frac{\partial e}{\partial y} = \frac{(x_2 - x_1)(e_0 - e_1) + (x_1 - x_0)(e_2 - e_1)}{(x_2 - x_1)(y_0 - y_1) + (x_1 - x_0)(y_2 - y_1)} \tag{4.6}$$

주어진 삼각형을 래스터화할 때 스캔 라인은 수평 스팬을 구하기 위하여 사용하는 두 개의 선분에 따라 두 그룹으로 나누어진다. 즉 그림 4.20(c)에서와 같은 예에서는 스캔 라인의 왼쪽 끝은 항상 변 $\overline{p_0p_1}$에 존재하지만, 오른쪽 끝은 래스터화 과

정의 전반부에는 변 $\overline{p_1p_2}$에, 그리고 후반부에는 변 $\overline{p_2p_0}$에 있게 된다. 어떠한 경우이건 항상 1. 현재 처리하려는 스캔 라인의 왼쪽 끝점의 위치와 이 점에서의 데이터 값 $e$, 그리고 수평 스캔의 폭을 구한 후, 2. 해당 폭만큼 화소를 차례로 스캔하면서 점진적으로, 즉 각 데이터 $e$에 대한 $\frac{\partial e}{\partial x}$를 더해 선형 보간을 하여, 3. 각 $f$에 대하여 $\frac{f}{w_c}$를 $\frac{1}{w_c}$로 나누어 화소에 대한 정확한 보간 값을 구하면 된다.

이를 위하여 우선 현재 처리하려고 하는 스캔 라인의 왼쪽 끝점의 위치와 데이터 값을 계산하여야 하는데, 이 또한 직전 스캔 라인의 데이터 값들에 적절한 증분을 더해 점진적으로 구할 수 있다. 예를 들어 그림 4.20(c)에서 $y = y_p$인 스캔 라인에서의 처리를 끝내고 다음 스캔 라인 $y = y_p + 1$에서의 래스터화를 수행하려 한다고 하자. 삼각형의 변 $\overline{p_0p_1}$의 좌표를 사용하여 $x_{step} = \frac{x_0 - x_1}{y_0 - y_1}$ 값을 구하면 이는 $y$ 좌표가 1만큼 증가할 때의 $x$값의 변화량을 나타낸다. 따라서 왼쪽 점의 $x$ 좌표는 단순히 직전 값에 $x_{step}$을 더하면 된다. 또한 새로운 스캔 라인으로 갈 때 이는 $x$축 방향으로 $x_{step}$만큼, 그리고 $y$축 방향으로는 1만큼 움직이는 것에 해당하므로 이 때의 증분은 $e_{step} = \frac{\partial e}{\partial x} \cdot x_{step} + \frac{\partial e}{\partial y}$이 된다. 이 값은 각 변에 대하여 상수 값이므로 주어진 삼각형에 대하여 한 번만 구하면 된다. 이렇게 현재 처리하려고 하는 수평 스캔의 왼쪽 끝점에서의 필요한 값들은 적절한 증분을 한 번씩만 더함으로써 효율적으로 선형 보간을 할 수가 있다.

이제 원근 교정을 통한 삼각형의 래스터화 과정을 요약하면 다음과 같다.

1. 삼각형의 세 꼭지점에 대하여 $\frac{1}{w_c}$과 보간을 하려하는 데이터 $f$ 각각에 대하여 $\frac{f}{w_c}$를 구한다.

2. 선형 보간을 하려고 하는 각 데이터 $e = \frac{1}{w_c}, \frac{f}{w_c}$에 대하여 그래디언트 $\frac{\partial e}{\partial x}$와 $\frac{\partial e}{\partial y}$를 구한다.

3. 삼각형의 왼쪽 변에 대하여 $x_{step}$과 $e_{step}$을 구한다.

4. 이렇게 삼각형 래스터화에 필요한 상수 값들을 모두 구한 후 스캔 라인별로 래스터화를 수행한다.

   (a) 현 스캔 라인상의 왼쪽 끝점의 위치와 선형 보간 값은 직전 스캔 라인에서의 보간 값에 $x_{step}$과 $e_{step}$을 더해 점진적으로 구하고, 또한 수평 스캔의 폭을 점진적으로 구한다.

   (b) 현 스캔 라인상의 각 화소에서의 선형 보간 값은 왼쪽 끝점의 데이터에 $\frac{\partial e}{\partial x}$를 계속 더해 점진적으로 구한다.

   (c) 각 화소에 대하여 보간을 하려고 하는 데이터 $f$에 대하여 $\frac{f}{w_c}$를 $\frac{1}{w_c}$로 나누어 정확한 선형 보간 값을 구한다.

래스터화 과정을 구현하기 위해서는 앞에서도 강조한 바와 같이 계산량을 최적화하여야 한다. 위의 계산 과정이 일견 복잡해 보이는데, 이 과정을 소프트웨어적으로 구현을 하건 하드웨어적으로 구현을 하건 간에 효율적으로 구현을 할 수 있도록 노력을 기울여야 한다. 예를 들어 각 데이터 $e$에 대한 그래디언트를 구할 때 식 (4.5)와 (4.6)을 잘 보면 두 식의 분모가 같고, 또한 많은 계산이 중복이 되어 있어 비교적 적은 양의 연산을 수행하여 계산을 할 수 있다. 또한 $\frac{f}{w_c}$를 $\frac{1}{w_c}$로 나눌 때 이를 여러 개의 데이터 $f$에 대하여 수행을 한다면 매번 나누는 것보다 한 번 나눗셈을 통하여 $\frac{1}{\frac{1}{w_c}}$을 구한 후, 각 $\frac{f}{w_c}$에 대하여 이 값들 곱하는 것이 더 빠르게 계산이 될 가능성이 많다.

한 가지 자세히 설명을 하지 않은 것은 화소 중심의 위치는 정수 격자 점(또는 OpenGL에서와 같이 정수 격자 점을 $x$, $y$축 방향으로 0.5만큼 이동한 지점) 위에 있는데, 매 스캔 라인의 왼쪽 끝점의 위치가 대개의 경우 화소의 중심에 떨어지지 않

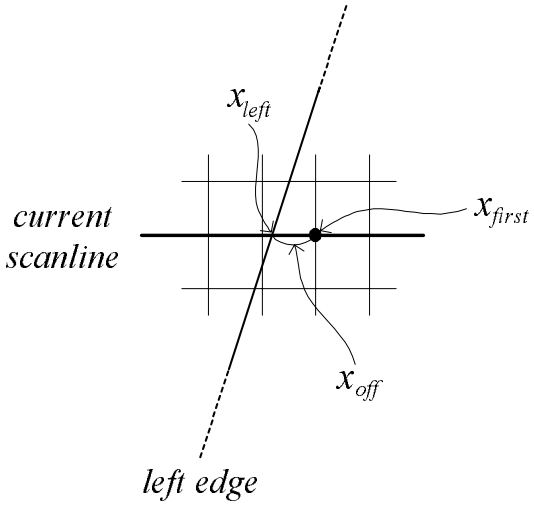



그림 4.21: 첫 번째 화소에 대한 교정

는다는 사실이다. 화소의 중심은 정수 격자 점과 일치한다는 가정 하에 그림 4.21을 보자. 현재 스캔 라인의 왼쪽 끝점의 $x$ 좌표가 $x_{left}$라 할 때 앞에서 설명한 규칙에 의하면 이 스캔 라인의 첫 번째 화소의 위치는 $\lceil x_{left} \rceil$임을 알 수 있다(그림의 $x_{first}$). 문제는 이 화소부터 시작하여 데이터 값 $e$를 점진적으로 계산을 하게 되는데, 변을 따라 선형 보간한 $e$값은 $x = x_{left}$인 점에서의 데이터 값이기 때문에 적절히 교정을 통하여 시작 화소에서의 데이터 값을 구해야 한다. 물론 이는 원래의 데이터 값에 $\frac{\partial e}{\partial x} \cdot (\lceil x_{left} \rceil - x_{left})$만큼의 증분을 더하면 된다. 이러한 초기 데이터 값의 교정 문제는 $y$축 방향에 대해서도 적용이 되는데 이에 대한 자세한 내용은 생략하기로 한다.

# 제 5 장

# 텍스춰 매핑

## 제 1 절  사실적인 렌더링을 위한 매핑 기법

컴퓨터 그래픽스 렌더링에 있어 가장 중요한 목표는 실제의 상황을 카메라로 촬영한 듯한 사실적인 영상을 생성하는 것이다. 다면체 모델을 사용하는 렌더링 시스템에서 물체의 사실성을 높이기 위한 방법 중의 하나는 더 많은 수의 다각형을 사용하여 정교하게 물체를 표현하는 것이다. 하지만 이는 계산 시간과 메모리 사용량에 민감한 실시간 렌더링에 있어 큰 부담이 되기 때문에 좋은 방법이라고는 할 수 없다. 물체를 정확하게 표현하려고 하면 의외로 그에 필요한 다각형의 개수가 빠른 속도로 증가하며, 비교적 복잡한 물체를 표현하려고 하면 쉽게 수만-수십만 개의 다각형이 필요하고는 한다. 따라서 컴퓨터 그래픽스의 기하 모델링 분야에서는 가능한 한 적은 개수의 다각형으로 물체를 사실적으로 표현하려하는 기법에 대하여 활발한 연구를 수행하여왔다.

반면에 비교적 적은 추가 비용으로 이미지의 사실성을 상당히 높일 수 있는 방법 중의 하나는 다면체 모델을 렌더링 할 때 미리 생성해 놓은 다양한 부류의 이미

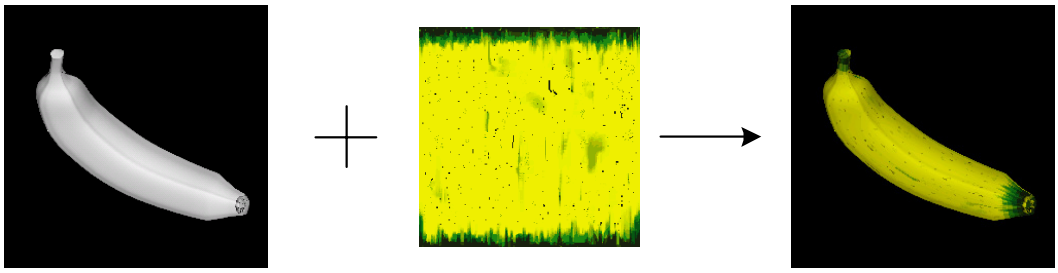



그림 5.1: 2차원 텍스춰 매핑의 예

지 데이터를 적절하게 적용하여 이미지를 생성하는 것이다. 그러한 방법의 대표적인 것이 텍스춰 매핑(texture mapping)인데, 우리가 가장 보편적으로 접하는 텍스춰 매핑 기법은 2차원 텍스춰 매핑(two-dimensional texture mapping)이라 할 수 있다. 이 방법에서는 물체 표면의 텍스춰, 즉 색깔, 패턴, 형태 등을 표현해주는 2차원 이미지를 미리 생성해 놓고, 이를 아주 얇은 고무판이라 생각하고 다면체 물체에 덮어 씌운 후 렌더링을 하여, 물체 표면에 마치 그러한 텍스춰가 존재하는 것처럼 보이도록 하는 기법이다.

그림 5.1은 다면체 모델에 대하여 스므드 쉐이딩을 사용하여 렌더링을 한 바나나에 텍스춰를 붙여 사실감을 높인 2차원 텍스춰 매핑의 예를 보여주고 있다. 이와 같이 텍스춰 매핑에 사용하는 이미지를 텍스춰 이미지(texture image) 또는 텍스춰 맵(texture map)이라고 하는데, 텍스춰 정보를 사용하여 물체 표면의 색깔을 적절히 바꾸어 줌을 알 수가 있다. 텍스춰 이미지도 하나의 래스터 이미지인데, 일반 이미지의 기본 요소를 화소라고 하는 반면 텍스춰 이미지의 기본 요소는 텍셀(texel)이라 부른다.

렌더링 시 텍스춰 매핑 기법을 적용함으로써 얻을 수 있는 이점 중의 하나는 대부분의 경우 기하 물체를 상세하게 표현할 필요가 없게 된다는 사실이다. 즉 많은

개수의 다각형을 사용하여 기하 물체의 자세한 형태를 표현하는 대신, 그보다 적은 개수의 다각형으로 근사화한 뒤, 텍스춰 이미지 정보를 사용하여 물체의 형태, 색상, 패턴 등을 사실적으로 표현할 수 있는 것이다. 그래픽스 전용 가속기의 급속한 발전에도 불구하고 응용 소프트웨어에서 표현하고자 하는 장면의 복잡도는 더 빠른 속도로 증가하고 있기 때문에, 사용하는 다각형의 개수를 줄이는 것이 무엇보다 중요하다. 따라서 텍스춰 매핑은, 특히 계산 시간이나 메모리 사용량과 같은 렌더링 자원을 최적화해야 하는 실시간 렌더링 소프트웨어 제작에 있어 핵심적인 역할을 한다. 실제로 3차원 비디오 게임과 같은 응용 소프트웨어를 보면 물체를 구성하는 다각형의 개수는 줄이는 대신 다양한 텍스춰를 사용하여 현실감을 높이려는 방식을 취하고 있음을 쉽게 알 수 있다.

위에서 예로 보인 2차원 텍스춰 매핑은 맵 데이터를 사용하여 렌더링 작업의 사실성을 높이려는 다양한 매핑 기법(mapping techniques)중의 하나이다. 매핑 기법은 단순히 실시간 렌더링에서뿐만 아니라 여러 렌더링 방법에서 다양한 방식으로 사용이 되어 왔다. 앞에서는 맵 데이터가 물체의 색깔, 특히 물체 표면의 기반 색깔을 결정하는 난반사 색깔에 영향을 끼쳤는데, 이것이 가장 전형적인 매핑 방법이다. 한편 사용하는 매핑 기법의 목적에 따라 맵 정보가 물체에 영향을 미치는 방법이 달라진다. 예를 들어 환경 매핑(environmental mapping)이라고 하는 방법에서 사용하는 환경 맵(environmental map)은 주로 물체의 정반사 색깔에 영향을 미쳐 마치 물체 표면에 주변 환경이 반사되는 것과 같은 효과를 내준다. 보통 환경 맵은 구맵(sphere map)의 형태를 많이 취하는데, 뒤에서 살펴보겠지만 구맵을 사용하여 정반사 색깔에 영향을 미치게 함으로써, 자연스러운 하이라이트 효과를 생성하기도 한다.

또한 맵 데이터 정보는 쉐이딩 색깔에 영향을 미칠 뿐만 아니라, 물체의 기하 정

보에 영향을 미치기도 한다. 예를 들어, 범프 맵(bump map)을 사용하는 범프 매핑(bump mapping)에서는 쉐이딩 계산을 할 때, 맵 데이터를 사용하여 법선 벡터를 조금씩 변화시킴으로써, 마치 물체 표면이 울퉁불퉁한 질감을 가지는 것처럼 해준다. 실제로 이 기법을 사용할 경우 물체의 기하 형태가 바뀌는 것은 아닌데, Pixar의 렌더맨 소프트웨어 등에서 사용하는 변위 매핑(displacement mapping)에서는 맵 데이터의 내용에 따라 물체의 형상 정보, 즉 물체의 좌표 값을 바꿔 다양한 형태의 렌더링 효과를 내준다. 한편 그림자 매핑(shadow mapping) 기법에서는 그림자 맵(shadow map)이라는 정보를 사용하여 렌더링 할 때 물체의 어느 지점에 그림자가 지는지를 판단하여 그림자 효과를 산출한다.

앞에서도 설명한 바와 같이 렌더맨 소프트웨어와 같은 렌더링 시스템에서는 쉐이딩 언어를 사용하여 렌더링 계산 과정을 자유자재로 프로그래밍을 할 수가 있다. 따라서 그러한 구조를 가지는 렌더링 시스템에서는 다양한 매핑 기법을 적용하기가 수월하다. 한편 OpenGL과 같이 조명 계산 부분이 고정된 구조에서는 내부적으로 OpenGL 파이프라인에 포함이 된 텍스춰 매핑 기능을 제외하고는 기본적인 매핑 기법을 사용하기가 자연스럽지가 않다. 그럼에도 불구하고 점차 OpenGL의 기능이 확대됨에 따라 이러한 매핑 기법을 적용하여 실시간적으로 이미지의 사실성을 높이려는 시도를 하고 있다.

렌더링 계산에 사용되는 고급 매핑 기법들을 상세히 설명하는 것은 이 책의 범위를 벗어나므로, 여기서는 가장 보편적인 매핑 기법, 즉 2차원 텍스춰 맵을 사용하여 기하 물체의 색깔에 영향을 미치는 2차원 텍스춰 매핑을 중심으로 설명하겠다. 특히 실시간 렌더링 문맥에서 다면체 모델에 효과적으로 텍스춰를 입히는 방법에 대하여 알아보고, OpenGL을 사용한 프로그래밍 방법에 대해 설명하도록 하겠다. 또한 비교적 고급 기법이라 할 수 있는 투영 텍스춰 기법과 퐁 쉐이딩에 기반한 하이

라이트 생성 기법에 대하여 살펴보겠다.

## 제 2 절 다면체 모델의 렌더링을 위한 2차원 텍스춰 매핑

2차원 텍스춰 매핑 기법에 대하여 구체적으로 설명하기 전에, 텍스춰 매핑의 개념적인 계산 과정에 대하여 알아보자. 그림 5.2에는 매핑 계산을 하는데 있어 중심이 되는 세 개의 좌표 공간이 도시되어 있다. 우선 기하 물체가 존재하는 가상의 세상인 물체 공간(object space)이 있는데, 이는 OpenGL의 물체 좌표계에 해당한다. 다음 텍스춰 이미지가 정의되는 공간을 텍스춰 공간(texture space)이라 하는데, 이 공간을 텍스춰 좌표계(texture coordinates)라 부르기도 하겠다. 마지막으로 화면에 해당하는 공간인 스크린 공간 (screen space)이 있는데, 이는 OpenGL에서의 윈도우 좌표계에 대응이 된다고 할 수 있다.

앞에서 강조한 바와 같이 2차원 텍스춰 매핑은 2차원 이미지를 3차원 기하 물체에 둘러싸서 마치 그 물체가 텍스춰 이미지가 제공하는 것과 같은 질감을 갖도록 하는 것이다. 따라서 텍스춰 매핑을 하는데 있어 가장 먼저 해야 할 작업은 텍스춰 이미지와 기하 물체간의 대응 관계를 설정하는 것이다. 다시 말해서 텍스춰를 물체에 입힐 때, 물체 표면 상에 존재하는 물체 공간의 점 $(x_o\ y_o\ z_o)^t$에 대응되는 텍스춰 공간의 점 $(x_t\ y_t)^t$을 효과적으로 설정을 해야 한다. 이는 일반적으로 쉬운 문제가 아닌데, 실시간 렌더링에서는 주로 다면체 모델의 꼭지점에 대해서만 대응 관계를 설정한다. 수학적으로 말하면 물체 공간과 텍스춰 공간간의 매핑 함수를 정의하는 것으로서, 이러한 과정을 표면 매개화(*surface parameterization*)라 한다.

다음 뷰잉과 조명 인자들을 결정하면, 앞에서 배운 바와 같이 기하 물체가 기하 변환(*geometric transform*)을 통하여 스크린 공간으로 투영이 된다. 이러한 과정도 수학적으로 보면 물체 공간의 점 $(x_o\ y_o\ z_o)^t$와 스크린 공간의 점 $(x_s\ y_s)^t$간의 매핑

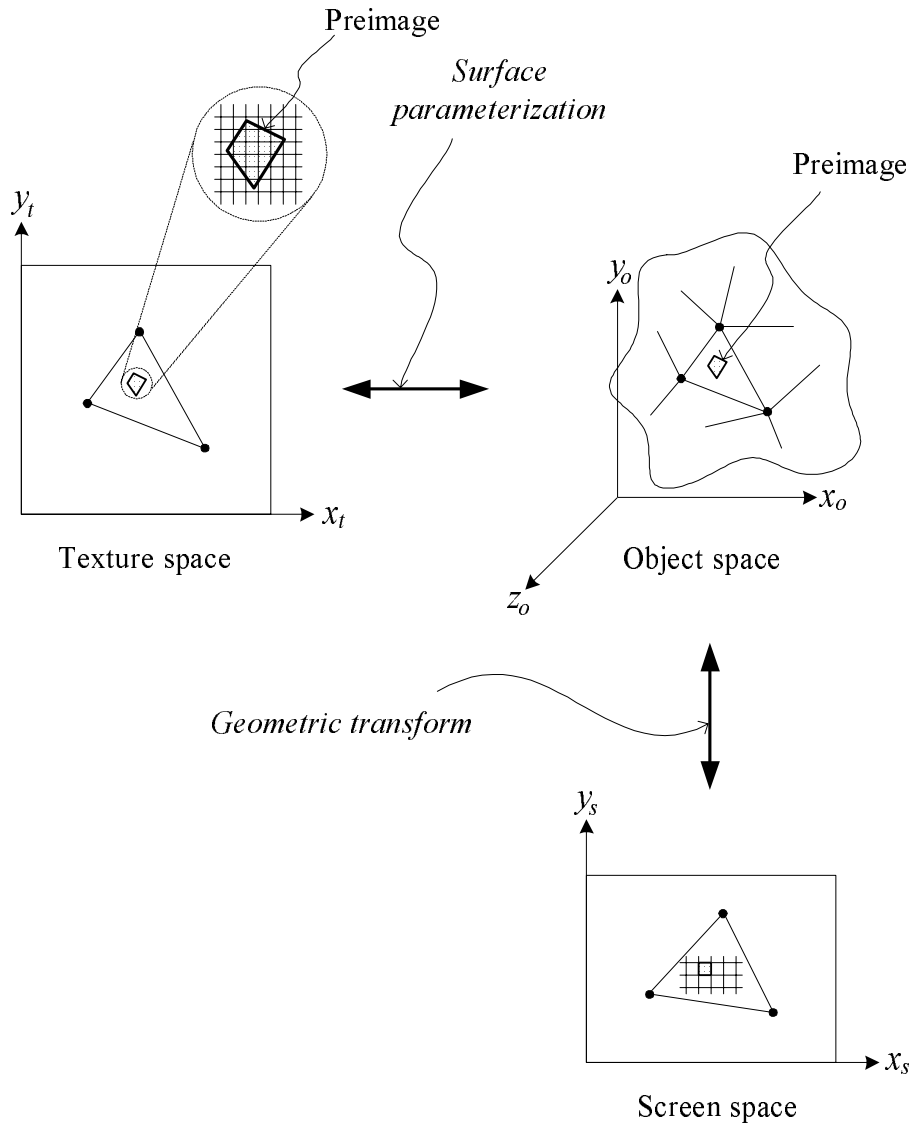



그림 5.2: 개념적인 2차원 텍스춰 매핑 과정

이라 할 수 있다. 일단 물체가 화면으로 투영이 되면, 그것을 구성하는 기하 프리미티브들에 대하여 래스터화 계산이 수행된다. 즉 각 기하 프리미티브가 투영이 되는 화면상의 화소들을 찾아 거기에 칠할 적절한 색깔을 구하게 된다. 텍스춰 매핑 기법을 적용할 때는 바로 이 시점에서 각 화소에 해당하는 텍스춰 색깔을 구해, 물체의 기반 색깔[1]과 혼합을 하여 텍스춰를 입혀준다.

여기서 문제는 각 화소, OpenGL 용어를 사용하면, 각 프래그먼트에 해당하는 텍스춰의 색깔을 어떻게 계산할 것인가 하는 것이다. 화소는 세상을 바라보는 조그마한 창문이라 할 수 있는데, 각 화소마다 그것을 통하여 보이는 물체 표면의 영역을 생각할 수 있다. 물론 이러한 영역을 정확하게, 그리고 효율적으로 계산하는 것은 일반적으로 매우 어려운 일이다. 앞에서 물체 공간에서 스크린 공간으로의 매핑에 대하여 언급했는데, 최소한 개념적으로는 이 함수의 역함수를 통하여 화소에 대응이 되는 물체 표면의 영역을 구할 수 있다. 이 영역은 화소에 해당하는 스크린 공간에서의 사각형 영역에 대한 물체 공간에서의 원상(preimage)인데[2], 이 화소에 대응되는 텍스춰의 색깔은 바로 이 영역에 입혀지는 텍스춰 이미지의 대응 영역의 내용이 된다. 표면 매개화 과정을 수학적인 함수로 볼 때, 역시 개념적으로 그 역함수를 통하여 텍스춰 공간에서의 프리 이미지를 구할 수가 있고, 그것을 통하여 이 화소에 대한 색깔을 계산할 수 있다. 일단 이렇게 텍스춰 색깔을 구하면 화소에 대한 기반 색깔과 원하는 효과에 따라 적절하게 합성을 하여 색깔 버퍼에 칠할 색깔을 얻을 수가 있다. 그림 5.2에서의 한 화소에 대한 물체 공간과 텍스춰 공간에서의 프리 이미지를 다시 한번 보기 바란다.

[1] OpenGL의 경우 눈 좌표계에서 꼭지점에 대하여 결정을 한 후 해당 화소로 래스터화를 한 색깔
[2] 텍스춰 매핑 과정에서의 원상을 그냥 프리 이미지라 부르도록 하겠다.

지금까지 표면 매개화라는 텍스춰 공간과 물체 공간간의 매핑 함수와 기하 변환에 해당하는 물체 공간과 스크린 공간간의 매핑 함수를 통하여 텍스춰 매핑 과정을 개념적으로 살펴보았다. 요약을 하면, 텍스춰 모델링을 통하여 표면 매개화 함수를 정의하고, 뷰잉 인자를 통하여 기하 변환 함수를 결정하면, 물체를 화면에 투영하여 래스터화 계산을 하면서, 물체가 투영이 되는 각 화소에 대한 텍스춰 공간에서의 프리 이미지를 계산을 하여, 그 영역 안의 텍셀 색깔들을 사용하여 텍스춰 색깔을 구한 후, 이를 물체의 기반 색깔과 합성함으로써, 2차원 텍스춰를 입혀주게 된다. 이러한 과정은 개념적으로는 단순하나, 실제로 이를 구현을 하기 위해서는 여러 가지 문제를 효과적으로 해결을 해야하는데, 실시간 렌더링과 밀접하게 관련된 몇 가지 기본적인 문제를 정리하면 다음과 같다.

- [문제 1] 텍스춰 이미지의 내용과 그에 관련된 성질을 어떻게 정의할 것인가? 또한 한 장의 이미지를 렌더링하기 위하여 여러 개의 텍스춰를 사용할 경우 텍스춰 이미지들을 어떻게 하면 효과적으로 다룰 수가 있을 것인가?

- [문제 2] 다면체 모델에 대하여 어떠한 방식으로 2차원 텍스춰를 붙여줄 것인가? 다시 말해서 표면 매개화 과정을 어떻게 정의할 것인가?

- [문제 3] 다면체 모델이 투영이 되는 각 화소에 해당하는 텍스춰의 색깔을 어떻게 구할 것인가?

- [문제 4] 텍스춰 매핑 계산에 영향을 미치는 여러 성질들을 어떻게 정의할 것인가?

- [문제 5] 각 화소에 대하여 텍스춰 색깔과 물체의 기반 색깔을 어떻게 혼합을 할 것인가?

# 제 3 절 예제 프로그램을 통한 OpenGL 텍스춰 매핑의 이해

## 3.1 OpenGL의 텍스춰 매핑 구조와 적용 예

이 절에서는 간단한 예제 프로그램을 통하여 OpenGL에서의 2차원 텍스춰 매핑의 기본적인 사항들을 빠르게 이해를 해보도록 하겠다(예제 프로그램 5.A참조). OpenGL 시스템에서의 텍스춰 매핑 계산의 방식을 이해하기 위하여 378쪽의 그림 4.1을 다시 보자. 텍스춰 매핑 계산은 프리미티브 단위의 데이터를 래스터 형태의 데이터로 변환을 해주는 래스터화 계산 과정 직후에 일어난다. 앞에서도 설명한 바와 같이 각 타입에 따른 래스터화 계산의 결과, OpenGL에서는 해당 프리미티브의 투영 영역에 떨어지는 화소마다 프래그먼트 형태의 래스터들이 생성이 되어, 계속해서 렌더링 파이프라인을 따라 흘러간다.

다시 한번 강조를 하면 프래그먼트는 프리미티브가 투영되는 각 화소에 대하여, 1. 윈도우 좌표계에서의 화소의 위치에 해당하는 정보 $(x_{wd}, y_{wd})$와 그에 연관된 데이터, 즉 2. 이 화소를 통하여 보이는 프리미티브 지점까지의 거리에 해당하는 깊이 값 $z_{wd}$, 그리고 3. 그 지점에 연관된 물체의 기반 색깔 $(r, g, b, a)$와 4. 텍스춰 좌표 $(s, t, r, q)$로 구성이 되어 있다. 그림 5.3을 보면 알 수 있듯이 OpenGL에서의 텍스춰 매핑 과정의 궁극적인 목적은 텍스춰 매핑 관련 인자들의 설정 값에 따라 텍스춰 색깔을 구해 래스터화 과정의 결과로 생성된 프래그먼트의 색깔을 적절히 바꾸어주는 것이다. 그 결과 OpenGL 함수를 통하여 어떠한 방식으로 텍스춰 매핑을 할 것인지를 설정한 후 기하 물체를 그리면, 렌더링 파이프라인의 계산 과정에서 그에 따라 텍스춰가 입혀진다. 물론 텍스춰 매핑을 원하지 않으면 프래그먼트의 색깔은 그대로 보존이 된다.

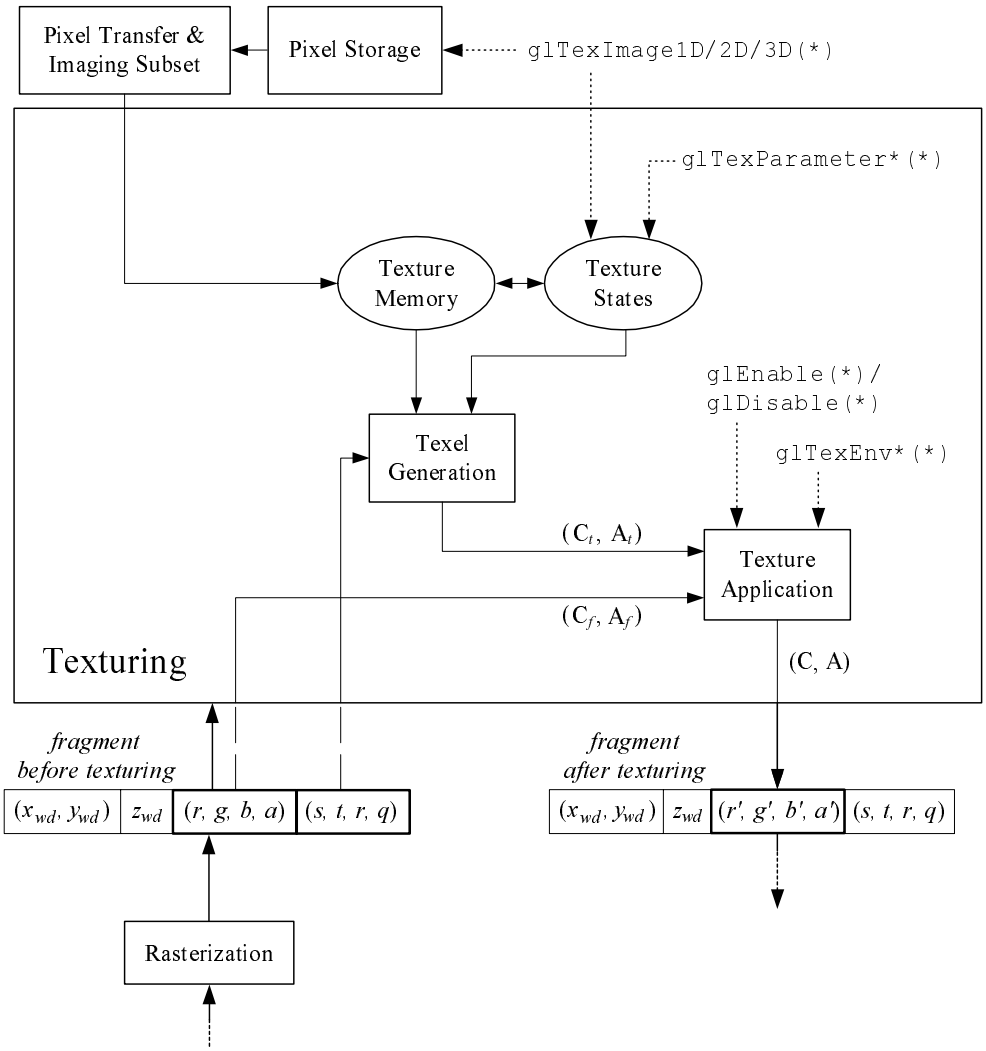
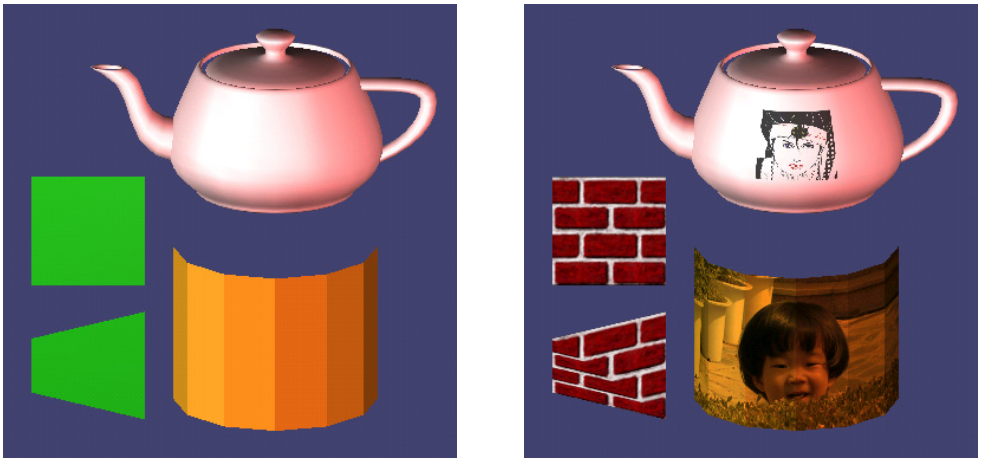



그림 5.3: OpenGL의 텍스쳐 매핑 계산 구조

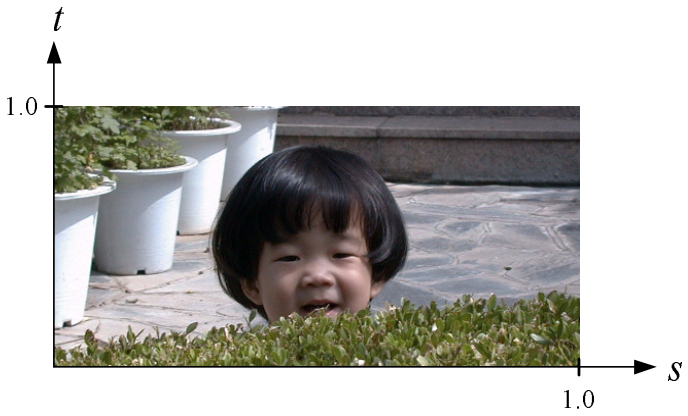


(a) 텍스춰 이미지의 적용 이전 (b) 텍스춰 이미지의 적용 이후

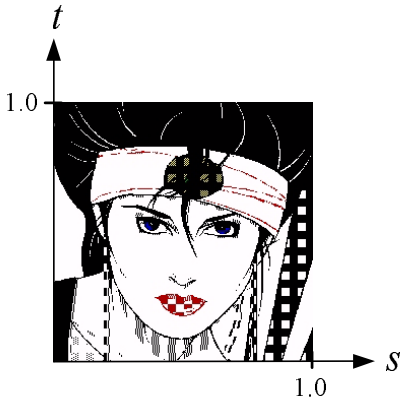
그림 5.4: 간단한 2차원 텍스춰 매핑의 예

텍스춰 매핑 과정에 있어 가장 중요한 두 가지 계산은 그림 5.3의 텍셀 생성(Texel Generation) 모듈에서 수행하는 텍셀의 생성(texel generation)과 텍스춰 적용(Texture Application) 모듈에서 수행하는 텍스춰의 적용(texture application) 계산이라 할 수 있다. 텍셀 생성 과정은 현재 처리하려하는 프래그먼트에 입힐 텍스춰의 색깔(그림 5.3에서의 $(C_t, A_t)$)을 계산하는 것을 목적으로 한다. 한편 그렇게 구한 색깔과 프래그먼트의 기반 색깔 $(C_f, A_f)$이 텍스춰 적용 모듈에서 프로그래머가 설정한 방식으로 적절히 합성이 되어 텍스춰 매핑 효과를 내주게 된다. OpenGL에서의 텍스춰 매핑을 위한 프로그래밍은 바로 이러한 과정을 프로그래머가 원하는 방식으로 수행할 수 있도록 필요한 인자를 설정하는 것이라 할 수 있다.

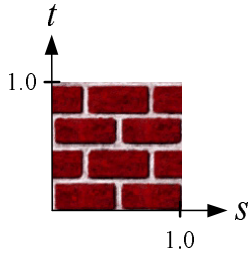
이제 그림 5.4의 두 이미지를 보자. 그림 (a)에는 라이팅 계산을 통하여 스므드 쉐이딩을 사용한 물주전자와 두 개의 사각형, 그리고 플랫 쉐이딩을 사용한 병풍(오른쪽 아래)이 렌더링되어 있다. 여기에 세 개의 서로 다른 텍스춰를 입힌 결과가 그림 (b)에 도시되어 있는데, 지금까지 살펴본 뷰잉과 라이팅 관련 코드에 이러한 텍




(a) 텍스춰 1




(b) 텍스춰 2




(c) 텍스춰 3

그림 5.5: 사용 텍스춰

스춰 매핑 기능을 첨가하기 위하여 추가적으로 필요한 OpenGL 프로그래밍에 대하여 살펴보도록 하자.

그림 5.5에는 여기서 사용한 세 개의 텍스춰 이미지가 주어져 있는데, 해상도는 각각 $512 \times 256$, $256 \times 256$, 그리고 $128 \times 128$이다. OpenGL의 1.2 버전과 기존의 1.1 버전과의 차이점 중의 하나는 1.2 버전에 1차원 및 2차원 텍스춰 매핑 기법 외에 3차원 텍스춰 매핑을 새로운 기본 기능으로 추가하였다는 점이다. 그 결과 1.2 버전 이전의 OpenGL에서는 3차원 텍스춰 매핑이 가능하도록 특별히 기능이 확장

된 OpenGL 1.1 버전을 사용할 경우에만 3차원 텍스춰 매핑을 통한 렌더링을 수행할 수 있었다. 그와는 달리 OpenGL 1.2 버전에서는 3차원 텍스춰 매핑을 다른 기본 기능들처럼 손쉽게 사용할 수 있다. OpenGL에서는 텍스춰 좌표를 표현하기 위하여 3차원 공간의 동차 좌표 $(s\ t\ r\ q)^t$를 사용한다. 이 좌표는 기하 물체를 표현하기 위한 3차원 공간의 점 $(x\ y\ z\ w)^t$에 대응이 되는데, OpenGL에서 2차원 공간의 점 $(x\ y)^t$가 $z = 0$, $w = 1$인 특수한 경우로 취급이 되는 것처럼, 2차원 텍스춰 이미지에 대한 좌표 $(s\ t)^t$는 마찬가지로 $r = 0$, $q = 1$인 특수한 경우로 취급이 된다. 물론 $r$과 $q$ 좌표는 3차원 텍스춰 매핑이나 투영 텍스춰와 같은 고급 기법에 유용하게 쓰이는데, 일단은 2차원 좌표 $(s\ t)^t$를 사용하는 전형적인 2차원 텍스춰 매핑에 대해서만 고려를 하자[3].

텍스춰 이미지가 어떠한 해상도를 갖건 OpenGL의 텍스춰 좌표계에서는 $s$와 $t$ 좌표 각각에 대하여 0과 1 사이의 구간으로 정규화가 된다. 그림 5.6을 보면 그림 5.4의 각 기하 물체의 꼭지점에 설정되어 있는 텍스춰 좌표를 알 수가 있다. 우선 병풍은 여섯 개의 동일한 모양의 직사각형으로 구성이 되어 있는데, $s$ 좌표는 $0, \frac{1}{6}, \frac{2}{6}, \cdots, 1$과 같이 변하며, $t$ 좌표는 0과 1 값을 가진다. 다음 물주전자의 굵게 표시된 영역의 다각형들의 꼭지점은 그림에 주어진 바와 같이 $s$와 $t$ 각 좌표에 대하여 0과 1 사이에서 $\frac{1}{6}$씩 증가하면서 적절히 텍스춰 좌표가 설정이 되어 있다. 반면 그 밖의 꼭지점에 대해서는 모두 $(-1.0, -1.0)$으로 설정되어 있다. 마지막으로 왼쪽 아래의 두 개의 사각형은 실제로 각각 두 개의 삼각형으로 구성이 되어 있는데, 각 꼭지점에 어떻게 텍스춰 좌표가 설정되어 있는지 쉽게 알 수가 있다.

[3] 편의상 문맥에 따라 2차원 텍스춰 좌표를 $(s, t)$ 와 같은 형태로 나타내겠다.

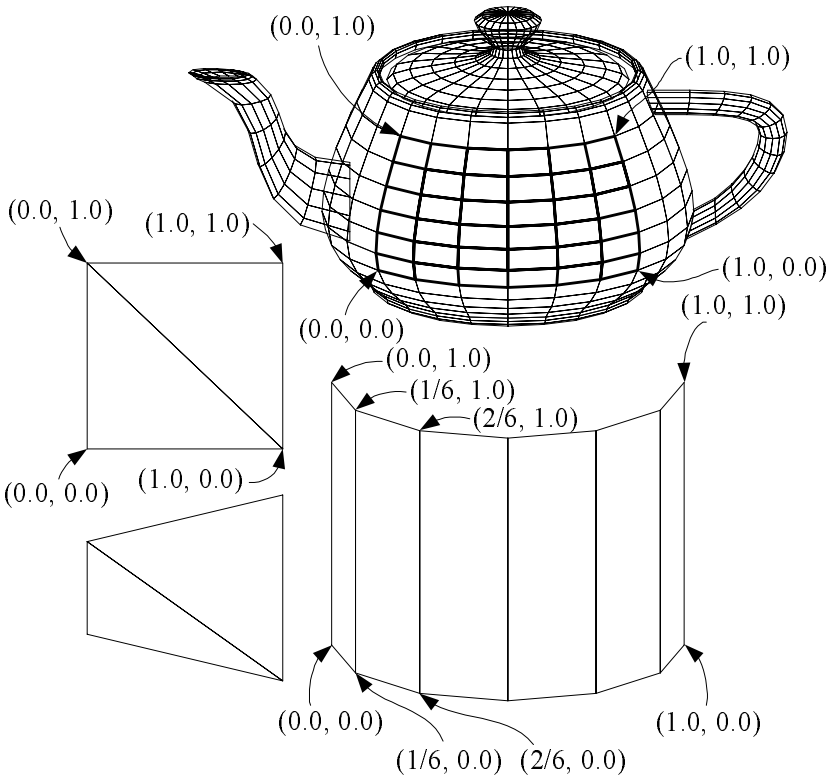



그림 5.6: 텍스춰 좌표의 설정

## 3.2 텍스춰 이미지의 설정과 텍스춰 객체

### 3.2.1 텍스춰 이미지의 설정

텍스춰 매핑과 관련한 프로그래밍을 하기 위해서 가장 먼저 해야 할 작업은 원하는 텍스춰 이미지를 OpenGL 시스템이 사용할 수 있도록 설정을 하는 것이다. 그 결과 텍스춰 이미지의 내용이 OpenGL 시스템에 맞게 해석이 되어 적절히 변환이 되는데, 2차원 텍스춰는 void glTexImage2D(GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *texels); 함수를 사용하여 설정하면 된다. 일반적인 경우 첫 번째 인자 target에 대하여 상수 GL_TEXTURE_2D를 넘기면, 이 함수는 마지막 인자 texels이 가리키는 메모리 영역에 저장이 되어 있는 2차원 이미지 데이터를 텍스춰 메모리에 올려주게 된다. 여기서 네 번째와 다섯 번째 인자 width와 height는 그 이름이 나타내듯이 텍스춰 이미지의 해상도를 설정하는데 사용이 되는데, 이 때 이 두 인자는 항상 2의 멱급수 값, 즉 0보다 같거나 큰 정수 $n$에 대하여 $2^n$ 형태의 값을 가져야 한다. 여섯 번째 인자 border는 0 또는 1 값을 가지는데, 만약 이 값이 1이라면 현재 올리려하는 텍스춰 이미지 둘레로 한 화소의 폭을 가지는 경계가 있음을 의미한다. 따라서 width가 256이고 height가 128, 그리고 border가 1 값을 가질 경우, $(256 + 1) \times (128 + 1)$의 해상도를 가지는 텍스춰를 사용함을 의미하게 된다. 물론 border에 0 값이 설정되면 경계가 없음을 의미하는데, 텍스춰의 경계에 나타나는 색깔은 필요에 따라 추후 필터링 과정에서 사용이 된다.

이 함수의 두 개의 인자 format과 type는 메인 메모리에 저장되어 있는 텍스춰 이미지 데이터의 저장 형식을 나타낸다. 우선 format은 각 텍셀을 구성하는 요소 값들을 나타낸다. OpenGL에서는 GL_RGB, GL_BGR, GL_RGBA, GL_BGRA, GL_RED,

GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA 등의 값을 가질 수가 있다[4]. 어떤 상수 값을 사용하는가에 따라 메인 메모리에 저장된 텍스춰 이미지의 텍셀이 몇 개의 필드로 구성이 되어 있는지, 그리고 각 필드가 어떠한 의미를 내포하는지가 결정된다. 이 때 각 필드의 요소 데이터가 실제로 저장이 되어 있는 타입은 type 인자에 의해 결정이 되는데, 가능한 상수 값으로 GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_BITMAT 등이 있다. 각 상수 값이 의미하는 바는 분명한데, 이 외에도 GL_UNSIGNED_BYTE_3_3_2와 같이 1, 2, 또는 4 바이트에 해당 요소 값들을 축약하여 저장하는 타입을 사용할 수가 있다. 예를 들어 이 두 인자에 대하여 GL_RGBA와 GL_UNSIGNED_BYTE가 설정이 되면, 메인 메모리에 저장되어 있는 텍스춰의 각 텍셀은 각각 한 바이트의 무부호 문자 값으로 표현된 RGBA 색깔로 구성되어 있음을 의미한다.

이렇게 설정이 되는 메인 메모리에 존재하는 텍스춰 데이터는 그림 5.3의 화소 저장(Pixel Storage) 모듈에서 $(R, G, B, A)$ 형태의, OpenGL 시스템 내부에서 사용되는 형식으로 변환이 된다. 즉 format 인자에 대하여 어떤 상수 값을 설정하건, 메인 메모리에 저장되어 있는 각 텍셀 값은 일단 내부적으로 $(R, G, B, A)$ 값으로 변환이 된다. 예를 들어 GL_RED를 사용할 경우, $R$ 채널 값은 메인 메모리에 저장 된 값, $G$와 $B$는 0, 그리고 $A$는 1 값으로 변환이 된다. 다음 화소 이동 및 이미징 서브셋(Pixel Transfer & Imaging Subset) 모듈에서 프로그래머가 설정한 방식으로 적절히 변환이 되어 텍스춰 메모리에 올려진다.

[4] 이 외에도 색깔 인덱스 모드를 위한 GL_COLOR_INDEX의 사용이 가능한데, 이에 대해서는 고려를 하지 말자.

| 내부 형식 | 사용 RGBA 채널 | 대응 내부 요소 |
|---|---|---|
| GL_ALPHA | A | $A$ |
| GL_LUMINANCE | R | $L$ |
| GL_LUMINANCE_ALPHA | R, A | $L$, $A$ |
| GL_INTENSITY | R | $I$ |
| GL_RGB | R, G, B | $R$, $G$, $B$ |
| GL_RGBA | R, G, B, A | $R$, $G$, $B$, $A$ |

표 5.1: RGBA 값으로부터 내부 형식으로의 변환

이 때 텍스춰 메모리에 올려지는 각 텍셀이 실제로 어떠한 형식으로 구성이 되는지를 internalFormat 인자를 통하여 결정할 수 있다. 텍스춰 이미지의 텍셀에 저장될 수 있는 요소 데이터로는 색깔의 R, G, B, 그리고 A 채널 값과 조도(luminance)와 밝기(intensity) 값 등이 있다. OpenGL에서는 38가지의 조합이 가능한데, 단순히 어떤 조합을 사용할 것인가뿐만 아니라, 각 요소 값에 대하여 몇 비트를 사용할 지를 설정할 수가 있다. 38가지의 서로 다른 내부 형식 중 가장 널리 쓰이는 것으로 GL_RGB, GL_RGBA, GL_LUMINANCE, GL_LUMINANCE_ALPHA 등을 들 수가 있다. 그외에도 GL_ALPHA와 GL_INTENSITY 등을 생각할 수가 있는데, 각 상수가 의미하는 내용은 분명하다. 한편 GL_RGB5_A1과 같이 구체적으로 각 구성 요소들에 대하여 몇 비트를 할당할 지를 설정할 수가 있다. 이 상수는 R, G, B 각 채널에 대하여 5 비트, 그리고 A 채널에 대하여 1비트, 즉 총 16비트를 사용할 것을 명시적으로 요구하는데, OpenGL에서는 항상 이러한 요구가 만족되는 것은 아니고, 시스템에서 구현된 것 중 프로그래머가 요구하는 형식에 가장 가까운 내부 저장 형식을 사용한다. 38가지 모든 형식에 대하여 자세한 내용을 알기 위해서는 OpenGL 매뉴얼을 참조하기 바란다. 참고로 표 5.1에는 기본이 되는 여섯 가지의 내부 형식 각각에 대하여 $(R, G, B, A)$의 채널 값들 중 어떤 값을 선별하여 해당 요소 필드로 사용하는지가 요약이 되어 있다.

마지막으로 glTexImage2D(*) 함수의 level 인자는 현재 설정하려하는 텍스춰 이미지의 상세 레벨(level-of-detail)을 설정하는데 사용이 된다. 텍스춰 매핑을 통한 렌더링을 할 때 뒤에서 설명할 밉매핑 기능을 사용하면 동일한 텍스춰 이미지를 여러 개의 해상도로 텍스춰 메모리에 올려준다. 이 때 원래의 텍스춰, 즉 해상도가 가장 높은 이미지의 레벨이 0이 되고, 해상도가 낮아질수록 레벨 값이 $1, 2, 3, \cdots$과 같이 증가한다. 따라서 밉매핑을 하지 않는다면 이 인자에 0 값을 설정하면 된다.

지금 살펴보고 있는 예제 프로그램에서는 텍스춰 데이터와 관련하여 다음과 같이 간단하게 스트럭춰 타입 Texture를 정의한 뒤, 그림 5.5의 세 개의 텍스춰에 대하여 각각 tex_sy(텍스춰 1), tex_ng(텍스춰 2), tex_br(텍스춰 3)의 변수를 선언하였다. 여기서 Texture 타입의 변수의 ns와 nt는 텍스춰 이미지의 해상도를 나타내고, tmap은 텍스춰 이미지가 저장된 메모리 영역을 가리키는 포인터 값을 의미한다.

```
typedef struct {
  int ns, nt;
  GLubyte *tmap;
} Texture;
Texture tex_sy, tex_ng, tex_br;
```

아래의 세 개의 OpenGL 문장은 앞에서 설명한 방식으로 각각 해당 텍스춰 이미지를 설정하고 있는데, 두 번째 텍스춰의 경우에만 A 채널을 포함하고 있다.

```
// texture 1
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_sy.ns, tex_sy.nt,
             0, GL_RGB, GL_UNSIGNED_BYTE, tex_sy.tmap);


// texture 2
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tex_ng.ns, tex_ng.nt,
             0, GL_RGBA, GL_UNSIGNED_BYTE, tex_ng.tmap);
// texture 3
```

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_br.ns, tex_br.nt,
             0, GL_RGB, GL_UNSIGNED_BYTE, tex_br.tmap);
```

### 3.2.2 텍스춰 객체와 바인딩

지금까지 설명한 것을 요약하면 glTexImage2D(*) 함수는 메인 메모리에 저장되어 있는 이미지를 텍스춰 매핑 계산에 사용을 할 수 있도록 원하는 내부 형식으로 변환시켜 텍스춰 메모리에 올려주는 역할을 한다. 만약 어떤 이미지를 렌더링 하기 위하여 한 개의 텍스춰만 사용한다면, 이 함수를 통하여 텍스춰를 텍스춰 메모리에 올려 놓고 렌더링 작업을 수행하면 된다. 하지만 일반적으로는 한 장의 이미지를 생성하기 위해서 여러 개의 텍스춰를 사용한다. 이런 경우에는 장면을 구성하는 각 기하 물체를 그리기 전에 매번 그 물체에 입힐 텍스춰 이미지를 glTexImage2D(*) 함수를 사용하여 텍스춰 메모리에 올려주어야 한다. 3차원 게임이나 가상 현실 응용 소프트웨어에서와 같이 이미지를 계속해서 생성해야 할 경우에는 사용하는 텍스춰 이미지에 대하여 이 함수를 반복적으로 호출해야 하는데, 이는 종종 OpenGL 시스템에 적지 않은 부하를 걸리게 한다.

여러 개의 텍스춰를 사용한다면 더 효율적인 방법은 텍스춰 객체(texture object)를 사용하는 것이다. 텍스춰 객체는 텍스춰 이미지 자체에 대한 정보와 그 텍스춰로부터 텍셀의 값을 구하는 방식에 대한 모든 정보를 저장하는 자료 구조라 생각하면 된다. 처음에 사용하려고 하는 각 텍스춰마다 텍스춰 객체를 한 개씩 생성을 해 놓고, 기하 물체를 그릴 때마다 해당 텍스춰 객체를 바인딩하면서 렌더링을 하면, glTexImage2D(*) 함수를 반복적으로 호출하면서 매번 텍스춰 이미지를 텍스춰 메모리에 올리는 것보다 더 빠르게 텍스춰 매핑 계산을 수행할 수가 있다.

텍스춰 객체를 사용하려면 우선 각 객체마다 고유한 이름이 필요한데, 이를 위하여 OpenGL에서는 무부호 정수 값을 사용한다. 텍스춰 객체는 동적인 생성과 제거가 가능하므로, 프로그래머가 직접 이름을 붙이기 보다는 OpenGL 시스템에서 제공하는 이름을 사용하는 것이 혼란을 막을 수가 있다. 사용자는 텍스춰 객체를 생성하기 위하여 이름이 필요할 때, void glGenTextures(GLsizei n, GLuint *textureNames); 함수를 수행시켜 이름을 할당 받을 수 있다. n개의 이름이 필요할 경우 이 함수를 수행시키면, OpenGL 시스템이 사용 가능한 객체 이름 n개를 생성하여 두 번째 인자 textureNames가 가리키는 배열에 저장을 해준다. 아래의 프로그램 예 5.1을 보면, Line (a)에서 세 개의 이름을 할당 받아 무부호 정수 타입의 배열 tex_name[3]에 저장하고 있음을 알 수가 있다.

**프로그램 예 5.1** 텍스춰 객체의 생성.

```
GLuint tex_name[3];
void set_textures(void) {
 glGenTextures(3, tex_name); // Line (a)

 glBindTexture(GL_TEXTURE_2D, tex_name[0]); // Line (b)
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_sy.ns,
tex_sy.nt,
             0, GL_RGB, GL_UNSIGNED_BYTE, tex_sy.tmap);

 glBindTexture(GL_TEXTURE_2D, tex_name[1]); // Line (c)
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
 // Line (d)
```

```
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tex_ng.ns,
tex_ng.nt,
               0, GL_RGBA, GL_UNSIGNED_BYTE, tex_ng.tmap);


  glBindTexture(GL_TEXTURE_2D, tex_name[2]); // Line (e)
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_br.ns,
tex_br.nt,
               0, GL_RGB, GL_UNSIGNED_BYTE, tex_br.tmap);
}
```

일단 이름을 할당을 받으면 텍스춰 객체를 생성할 수가 있다. 이러한 과정은 glBindTexture(GLenum target, GLuint textureName); 함수를 통하여 수행하면 되는데, 여기서 첫 번째 인자 target은 현재 2차원 텍스춰 매핑을 고려하고 있으므로 GL_TEXTURE_2D 값을 넘기고, 두 번째 인자 textureName은 생성을 하려하는 텍스춰 객체에 붙일 이름을 넘기면 된다. 위의 코드의 Line (b), (c), (e)에서는 할당 받은 세 개의 이름을 사용하여 해당 이름을 가지는 텍스춰 객체를 생성하고 있다. 텍스춰 객체는 텍스춰와 그에 관련된 정보를 저장하는 자료 구조라 하였는데, 이렇게 처음 텍스춰 객체가 생성이 되면 모든 정보가 디폴트 상태로 초기화가 된다[5].

[5]OpenGL에서 이름이 0인 텍스춰 객체는 초기 상태 값을 저장하는 객체로 사용이 되는데, 새로운 텍스춰 객체가 생성이 될 때마다 이 객체의 값이 복사가 된다. 따라서 OpenGL에서는 이 객체를 특별하게 취급하므로, 프로그래머는 자신이 생성하는 텍스춰 객체에 대하여 0번 이름을 사용을 할 수가 없다.

OpenGL에서 할당 받은 객체 이름에 대하여 glBindTexture(*) 함수를 최초로 부를 경우에는 우선 객체에 대한 자료 구조가 생성이 된 후, 이 객체의 텍스춰가 현재 텍스춰(current texture)로 설정이 된다. 한편 이미 만들어진 객체에 대하여 이 함수를 호출하면 해당 텍스춰가 현재 텍스춰로 지정이 된다. 물론 렌더링 과정에서 텍스춰 객체가 더 이상 필요가 없으면, void glDeleteTexture(GLsizei n, const GLuint *textureNames); 함수를 사용하여 동적으로 제거를 할 수가 있다. 이 함수는 textureNames에 연달아 저장되어 있는 n개의 이름에 대한 텍스춰 객체를 제거하는데, 텍스춰 객체는 시스템의 자원을 사용하는 대상이기 때문에, 더이상 불필요한 객체는 제거하는 것이 좋은 방법이라 하겠다. 참고로 OpenGL에서는 현재 텍스춰로 지정된 객체를 제거하면 자동적으로 0번 텍스춰 객체가 현재 텍스춰로 지정이 된다.

### 3.3 텍스춰 상태와 텍셀 생성 인자의 설정

텍스춰 매핑 계산을 원하는 방식으로 수행하기 위해서는 그에 관련된 여러 가지 상태 인자를 적절하게 설정해주어야 한다. OpenGL에서는 텍스춰에 관련된 상태를 변경하는 함수를 호출을 하면, 현재 텍스춰로 지정된 텍스춰 객체에 영향을 미치는 방식을 취한다. 위의 예제 프로그램에서는 glTexParameteri(*) 함수와 glTexImage2D(*) 함수가 텍스춰의 상태를 변경시키는 역할을 하는데, 이는 바로 현재 텍스춰에 영향을 미치게 된다.

텍스춰 매핑 과정을 정확하게 이해하기 위하여 452쪽의 그림 5.3을 다시 보도록 하자. 다른 관점에서 보면 텍스춰(Texturing) 모듈의 역할은 텍스춰 매핑의 목적에 맞게 프래그먼트의 색깔을 변화시키는 것이다. 이를 위하여 텍셀 생성과 텍스춰 적용이라는 두 가지 계산 과정을 거쳐야 한다. 텍셀 생성 과정은 현재 흘러 들어오는

프래그먼트에 연관된 텍스춰 좌표 $(s, t, r, q)$를 사용하여 현재 적용을 하려하는 텍스춰 이미지로부터 적절한 텍스춰 색깔 $(C_t, A_t)$를 구하는 것을 목표로 한다고 하였다. 이러한 계산을 위해서 어떤 텍스춰 이미지로부터 색깔을 가져올 것인지와, 또한 어떠한 방식으로 그러한 값을 구할 것인지에 대한 인자를 명확히 설정해주어야 한다. 사용하고자 하는 각 텍스춰마다 서로 다른 값들을 사용하여 이에 관련된 인자들을 설정할 수가 있는데, 그러한 설정 정보가 텍스춰 상태(texture states)라는 형태로 표현이 되어 각 텍스춰 객체에 저장이 된다.

텍스춰 상태를 구성하는 정보는 크게 두 가지 부류로 나누어진다. 첫 번째 부류는 glTexImage2D(*) 함수로 설정이 되는 정보로서, 뒤에서 설명할 밉맵을 포함한 텍스춰 이미지 자체에 대한 인자들로 형성이 되는데, 텍스춰 이미지의 해상도, 경계 두께, 내부 형식, 그리고 각 요소 필드에 할당된 비트 수 등이 이에 포함된다. 두 번째 부류는 현재 프래그먼트의 텍스춰 좌표가 주어졌을 때, 텍스춰 이미지를 액세스하여 텍스춰 색깔을 계산하는 방식에 영향을 미치는 인자들로 구성이 된다. 이러한 인자들은 glTexParameterf(GLenum target, GLenum pname, GLfloat param); 함수를 사용하여 설정할 수가 있다[6].

표 5.2에는 이 함수를 통하여 설정할 수 있는 11 가지의 텍스춰 상태와 그에 대응되는 OpenGL 인자 이름, 그리고 사용 가능한 인자 값들이 요약이 되어 있다. glTexParameter*(*) 함수의 첫 번째 인자 target으로는 GL_TEXTURE_2D를 사용하면 되고, 두 번째와 세 번째 인자 pname과 param에는 각각 이 표의 인자 이름과 그에 해당하는 사용 가능 인자를 사용하면 되는데, 각 인자들이 의미하는 바는 필요할 때마다 설명을 하도록 하겠다.

[6] 이 함수 외에도 정수 버전인 glTexParameteri(GLenum target, GLenum pname, GLint param);와 벡터 버전인 glTexParameterfv(GLenum target, GLenum pname, const GLfloat *params);와 glTexParameteriv(GLenum target, GLenum pname, const GLint *params); 등을 사용할 수가 있다.

| 상태 종류 | OpenGL 인자 이름 | 사용 가능 인자 |
|---|---|---|
| 확대 필터 | GL_TEXTURE_MAG_FILTER | GL_NEAREST, GL_LINEAR |
| 축소 필터 | GL_TEXTURE_MIN_FILTER | GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR GL_LINEAR_MIPMAP_LINEAR |
| 랩 모드 ($s$) | GL_TEXTURE_WRAP_S | GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_EDGE |
| 랩 모드 ($t$) | GL_TEXTURE_WRAP_T | 상 동 |
| 랩 모드 ($r$) | GL_TEXTURE_WRAP_R | 상 동 |
| 텍스춰 경계 색깔 | GL_TEXTURE_-BORDER_COLOR | RGBA 색깔: (0, 0, 0, 0) |
| 최소 LOD | GL_TEXTURE_MIN_LOD | 부동 소수점 숫자: -1000.0 |
| 최대 LOD | GL_TEXTURE_MAX_LOD | 부동 소수점 타입 숫자: 1000.0 |
| 기저 밉맵 레벨 | GL_TEXTURE_BASE_LEVEL | 정수 타입 숫자: 0 |
| 최대 밉맵 레벨 | GL_TEXTURE_MAX_LEVEL | 정수 타입 숫자: 1000 |
| 텍스춰 우선 순위 | GL_TEXTURE_PRIORITY | 0과 1 사이의 부동 소수점 타입 숫자 |

표 5.2: glTexParameter*(*) 함수로 설정 가능한 텍스춰 상태

요약을 하면 텍셀 생성 모듈은 현재 텍스춰로 지정이 된 텍스춰 객체에 저장이 되어 있는 텍스춰 상태의 값들(텍스춰 이미지 포함)을 사용하여 텍스춰 색깔을 구한다. 이러한 텍스춰 상태 값들은 glTexImage2D(*) 함수와 glTexParameter*(*) 함수, 그리고 glTexSubImage2D(*), glCopyTexImage2D(*), glCopyTexSubImage2D(*), glPrioritizeTextures(*) 함수 등을 사용하여 설정할 수 있는데, 물론 이 함수들은 현재 텍스춰 객체의 상태 값들을 변경시키는 역할을 한다.

이러한 관점에서 프로그램 예 5.1을 다시 보면 `set_textures()` 함수는 사용하려는 세 개의 텍스춰 각각에 대하여 텍스춰 객체를 생성하여 각 텍스춰에 맞는 텍스춰 상태를 설정하는 함수라는 것을 쉽게 알 수가 있다. 텍스춰 상태에 영향을 미치는 함수는 현재 텍스춰로 바인딩이 되어 있는 텍스춰 객체에 영향을 미치는데, 텍스춰 상태는 언제나 동적으로 변경을 할 수가 있다.

## 3.4 텍스춰 적용 함수의 설정

텍셀 생성 모듈과 함께 텍스춰 매핑 계산의 중요한 부분을 이루는 텍스춰 적용 모듈은 앞에서도 기술한 바와 같이 텍스춰 좌표와 현재 텍스춰 상태를 사용하여 계산하는 텍스춰 색깔 $(C_t, A_t)$를 현재 프래그먼트의 기반이 되는 색깔 $(C_f, A_f)$와 혼합하여, 프래그먼트의 새로운 색깔 $(C, A)$를 구하는 것을 목표로 한다(452쪽의 그림 5.3을 다시 볼 것.). 이 모듈은 라이팅 계산 과정에서 구한 물체의 기반이 되는 색깔과 텍스춰 이미지로부터 가져온 색깔을 어떻게 섞을 것인가를 결정하는데, 어떠한 방법을 사용하는가에 따라 텍스춰를 물체에 입히는 방식이 달라진다.

OpenGL 시스템은 네 가지의 적용 방법을 제공하는데, 어떠한 방식을 선택할 것인가는 glTexEnvi(GLenum target, GLenum pname, GLint param); 함수와 glTexEnvf(GLenum target, GLenum pname, GLfloat param); 함수, 그리고 함수 이름 끝에 v가 붙는 포인터 버전 함수를 사용하여 설정할 수가 있다. 우선 이 함수의 첫 번째 인자 target에 대해서는 상수 GL_TEXTURE_ENV를 사용하여야 한다. 다음 pname을 GL_TEXTURE_ENV_MODE로 설정을 하면, param으로 GL_DECAL, GL_REPLACE, GL_MODULATE, 그리고 GL_BLEND 중 하나를 선택하여 물체의 기반 색깔과 텍스춰 색깔을 결합하는 방식을 결정할 수가 있다.

표 5.3에는 459쪽의 표 5.1에 설명되어 있는 여섯 개의 내부 형식에 대하여 네 가지의 텍스춰 적용 방식 함수가 요약되어 있다. 설명의 편의상 두 가지 내부 형식 GL_RGB와 GL_RGBA를 중심으로 텍스춰 적용 방식을 살펴보자. 가장 단순한 형태의 텍스춰 적용은 GL_REPLACE 모드를 사용하는 것인데, 보다시피 프래그먼트의 원래의 기반 RGB 색깔 $C_f$는 무시가 되고, 텍스춰 색깔 $C_t$가 프래그먼트의 새로운 색깔이 됨을 알 수가 있다. GL_RGB와 GL_RGBA의 두 가지 경우의 차이점은 후자의 경우 텍스춰에 알파 채널 값 $A_t$가 주어져 있기 때문에 그 값을 결과 알파 값으로 사

| 내부 형식 | GL_REPLACE | GL_MODULATE |
|---|---|---|
| GL_ALPHA | $C = C_f$, $A = A_t$ | $C = C_f$, $A = A_f A_t$ |
| GL_LUMINANCE | $C = L_t$, $A = A_f$ | $C = C_f L_t$, $A = A_f$ |
| GL_LUMINANCE_ALPHA | $C = L_t$, $A = A_t$ | $C = C_f L_t$, $A = A_f A_t$ |
| GL_INTENSITY | $C = I_t$, $A = I_t$ | $C = C_f I_t$, $A = A_f I_t$ |
| GL_RGB | $C = C_t$, $A = A_f$ | $C = C_f C_t$, $A = A_f$ |
| GL_RGBA | $C = C_t$, $A = A_t$ | $C = C_f C_t$, $A = A_f A_t$ |
| 내부 형식 | GL_DECAL | GL_BLEND |
| GL_ALPHA | 미 정 | $C = C_f$, $A = A_f A_t$ |
| GL_LUMINANCE | 미 정 | $C = C_f(1 - L_t) + C_c L_t$, $A = A_f$ |
| GL_LUMINANCE_ALPHA | 미 정 | $C = C_f(1 - L_t) + C_c L_t$, $A = A_f A_t$ |
| GL_INTENSITY | 미 정 | $C = C_f(1 - I_t) + C_c I_t$, $A = A_f(1 - I_t) + A_c I_t$ |
| GL_RGB | $C = C_t$, $A = A_f$ | $C = C_f(1 - C_t) + C_c C_t$, $A = A_f$ |
| GL_RGBA | $C = C_f(1 - A_t) + C_t A_t$, $A = A_f$ | $C = C_f(1 - C_t) + C_c C_t$, $A = A_f A_t$ |

표 5.3: 네 가지의 텍스춰 적용 방식

용을 하고, 전자의 경우에는 $A_f$를 그대로 알파 값으로 사용을 한다는 점이다. 다시 말해서 GL_REPLACE 모드는 물체의 원래의 색깔을 무시하고 그 위에 텍스춰를 입히는 방식을 취한다. 그림 5.4(b)의 왼쪽의 두 사각형은 이 모드를 사용하여 텍스춰를 입혔는데, 물체에 대한 라이팅 계산이 모두 무시가 되고 단순히 텍스춰 이미지가 나타나기 때문에, 라이팅 계산을 할 경우에는 별로 자연스러운 방법은 아니라고 할 수가 있다.

다음 GL_DECAL 모드는 GL_RGB와 GL_RGBA 두 가지 내부 형식에 대해서만 의미를 가진다. 이 모드는 내부 형식 GL_RGB에 대해서는 GL_REPLACE 모드와 동일하지만, 알파 값을 가지는 텍스춰 이미지를 위한 내부 형식 GL_RGBA에 대해서는 다른 의미를 가지는 것을 알 수가 있다. 이 경우에는 물체의 기반 색깔 $C_f$와 텍스춰 이미지로부터의 색깔 $C_t$를 텍스춰의 알파 값 $A_t$의 비율로 섞는다. 이 모드는 우리가 어릴 때 그림이 그려져 있는 비닐을 물체에 대고 문지르면 그림이 물체에 달

라붙게 되는, 그러한 투명 비닐의 그림을 물체에 입히는 효과를 내는데 사용한다고 생각하면 된다. 즉 비닐 위의 그림의 색깔이 짙으면($A_t$가 크면) 상대적으로 물체의 색깔이 덜 보이고, 그림의 색깔이 옅어 투명하게 보이면($A_t$가 작으면) 물체의 색깔이 상대적으로 더 강조가 되는 방식으로 결과 색깔이 계산이 된다. 따라서 이 모드는 주로 알파 값을 가지는 텍스춰를 알파 값의 의미에 따라 적절이 입히는데 사용이 된다. 그림 5.4(b)의 물주전자는 이 모드를 사용하여 텍스춰를 입힌 것이다. 여기에서 사용한 그림 5.5(b)의 텍스춰 이미지를 다시 보면, 이 텍스춰의 각 텍셀의 알파 값은 흰색이 나타나는 지역은 0으로, 그리고 다른 부분은 0.75로 설정이 되어 있다. 또한 텍스춰 이미지의 경계에 있는 텍셀들의 알파 값들도 0으로 설정이 되어 있다. GL_DECAL 모드를 사용하여 이러한 텍스춰를 입히면 알파 값이 0이 아닌 영역의 텍스춰 색깔이 0.75의 비율로 섞이고, 나머지 부분은 물체의 기반 색깔이 그대로 나타나는데, 그 결과 조명 계산의 효과가 잘 보존이 됨을 알 수 있다.

다음 GL_MODULATE 모드는 표 5.3를 보면 알 수가 있듯이, 텍스춰 색깔의 각 채널 값을 물체의 기반 색깔의 해당 채널에 곱해서 결과 색깔을 구하는 방식을 취한다. 따라서 이 모드를 사용하면 물체의 기반 색깔이 글자 그대로 텍스춰 색깔에 의해 조절이 되는 효과가 발생한다. 그림 5.4(b)의 병풍은 이 모드를 사용한 것인데, 주황색 계통의 기반 색깔이 텍스춰와 자연스럽게 결합이 되었음을 알 수가 있다. 이 모드는 조명 계산을 할 경우에 가장 보편적으로 사용이 되는 텍스춰 적용 모드인데, 일반적으로 물체의 바탕색을 흰색으로 하여 조명 계산을 한 후, 이 모드를 사용하여 텍스춰를 입히면 라이팅 효과를 잘 보존하면서 텍스춰 이미지를 자연스럽게 물체에 입힐 수가 있다.

마지막으로 GL_BLEND 모드를 사용하면 약간 복잡한 형태의 함수를 사용하여 텍스춰 색깔을 혼합한다. 이 때 사용하는 색깔 $C_c$는 사용자가 지정한 값으로서,

glTexEnvf() 함수의 두 번째 인자를 GL_TEXTURE_ENV_COLOR, 그리고 세 번째 인자를 설정하려고 하는 색깔의 R, G, B, A 값이 부동 소수점 타입으로 저장되어 있는 곳의 주소를 사용하여 설정할 수가 있다. 이 모드는 텍스춰 이미지의 색깔을 원하는 색깔로 한 번 필터링하여 물체에 입히는데 유용하게 사용을 할 수가 있다.

### 3.5 텍스춰 좌표의 설정과 변환

지금까지는 텍스춰 매핑 과정에 있어 중요한 두 가지 계산 과정에 영향을 미치는 텍스춰 인자의 설정에 대하여 알아보았다. 이러한 인자를 모두 설정을 하고난 후, 적절하게 뷰잉과 라이팅 계산을 하면서 물체를 그리면 된다. 이 때 텍스춰를 올바르게 입혀주기 위해서 물체를 구성하는 각 기하 프리미티브의 꼭지점에 대응이 되는 텍스춰 좌표를 설정해주어야 한다. 텍스춰 좌표를 설정하는 방법은 크게 두 가지로 나누어 진다. 하나는 물체를 모델링하는 단계에서 꼭지점의 좌표와 법선 벡터뿐만 아니라 각 꼭지점에 해당하는 텍스춰 좌표를 미리 구해 놓고, 렌더링 과정에서 glBegin(*) 함수와 glEnd() 함수 사이에서 각 꼭지점에 대한 정보를 기술할 때 텍스춰 좌표도 같이 설정을 하는 방법이다. 다른 하나는 그렇게 텍스춰 좌표를 미리 생성을 해 놓는 것이 아니라, 렌더링 계산 시 프로그래머가 설정한 방식으로 각 꼭지점마다 동적으로 텍스춰 좌표를 자동 생성하여 사용하는 방식이다. 보통 후자는 환경 매핑나 투영 텍스춰 등 고급 렌더링 기법을 구현하는데 있어 핵심적인 기능을 제공하는데, 이에 대해서는 뒤에서 살펴 보기로 하고, 일단은 전자의 방식에 대하여 알아보도록 하자.

2차원 텍스춰 좌표는 glTexCoord2f(GLfloat s, GLfloat t); 함수와 glTexCoord2fv(const GLfloat *v); 함수를 사용하여 설정을 할 수가 있다. 이 함수는 다른 타입의 좌표 값을 사용하는 버전도 있는데 이에 대해서는 OpenGL 매뉴얼을 참조

하기 바란다. 앞에서도 언급한 바와 같이 OpenGL에서 텍스춰 좌표는 $(s, t, r, q)$와 같이 동차 좌표계 형태로 표현이 되는데, 일반적으로 2차원 텍스춰에 대해서는 $r = 0$, 그리고 $q = 1$이라 생각하면 된다. 아래의 프로그램 예 5.2의 `draw_object(*)` 함수를 보면, 이 함수의 사용 방식을 쉽게 이해를 할 수가 있다. glNormal3*(*) 함수가 법선 벡터에 대한 현재 값을 설정하는 것과 같이(Line (b)), 이 함수는 텍스춰 좌표에 대한 현재 값을 설정을 하는데 사용이 된다(Line (a)). 따라서 glVertex*(*) 함수를 사용하여 꼭지점의 좌표를 기술할 때(Line (c)), 현재 텍스춰 좌표가 현재 법선 벡터와 함께 꼭지점에 달라붙어 기하 파이프라인으로 흘러 들어간다.

**프로그램 예 5.2** 텍스춰 좌표의 설정.

```
void draw_object(Object *obj) {
  int i, j;
  Polygon *ptr;

  i = 0; ptr = obj->polygon;
  while(i++ < obj->npoly) {
    glBegin(GL_POLYGON);
      for (j = 0; j < ptr->nvert; j++) {
        glTexCoord2fv(ptr->tcoord[j]); // Line (a)
        glNormal3fv(ptr->norm[j]); // Line (b)
        glVertex3fv(ptr->vert[j]); // Line (c)
      }
    glEnd();
    ptr++;
  }
}

void draw_teapot(void) {
  glShadeModel(GL_SMOOTH);
  glBindTexture(GL_TEXTURE_2D, tex_name[1]);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

```
  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient1);
       ⋮
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
    glTranslatef(0.8, 0.7, 0.0);
    glRotatef(180.0, 0.0, 1.0, 0.0);
    glRotatef(-90.0, 1.0, 0.0, 0.0);
    glScalef(0.95, 0.95, 0.95);

    glMatrixMode(GL_TEXTURE); // Line (d)
    glPushMatrix(); // Line (e)
      glTranslatef(-0.5, 0.0, 0.0); // Line (f)
      glScalef(2.0, 1.0, 1.0);
      draw_object(&teapot); // Line (g)
    glPopMatrix(); // Line (h)
    glMatrixMode(GL_MODELVIEW);
  glPopMatrix();
}

void draw_hcylinder(void) {
  glShadeModel(GL_FLAT);

  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient0);
       ⋮
  glBindTexture(GL_TEXTURE_2D, tex_name[0]);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
    glTranslatef(0.8, -1.7, 0.0);
    glScalef(1.8, 1.8, 1.8);
    draw_object(&hcyl);
  glPopMatrix();
}

void draw_quadrilaterals(void) {
  glBindTexture(GL_TEXTURE_2D, tex_name[2]);
```

```
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient2);
       ⋮
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
    glTranslatef(-2.5, 0.0, 0.0);
    glBegin(GL_TRIANGLES);
      glNormal3f(0.0, 0.0, 1.0);

      glTexCoord2f(1.0, 0.0);
      glVertex3f(1.0, -1.0, 0.0);
      glTexCoord2f(1.0, 1.0);
      glVertex3f(1.0, 1.0, 0.0);
      glTexCoord2f(0.0, 1.0);
      glVertex3f(-1.0, 1.0, 0.0);

      glTexCoord2f(1.0, 0.0);
      glVertex3f(1.0, -1.0, 0.0);
      glTexCoord2f(0.0, 1.0);
      glVertex3f(-1.0, 1.0, 0.0);
      glTexCoord2f(0.0, 0.0);
      glVertex3f(-1.0, -1.0, 0.0);
    glEnd();
  glPopMatrix();

  glPushMatrix();
    glTranslatef(-2.5, -2.5, 0.0);
    glBegin(GL_TRIANGLES);
      glNormal3f(0.0, 0.0, 1.0);
      glTexCoord2f(1.0, 0.0);
      glVertex3f(1.0, -1.0, 0.0);
        ⋮
    glEnd();
  glPopMatrix();
}

void display(void) {
```

```
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  draw_hcylinder(); // Line (i)
  draw_teapot(); // Line (j)
  draw_quadrilaterals(); // Line (k)
  glFlush();
}
```

마지막에 있는 함수 display()는 디스플레이 컬백 함수로서 병풍(Line (i)), 물주전자(Line (j)), 그리고 두 개의 사각형을 그려주는 함수(Line (k))들을 차례대로 호출하고 있다. 이 함수들은 각각 glBindTexture(*) 함수를 사용하여 해당 텍스춰를 현재 텍스춰로 설정을 한 후, glTexEnv*(*) 함수로 텍스춰 적용 모드를 설정한 뒤 기하 물체를 그리고 있다. 한 가지 중요한 것은 glTexCoord2f(*) 함수를 통하여 텍스춰 좌표를 기술할 때 어떠한 일이 일어나는가 하는 것이다. 여러 번 강조한 바와 같이 OpenGL의 기하 파이프라인에는 두 개의 중요한 행렬 스택이 있다. 기하 물체를 그릴 때 꼭지점 좌표와 법선 벡터는 첫 번째 행렬 스택인 모델뷰 행렬 스택의 탑에 있는 행렬에 곱해져 눈 좌표계로 변환이 되고, 이후 라이팅 계산이 끝나면 꼭지점 좌표가 두 번째 행렬 스택인 투영 행렬 스택의 탑에 있는 행렬에 곱해져 계속해서 기하 파이프라인을 따라 흘러 간다고 하였다. 텍스춰 좌표도 꼭지점에 연관되기 전에 텍스춰 행렬 스택(texture matrix stack)이라 하는 또 하나의 기하 변환을 위한 행렬 스택의 탑에 있는 행렬에 곱해져 그 결과 값이 꼭지점에 붙게 된다.

텍스춰 행렬 스택도 4행 4열의 임의의 변환 행렬을 저장할 수 있는 스택으로서, 행렬 스택에 영향을 미치는 모든 OpenGL 함수를 사용하여 조작을 할 수가 있다. OpenGL 규약에 의하면 이 행렬 스택은 최소한 깊이가 2 이상이어야 하는데, glMatrixMode(GL_TEXTURE);와 같은 문장을 통하여 현재 행렬 스택으로 지정을 할 수가 있다. 디폴트로 텍스춰 행렬 스택의 탑의 행렬은 단위 행렬로 초기화가 되

어 있어, `draw_hcylinder()` 함수와 `draw_quadrilaterals()` 함수에서와 같이 이 행렬 스택을 명시적으로 조작을 하지 않으면 glTexCoord2f(*) 함수를 사용하여 설정한 텍스춰 좌표 $(s, t, 0, 1)$이 단위 행렬에 곱해진 후, 바로 원래의 값이 꼭지점에 붙게 된다.

반면 `draw_teapot()` 함수를 보면 Line (g)의 `draw_object(&teapot)` 함수 호출을 통하여 물체를 그리기 전에 텍스춰 좌표에 대한 조작을 하고 있음을 볼 수가 있다. Line (d)에서 텍스춰 행렬 스택을 선택을 하는데, 이전의 모델뷰 행렬 스택에 대한 조작은 모델링 변환임은 쉽게 알 수가 있을 것이다. Line (e)에서 단위 행렬인 현재 텍스춰 행렬(current texture matrix)의 내용을 보존하고, Line (f)와 그 다음 문장을 통하여 원하는 변환 행렬을 올려 놓고 물체를 그린 후, Line (h)에서 원래의 상태로 돌아오고 있다[7].

그러면 그 결과 어떠한 일이 일어날까? 이를 파악하기 위하여 456쪽의 그림 5.6을 다시 보자. 물주전자의 굵게 표시된 영역의 다각형에 대하여 왼쪽에서 오른쪽 방향으로는 $s$ 좌표가 0에서 1로, 그리고 아래에서 위쪽 방향으로는 $t$ 좌표가 0에서 1로 변하도록 텍스춰 좌표가 붙여져 있고, 나머지 꼭지점에 대해서는 $(-1, -1)$ 값이 설정되어 있다. 이 프로그램에서 텍스춰 행렬 스택에 올리는 변환은 $M_T = T(-0.5, 0, 0) \cdot S(2, 1, 1)$이므로, 변환 후의 텍스춰 좌표는 그림 5.7에 주어진 것과 같이 굵게 표시된 영역에 대해서는 텍스춰 좌표가 $[-0.5, 0] \times [1.5, 1]$의 범위를 가지고, 그 나머지 꼭지점은 $(-2.5, 1)$ 값을 가진다.

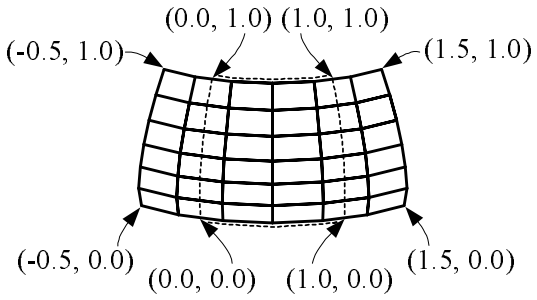
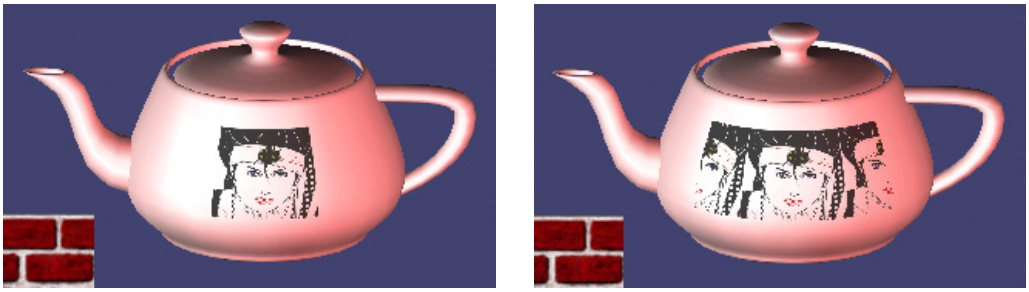



그림 5.7: 텍스춰 좌표의 변환

앞에서 텍스춰 이미지는 텍스춰 공간에서 $s$와 $t$ 좌표 각각에 대하여 0과 1 사이의

[7]물론 굳이 스택을 푸쉬할 필요는 없고, 변환 행렬을 올려 놓고 물체를 그린 후 다시 단위 행렬을 올려줘도 된다.



(a) GL_CLAMP 모드의 사용 (b) GL_REPEAT 모드의 사용

그림 5.8: 텍스춰 랩 모드

값을 가지도록 정의가 된다고 하였는데, 문제는 물주전자의 꼭지점에는 이러한 범위를 벗어나는 텍스춰 좌표 값이 설정이 된다는 점이다. 과연 OpenGL에서는 이러한 상황을 어떻게 처리를 할까? 이는 텍스춰 랩 모드(texture wrap mode)라 하는 텍스춰 상태에 설정된 값에 따라 다르게 처리가 된다(표 5.2 참조). 각 텍스춰 좌표 변수 마다 적절한 랩 모드를 설정할 수가 있는데, 이 프로그램에서는 각 좌표 변수에 대하여 GL_CLAMP 모드로 설정하여 렌더링을 하였다(462쪽의 프로그램 예 5.1의 Line (d)와 그 다음 문장). 이 모드에서는 각 변수에 대하여 0보다 작으면 0으로, 그리고 1보다 크면 1로 무조건 변환을 하여 그 값을 텍스춰 이미지를 액세스하는데 사용한다. 따라서 이 경우에는 그림 5.7에서 텍스춰 좌표가 $[0.0, 0.0] \times [1.0, 1.0]$인 영역(점선으로 둘러싼 영역)에 대해서는 변함이 없으나, 그 외의 지역에 대해서는 $s$와 $t$ 좌표 중 최소한 하나는 0 또는 1 값을 가지게 된다.

그림 5.8(a)를 보면 그림 5.7의 점선으로 둘러싼 영역에 대해서만 텍스춰가 입혀진 것을 알 수가 있다. 그러면 나머지 영역에 대해서는 왜 아무런 텍스춰가 나타나지 않을까? 앞에서 기술한 바와 같이 여기서 사용하는 텍스춰는 내부 형식이 GL_RGBA로서, 변두리의 텍셀들, 다시 말해서 $s$와 $t$ 값이 0이거나 1인 영역에 대응되는 텍셀들의 알파 값을 0으로 만들어 주었다. 텍스춰가 나타나지 않는 지점에 대

해서는 텍스춰 좌표가 텍스춰 이미지의 변두리 영역의 해당하는 값을 가지게 되는데, 텍스춰를 적용할 때 GL_DECAL 모드를 사용을 하기 때문에 알파 값이 0인 색깔을 사용하여 텍스춰를 입히게 되고, 따라서 결과적으로 텍스춰가 아무런 영향을 미치지 않는 것이다. 만약 GL_REPLACE나 GL_MODULATE 모드를 사용하면 전혀 다른 현상이 나타날 것이다.

텍스춰 랩 모드의 디폴트 모드는 GL_REPEAT로서 GL_CLAMP 모드와는 다른 방식으로 텍스춰 좌표를 처리한다. 이 모드에서는 해당 텍스춰 변수 값이 $a$라면 $a - \lfloor a \rfloor$ 값을 텍스춰 좌표 값으로 사용을 한다. 예를 들어 $a$가 -1.2라면 0.8, 그리고 3.4라면 0.4가 되는데, 이는 텍스춰를 반복하여 입히는데 사용하는 모드임을 알 수가 있다. 따라서 이 모드를 사용한다면 그림 5.8(b)와 같이 텍스춰가 입혀질 것이다.

지금까지 그림 5.4(b)의 이미지를 생성하기 위하여 뷰잉과 라이팅에 관련된 코드 외에 텍스춰를 적용하기 위하여 추가적으로 수행해야할 OpenGL 프로그래밍에 대하여 살펴 보았다. OpenGL에서는 디폴트로는 텍스춰 매핑 계산을 하지 않기 때문에 glEnable(GL_TEXTURE_2D); 문장을 수행시켜 명시적으로 그러한 계산이 일어나도록 해야 한다. 물론 그러한 기능을 정지하려면 glDisable(GL_TEXTURE_2D); 문장을 수행시키면 된다. 요약을 하면 텍스춰 매핑을 하기 위해서는,

1. 사용하려는 텍스춰 이미지를 텍스춰 메모리에 올려 놓고 (예를 들어 glTexImage2D(*) 함수 사용),

2. 텍스춰 좌표 값이 주어졌을 때 텍스춰 이미지로부터 그에 대응되는 텍스춰 색깔을 계산하는 방식을 결정하는 텍스춰 상태 값을 적절히 설정을 하고 (glTexParameter*(*) 함수 사용),

3. 텍스춰 색깔과 물체의 기반 색깔을 어떻게 합성을 할 것인가를 결정한 후

(glTexEnv*(*) 함수 사용),

4. 기하 물체를 그릴 때 각 꼭지점에 대하여 텍스춰 좌표를 적절히 설정을 해주면 된다 (glTexCoord2*(*) 함수를 사용하거나 자동 생성되도록 설정).

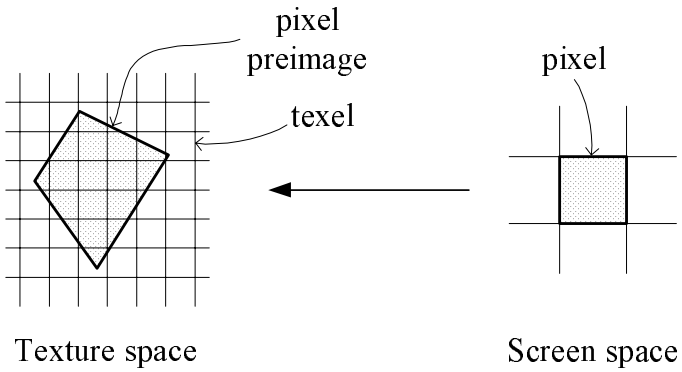



그림 5.9: 텍스춰 필터링

# 제 4 절 텍스춰 필터링

## 4.1 텍스춰의 확대와 축소, 그리고 필터링

앞에서 다면체 모델에 대한 텍스춰 매핑 과정을 개념적으로 살펴보았는데(448쪽의 그림 5.2 참조), 여기서의 핵심은 기하 물체에 텍스춰를 입혀 렌더링을 하면, 래스터화 과정에서 물체를 구성하는 기하 프리미티브가 투영되는 영역에 존재하는 각 픽셀에 대하여, 픽셀이라는 작은 창문을 통하여 보이는 텍스춰 이미지 영역의 색깔을 계산하여 이를 텍스춰 색깔로 사용한다는 것이다. 개념적으로는 단순하나 이 과정을 구현하는데 있어 세심한 주의를 하지 않으면, 계산 시간이나 결과 이미지에 여러 가지 문제가 발생한다.

그림 5.9는 텍스춰 매핑의 과정을 요약한 그림인데, 스크린 공간의 화소, 즉 픽셀이 주어졌을 때 그에 대응이 되는 텍스춰 공간에서의 프리 이미지를 도시하고 있다[8]. 픽셀의 텍스춰 색깔을 계산할 때, 가장 이상적인 방법은 픽셀의 프리 이미지 영역 안에 들어오는 텍셀들을 정확하게 찾아낸 후, 그것들의 색깔을 사용하여 텍스춰 색깔을 계산하는 것이다. 이를 위해서 몇 가지 사항을 고려해야 하는데, 그중

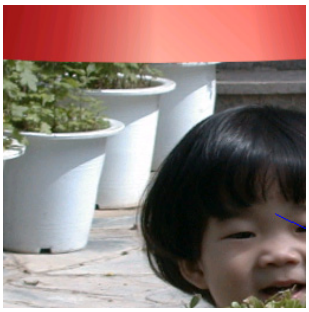
[8] 이 장에서는 화면의 화소와 텍스춰 이미지의 화소를 구별하기 위하여, 전자에 대해서는 픽셀, 그리고 후자에 대해서는 텍셀이라는 용어를 사용하겠다.

하나가 텍스춰 필터링(texture filtering) 문제로서, 과연 픽셀를 통하여 보이는 텍스춰 이미지의 영역을 대표하는 색깔을 어떻게 구할 것인가 하는 것이다. 기하 물체에 텍스춰를 입힐 때 물체가 화면에 투영되는 영역의 상대적인 크기에 따라 픽셀을 통하여 한 개의 텍셀이 보일 수도 있지만, 반면에 여러 개의 텍셀이 보일 수도 있다. 이러한 상황을 이해하기 위하여 그림 5.10(a), (b), (c)를 보자. 이 세 개의 그림은 $512 \times 256$의 해상도를 가지는 텍스춰 이미지를 물주전자에 입힌 후, 각각 화면에서 서로 다른 거리에 놓고 원근 투영을 사용하여 렌더링을 한 것이다. 세 이미지 모두 해상도가 $256 \times 256$이 되도록 하였는데, 그림 (a)는 물체를 화면에 아주 가까이 놓은 것이고, 반면에 그림 (c)는 비교적 멀리 놓고 렌더링을 하였다.

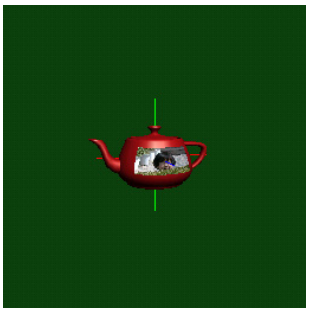
이 때 텍스춰가 투영되는 영역에 있는 픽셀들을 통해서 보이는 텍스춰 이미지의 내용을 상상해보자. 그림 (a)의 경우에는 텍스춰 이미지의 입장에서 보면 한 개의 픽셀을 통하여 보이는 영역이 매우 작고, 반대로 그림 (c)의 경우에는 상당히 큰 영역이 픽셀을 통하여 보이게 될 것이다. 다시 말해서 전자는 텍스춰 이미지가 확대가 되어 텍셀의 크기가 픽셀의 크기보다 큰 경우이고, 후자는 픽셀의 크기가 텍셀보다 커서 한 개의 픽셀을 통하여 여러 개의 텍셀이 보이는 경우이다. 이 두 가지 상황이 그림 5.10(d)와 (e)에 도시되어 있는데, 각 경우를 텍스춰의 확대(magnification)와 축소(minification)라 부른다. 그림 (b)는 픽셀의 프리 이미지의 크기와 텍셀의 크기가 거의 동일한 상황의 렌더링 결과로서, 이를 경계로 확대와 축소 두 가지 상황으로 나누게 된다.

그림 (d)와 (e)에서 굵은 선으로 표시된 사각형은 텍스춰 공간으로 역변환된 픽셀의 프리 이미지이다. 물론 일반적으로 프리 이미지는 정사각형이 아니고, 또한 텍셀과 동일한 방향으로 정렬이 되지는 않지만, 일단은 설명의 편의상 그렇다고 가정하자. 바로 앞에서도 말한 바와 같이 픽셀에 대한 텍스춰 색깔을 구하기 위하여 가

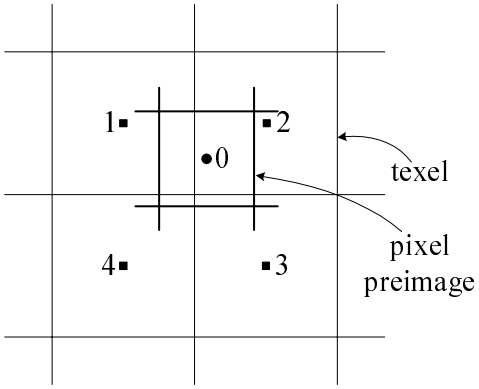
(a) 확 대



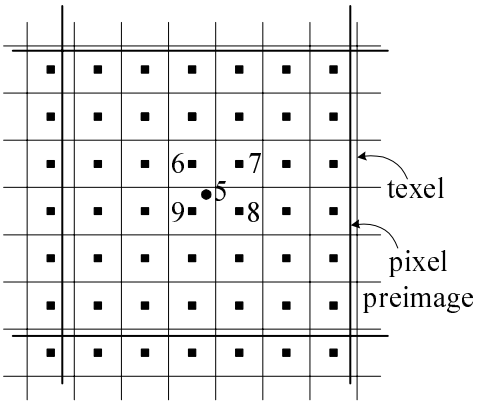
(b) 확대와 축소의 경계



(c) 축 소




(d) 확대 상황




(e) 축소 상황

그림 5.10: 텍스춰의 확대와 축소

장 이상적인 방법은 픽셀, 즉 프리 이미지 영역에 대한 적분을 통하여 텍스춰 이미지의 색깔을 계산을 하는 것이다. 물론 이는 상당한 계산량을 요구하므로 특히 실시간 렌더링 시스템에서는 적합한 방법이라 할 수가 없다.

이 시점에서 샘플링과 필터링의 문제를 텍스춰 매핑의 관점에서 다시 살펴보자. 실세계에서는 물체에 입히는 텍스춰 이미지는 유한 개의 텍셀로 이루어진 이산적인 이미지가 아니라 연속적인 내용을 가지는 이미지이다. 또한 화면도 유한 개의 픽셀로 구성된 이산적인 공간이 아니라 연속 공간에서의 사각형으로 정의가 된다. 하지만 이산적인 개념에 기반한 래스터 그래픽스 시스템에서는 픽셀에 해당하는 유한 개의 영역에 대하여 색깔을 구해 화면의 내용을 표현한다. 마찬가지로 텍스춰 이미지도 텍셀이라고 하는 유한 개의 영역에 대한 색깔들로 텍스춰의 내용을 표현한다. 따라서 텍스춰 매핑에서의 샘플링과 필터링은 두 개의 이산적인 이미지간의 문제라고 할 수가 있다. 기본적으로 3차원 렌더링 시스템에서는 연속 공간인 3차원 물체 공간과 이산 공간인 2차원 화면 공간간의 샘플링과 필터링의 문제를 처리하여야 한다. 이러한 성질을 가지는 3차원 렌더링 구조에 텍스춰 매핑을 실시간적으로 처리하기 위해서는 두 이산 공간간의 샘플링과 필터링 문제를 추가적으로 고려해야 하기 때문에 문제가 복잡해지고, 그 결과 심각한 앨리어스가 발생할 가능성이 높아진다. 이러한 이유로 텍스춰 매핑 과정에서의 필터링 문제가 중요하게 고려가 되어왔다.

그러면 텍스춰 매핑의 구현 시 널리 쓰이는 기본적인 샘플링과 필터링 방법에 대하여 알아보자. 보통 이산적인 텍스춰 이미지에 대해서는 고정된 지점, 즉 텍셀의 중심에 대해서만 샘플링을 한다. 반면에 픽셀에 대해서는 픽셀의 중심 주변의 어느 지점에 대해서도 샘플링을 할 수가 있다. 가장 간단한 방법은 픽셀의 중심에 대하여 대응이 되는 텍스춰 공간에서의 지점을 구한 후, 그 위치에서 중심이 가장 가까운

텍셀의 색깔을 텍스춰 색깔로 사용을 하는 것이다. 이러한 방식을 최근 필터(nearest neighbor filter)를 사용한 필터링 방법이라 한다. 예를 들어 그림 5.10(d)와 (e)에서 픽셀의 중심에 대응이 되는 0번과 5번 지점에 가장 가까운 2번과 9번 지점에 해당하는 텍셀의 색깔을 픽셀의 텍스춰 색깔로 사용한다. 물론 최근 필터를 사용하면 인접한 픽셀들간의 색깔이 급격하게 변할 가능성이 많기 때문에 보통 눈에 거슬리는 이미지를 생성한다.

또 다른 필터링 방법으로서 이선형 필터(bilinear filter)가 자주 사용된다. 이는 텍스춰 공간에서 픽셀의 중심에 대응이 되는 지점을 둘러싸는 네 개의 텍셀 중심을 찾아, 그림 5.11에서와 같이 각 텍셀 색깔에 대하여 이선형 보간을 통하여 텍스춰 색깔을 구하는 방법이다. 따라서 이선형 필터는 픽셀에 대하여 네 개의 지점에 대하여 샘플링을 하는 셈이고, 각 지점의 텍셀 색깔에 대한 선형 보간을 통하여 가중 평균을 구하는 방식을 취하는 필터이다.

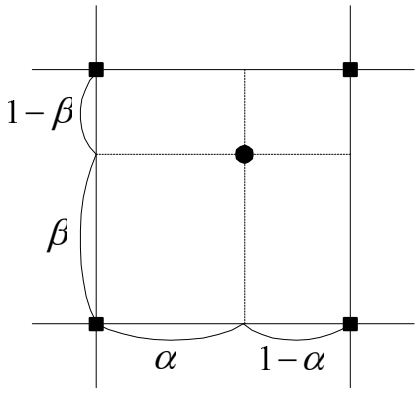



그림 5.11: 이선형 보간

일반적으로 이선형 필터는 확대 상황에서 충분히 좋은 결과를 생성하지만, 축소 상황에서는 문제가 발생할 소지가 많다. 즉 픽셀에 대한 프리 이미지의 크기와 텍셀의 크기가 비슷하다면 이선형 필터는 잘 작동을 할 것이다. 그러나 축소의 정도가 심해지면, 예를 들어 그림 5.10(e)에서와 같이 프리 이미지 영역에 많은 수의 텍셀이 포함이 되면, 이선형 필터를 사용을 한다해도 6, 7, 8, 9 지점의 텍셀 값만 사용을 하기 때문에 프리 이미지 안의 다른 텍셀에 대한 정보를 잃어 버릴 것이다. 따라서 인접한 픽셀간의 텍스춰 색깔에 급격한 변화가 올 것이고, 그 결과 최근 필터

를 사용할 때와 같이 상당히 거친 이미지를 생성하게 된다.

이러한 문제를 해결하기 위해서 구현할 수 있는 한 가지 방법은 픽셀의 프리 이미지 영역에 들어오는 모든 텍셀들의 색깔의 평균을 구하는 것이다. 이는 가장 이상적인 픽셀 영역에 대한 적분 계산을 잘 근사화할 수 있는 방법이나, 한 픽셀 안에 들어오는 텍셀의 개수가 많아질수록 계산 시간이 증가한다. 그뿐만 아니라 상황에 따라 텍스춰 필터링의 시간이 들쑥날쑥하게 된다는 문제가 발생하는데, 항상 일정한 응답 시간을 요구하는 실시간 계산에 있어 좋은 방법은 아니다. 실시간 렌더링 시스템에서는 주로 삼선형 필터(trilinear filter)의 일종인 밉매핑 기법을 사용하여 일정한 시간안에 텍셀들의 평균을 구하는 계산을 수행하는데, 이에 대해서는 뒤에서 자세하게 살펴보겠다.

## 4.2 OpenGL에서의 텍스춰 필터링

필터링 기법에 대해서는 컴퓨터 그래픽스 분야에서뿐만 아니라 영상 처리 분야에서 많은 연구가 진행이 되어 왔다. 그 결과 다양한 필터링 방법들이 개발이 되었는데, 그에 대한 자세한 내용은 이 책의 범위를 벗어나므로 여기서는 OpenGL 시스템에서 제공하는 텍스춰 필터링 방법에 대해서만 알아보겠다. 기본적으로 OpenGL에서는 최근 필터와 선형 보간에 기반한 이선형 필터와 삼선형 필터, 그리고 최근 필터와 선형 필터를 혼합한 혼성(hybrid) 필터를 제공한다[9].

텍스춰 필터링을 하기 위해서는 텍스춰 이미지에 대하여 확대와 축소 상황에 대하여 각각 사용할 필터를 지정하여야 한다. OpenGL에서는 glTexParameter*(*) 함수를 호출하여 텍스춰마다 서로 다른 필터를 설정할 수 있는데, 그러한 정보는 텍스춰 상태로 저장이 된다. 앞에서 본 표 5.2에는 사용이 가능한 두 개의 확대 필터

[9]여기서는 2차원 텍스춰 매핑에 관련된 필터링 방법에 대하여 알아보겠다. 1차원 텍스춰 매핑이나 3차원 텍스춰 매핑의 경우에는 (일)선형 필터와 사선형 필터의 사용도 가능하다.

와 여섯 개의 축소 필터가 요약이 되어 있다. 우선 확대 필터를 생각해보자. 다음과 같이 필터를 지정하면 확대의 상황에서 최근 필터가 사용이 되고,

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
```

GL_NEAREST대신에 GL_LINEAR를 사용하면 이선형 필터가 사용이 된다. 그림 5.12(a)와 (b)는 각각 물주전자 이미지에 대하여 최근 필터와 이선형 필터를 사용한 결과를 보여주고 있다. 또한 그림 (c)와 (d)는 체커 보드 텍스춰를 사용하여 두 필터를 비교하고 있는데, 이선형 필터를 사용하는 경우가 훨씬 부드러운 이미지를 생성하고 있음을 관찰할 수가 있다. 이선형 필터는 최근 필터를 사용할 때 거친 느낌의 형태로 나타나는 앨리어스를 부옇게 만듬으로써, 부드럽게 느끼게, 다시 말해서 눈에 덜 거슬리게 해준다. 그림 (c)와 (d)를 다시 보면 최근 필터를 사용할 때 체커 보드의 경계가 계단처럼 렌더링이 되어 눈에 거슬리는데, 그림 (d)에서 대응이 되는 영역을 보면 그러한 계단 현상이 부예져 있음을 볼 수가 있다. 선형 보간에 기반을 둔 필터는 주변의 색깔을 혼합하기 때문에 부드러워지는 효과가 있지만, 그러한 정도가 심해지면 렌더링이 된 이미지가 너무 부예지는 효과(blurring effect)가 나타난다. 경우에 따라서 선명한 이미지를 선호하기도 하기 때문에, 이선형 필터가 항상 최근 필터보다 좋은 결과를 낸다고 할 수는 없지만, 확률적으로 전자를 선택하는 것이 안전한 방법이라 할 수가 있다.

다음 표 5.2에 나열된 축소 필터 중 GL_NEAREST와 GL_LINEAR는 확대 필터와 마찬가지로 최근 필터와 이선형 필터에 해당한다. 물론 인자 이름으로 GL_TEXTURE_MIN_FILTER를 사용하면 되는데, 그림 5.13(a)와 (b)는 각각 축소 상황에서 최근 필터와 이선형 필터를 사용하였을 때의 결과를 보여주고 있다. 역시 이 경우에도 이선형 필터가 우수한 효과를 내고 있음을 알 수가 있다.

하지만 앞에서도 설명한 바와 같이 축소의 정도가 심해지면 이선형 필터도 최근

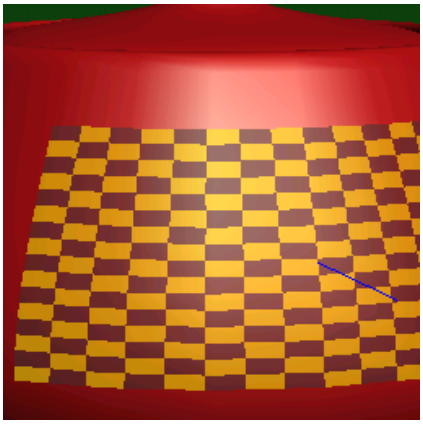
(a) 최근 필터

(b) 이선형 필터

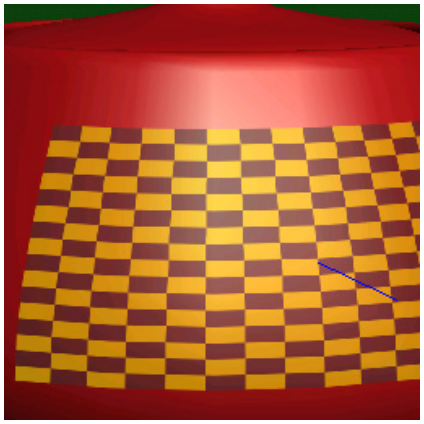
(c) 최근 필터

(d) 이선형 필터

그림 5.12: 확대 상황에서의 필터링



(a) 최근 필터



(b) 이선형 필터

(c) 이선형 필터

(d) 삼선형 필터

그림 5.13: 축소 상황에서의 필터링

필터와 동일한 문제를 야기한다. 그림 (c)를 보면 물주전자가 화면에서 더 멀리 떨어졌을 때 역시 이선형 필터도 심한 앨리어스를 생성하고 있음을 알 수가 있다. 그림 (d)는 다음 절에서 설명할 밉매핑 기법을 통하여 생성한 결과인데, 텍스춰가 이선형 필터를 사용할 때보다 부드럽게 입혀져 있음을 관찰할 수가 있다. 사실 정지영상인 그림 (c)와 (d)만 놓고 비교하면 삼선형 필터에 기반한 밉매핑 방법이 약간 더 부드러울 뿐 훨씬 우수한 방법인지에 대한 확신을 가질 수가 없다. 하지만 물주전자를 움직여 보면 실시간 텍스춰 매핑에서 삼선형 필터의 중요성을 이해할 수가 있다.

이를 위하여 예제 프로그램 5.B를 수행시켜보자. 마우스 커서를 윈도우 안으로 옮긴 후 오른쪽 마우스 버튼을 누르면 사용 가능한 메뉴가 뜨게 된다. 이 때 TEXTURE를 선택한 후 CHECKER128 텍스춰를 선택하자. 다음 FILTER_MODE → MIN_FILTER → GL_LINEAR를 통하여 축소 필터로 이선형 필터를 선택하자. 다음 마우스를 윈도우 위쪽으로 움직여 물체를 충분히 멀리 보낸 후 ANIMATION_MODE에서 YES를 선택하여 회전을 시키자. 텍스춰가 입혀진 부분을 자세히 보면 물주전자가 회전함에 따라 텍스춰가 출렁거리는 현상을 발견할 수 있을 것이다. 다음 축소 필터를 GL_LINEAR_MIPMAP_LINEAR를 눌러 삼선형 필터를 사용하는 밉매핑을 선택한 후, 그 결과를 보자. 텍스춰가 좀 필요 이상으로 부드럽게 입혀지는 듯한 느낌은 있으나, 물체가 회전을 해도 상당히 일관되게 텍스춰 매핑이 이뤄지고 있으며, 이선형 필터처럼 눈에 거슬리는 현상이 사라짐을 관찰할 수 있다. 축소 필터 이외에도 확대 필터의 선택에 따른 차이를 실제로 관찰해보기 바란다.

한 가지 재미있게 실험할 수 있는 것이 있는데, 확대 필터를 최근 필터로 설정하고, 축소 필터를 삼선형 필터로 선택을 한 후 물체를 멀리서 가까이 천천히 확대시키면서 텍스춰가 입혀진 부분을 유심히 살펴보자. 한 순간부터 부드럽던 이미지가

거칠어 지는데, 바로 그 시점이 텍스춰의 축소에서 확대로 넘어가는 경계가 된다.

# 제 5 절 밉매핑과 삼선형 보간

## 5.1 밉매핑의 원리

밉매핑(mip-mapping)의 밉(mip)은 라틴어로 *Multum In Parvo*라는 어구에서 나온 것인데, 이는 조그마한 장소에 많은 것을 집어 넣는다는 것을 뜻한다. 이 방법은 1983년에 윌리엄스가 SIGGRAPH에 발표한 논문에서 처음 제안된 방법으로서[10], 특히 OpenGL과 같은 실시간 렌더링 시스템에서 거의 표준처럼 쓰이고 있으며, 최근의 대부분의 그래픽스 가속기에서 밉매핑 방법이나 그의 변형들이 하드웨어적으로 구현이 되고 있다.

밉매핑은 컴퓨터 그래픽스 분야에서 실시간적인 성능을 내기 위하여 미리 전처리를 통하여 필요한 데이터를 만들어 놓고, 런타임 시에 그러한 정보를 바탕으로 빠르게 계산을 하는 전형적인 전처리 기법(pre-calculation techniques) 중의 하나이다. 또한 대표적인 다중 해상도 기법(multi-resolution techniques) 중의 하나로서, 이름이 의미하는 바와 같이 동일한 텍스춰 이미지에 대하여 서로 다른 해상도를 가지는 이미지들을 미리 만들어 놓고 적절한 처리를 통하여 일정한 시간 안에 삼선형 필터링을 수행할 수 있도록 해준다.

주어진 텍스춰 이미지에 대하여 $s$와 $t$축 각 방향으로 해상도를 반으로 줄인 이미지를 생각해보자. 예를 들어 해상도가 $256 \times 256$인 이미지에 대하여 인접한 텍셀을 네 개씩 묶어 그 평균 값을 취해 $128 \times 128$의 해상도를 가지는 이미지를 생성할 수가 있다. 이 새로운 텍스춰 이미지는 원래의 이미지의 해상도를 반으로 떨어뜨린

[10] L. Williams. "Pyramidal Parametrics," *ACM SIGGRAPH* '83, pp. 1-11, 1983.

(a) 256 × 256 텍스춰에 대한 밉맵

(b) 레벨 3 이미지

그림 5.14: 밉맵의 구성 예

것으로서, 한 텍셀의 한 변의 길이가 두 배로, 따라서 면적이 네 배로 증가하게 된다. 이러한 과정을 반복으로 하면 $64 \times 64$, $32 \times 32$, $\cdots$, $1 \times 1$ 등의 해상도의 이미지를 만들 수가 있는데, 제일 마지막의 $1 \times 1$ 이미지는 원래 텍스춰의 모든 텍셀 값들의 평균을 낸 색깔을 나타낸다.

원래의 텍스춰 이미지의 레벨을 0이라 하고 매번 해상도를 반으로 떨어뜨릴 때마다 레벨이 1씩 증가한다고 하면 $256 \times 256$ 텍스춰 이미지에 대해서 레벨 8까지의 텍스춰가 생성이 된다. 이렇게 동일한 텍스춰에 대하여 여러 레벨의 텍스춰 이미지를 모아 놓은 것을 밉맵(mip-map)이라고 한다. 그림 5.14는 $256 \times 256$의 해상도를 가지는 텍스춰에 대하여 밉맵을 구성한 예를 보여주고 있다. 물론 밉맵을 사용하려면, 필요한 텍스춰 메모리의 크기가 증가하게 되는데, 예를 들어 한 텍셀 당 네 바이트를 사용하는 $256 \times 256$ 텍스춰에 대하여 밉맵을 구성할 때 추가적으로 어느 정도의 메모리가 필요한 지를 계산해보기 바란다.

이제 밉매핑의 기본 원리를 알아보기 위하여, 그림 5.10(e)의 축소 상황을 다시 생각해보자. 지금 픽셀의 프리 이미지 안에 들어오는 모든 텍셀들의 평균 색깔을 구하려고 하는데, 여기서 중요한 사실은 실시간 렌더링을 위하여 프리 이미지의 면

적, 즉 텍셀의 개수에 상관 없이 일정한 시간 안에 원하는 색깔을 구해야 한다는 것이다. 일반적으로 이러한 상황이 발생할 확률은 거의 없지만 그림 5.15와 같은 경우를 생각해보자. 이 경우에서는 픽셀의 프리 이미지 안에 정확하게 16개의 텍셀이 들어 오는데, 밉맵이 구성되어 있다면 레벨 2 이미지의 해당 지점을 한 번만 액세스하여 이들의 평균 색깔을 구할 수가 있다. 즉 레벨 0의 텍스춰 이미지를 16번 엑세스하여 그것들의 평균을 구하는 것이 아니라, 미리 계산 해놓은 밉맵의 적절한 장소를 한 번만 액세스하여 텍스춰 색깔을 구할 수가 있다.

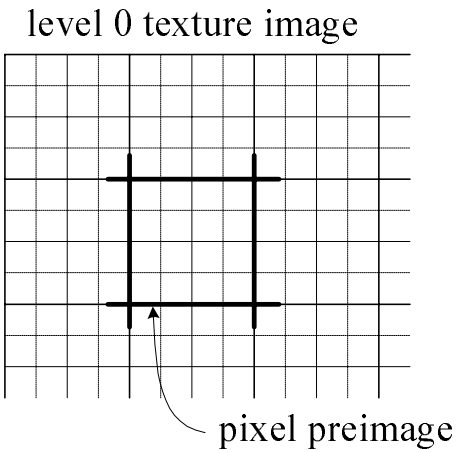



그림 5.15: 밉매핑의 기본 원리

하지만 일반적으로 픽셀의 프리 이미지의 영역이 어떤 한 레벨의 텍셀 영역과 정확하게 일치하는 일은 거의 일어나지 않는다. 사실 프리 이미지의 모양이 정사각형이 되는 경우는 상당히 예외적인 상황이다. 일단은 설명을 간단하게 하기 위하여 텍스춰 공간에서 중심이 $(s,t)$이고 한 변의 길이가 $d$인 정사각형의 형태를 가지는 프리 이미지에 대해서만 고려하자. 또한 원래 주어진 텍스춰, 즉 레벨 0인 텍스춰의 해상도가 $2^n \times 2^n$이라고 가정하자. 레벨 값이 얼마이건 밉맵을 구성하는 각 텍스춰는 텍스춰 공간에서 $[0,1] \times [0,1]$ 영역에 대응이 된다. 따라서 텍스춰 공간에서 레벨 0인 텍스춰의 텍셀의 한 변의 길이는 $\frac{1}{2^n}$이고, 레벨 $k$인 텍스춰의 경우에는 $d_k = \frac{1}{2^{n-k}}$이 된다.

한 변의 길이가 $d$인 프리 이미지에 대한 텍스춰 색깔을 구하기 위해서는 텍셀의 변의 길이가 $d$와 가장 가까운 레벨의 텍스춰를 선택하여야 하는데, 일반적으로는 두 인접한 레벨의 사이에 들어오게 된다. 따라서 $\frac{1}{2^{n-k_0}} < d < \frac{1}{2^{n-(k_0+1)}}$를 만족시켜

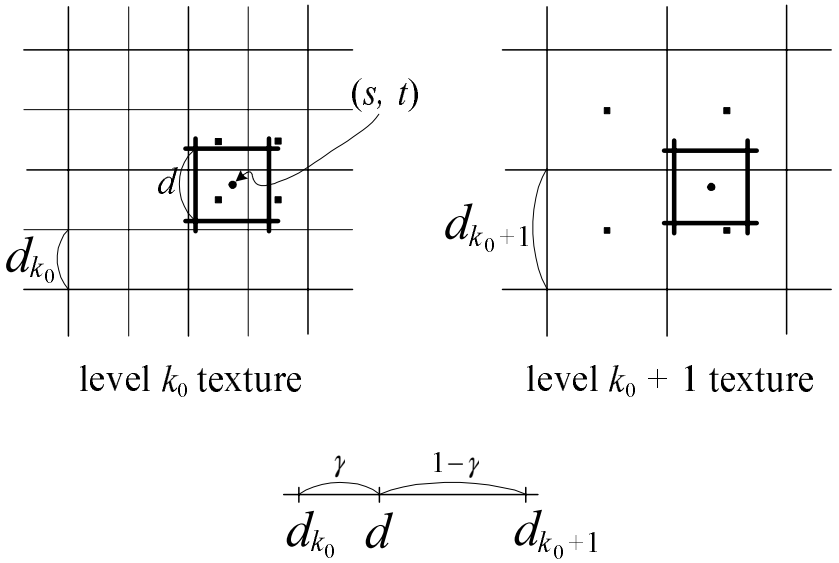



그림 5.16: 삼선형 보간을 통한 밉매핑

주는 $k_0$에 대하여 레벨이 $k_0$와 $k_0+1$인 텍스춰 이미지를 사용하여 텍스춰 색깔을 구하게 된다. 그림 5.16은 밉매핑의 계산 과정을 보여주고 있다. 우선 각각 레벨이 $k_0$와 $k_0+1$인 텍스춰를 사용하여 프리 이미지의 중심 $(s,t)$에 대하여 이선형 필터를 사용하여 두 레벨으로부터의 텍스춰 색깔 $C_{k_0}$와 $C_{k_0+1}$을 구한다. 다음 이 두 색깔에 대하여 다시 한번 선형 보간을 적용하여 최종 텍스춰 색깔을 구하는데, 이 두 색깔을 혼합하는데 사용하는 비율 $\gamma$로 자연스럽게 $\frac{d-d_{k_0}}{d_{k_0+1}-d_k}$를 사용할 수 있다. 즉 $\gamma = 2^{n-k_0} \cdot d - 1$에 대하여 $C = (1-\gamma)C_{k_0} + \gamma C_{k_0+1}$과 같이 텍스춰 색깔을 구할 수가 있다.

밉매핑 과정은 그림 5.17에서와 같이 각 레벨에서 네 개씩 가져온 총 여덟 개의 데이터를 3차원 공간에 배치하고, $(s,t)$와 $d$에 대응되는 지점에 대하여 선형 보간을 반복적으로 적용하여 보간을 하는 삼선형 보간이라 할 수 있다. 따라서 밉매핑 필터를 삼선형 보간이라 하는 것이다. 지금까지 밉매핑 과정을 약간 단순화시켜 살펴보았는데, 소프트웨적으로 또는 하드웨어적으로 구현되는 대부분의 밉매핑 과정은

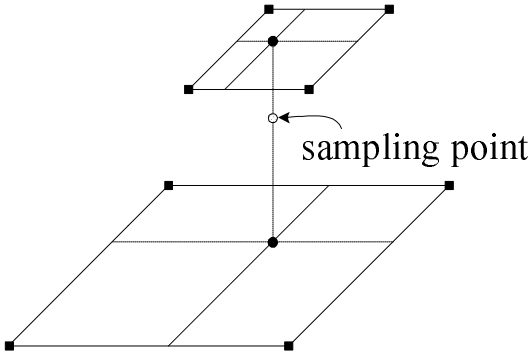



그림 5.17: 삼선형 보간

구현상 약간의 차이가 있을 수 있으나 기본적으로 이러한 방식에 기반을 두고 있다.

## 5.2 OpenGL에서의 밉매핑과 그 변형

OpenGL에서 밉매핑 필터를 사용하려면 우선 다음과 같이 축소 필터를 선택하여야 한다.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);
```

다음 원래의 텍스춰 이미지에 대하여 각 레벨에 해당하는 텍스춰를 만들어 명시적으로 텍스춰 메모리에 올려주어야 한다. 예를 들어 $256 \times 256$ 텍스춰에 대하여 밉맵을 만들어 배열 `texture[i]`에 레벨 `i` 텍스춰 이미지에 대한 포인터를 저장하였다면 다음과 같이 밉맵을 텍스춰 메모리에 탑재할 수가 있다.

```
size = 256;
for (i = 0; i <= 8; i++) {
  glTexImage2D(GL_TEXTURE_2D, i, GL_RGBA, size, size,
               0, GL_RGBA, GL_UNSIGNED_BYTE, texture[i]);
  size /= 2;
}
```

OpenGL에서는 각 방향으로의 크기가 2의 멱급수이기만 하면 직사각형의 텍스춰 이미지를 허용하는데, 이 경우 마찬가지로 $1 \times 1$의 해상도에 이를 때까지 반복

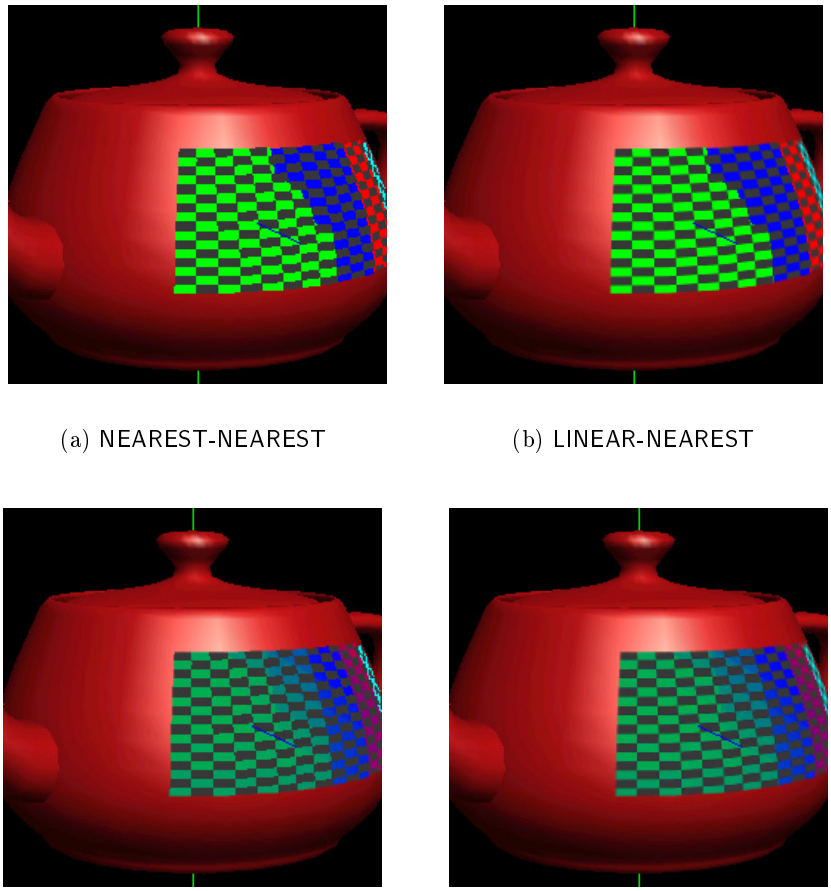
해서 텍스춰를 생성하여야 한다. 예를 들어 $128 \times 32$의 텍스춰에 대하여 밉맵을 구성하려면 $64 \times 16$, $32 \times 8$, $\cdots$, $8 \times 2$, $4 \times 1$, $2 \times 1$, $1 \times 1$과 같이 텍스춰 이미지를 만들면 된다. 사실 주어진 텍스춰에 대하여 밉맵을 구성하는 각 레벨의 텍스춰를 만들어 일일이 텍스춰 메모리에 올려주는 것은 약간은 귀찮은 일이다. OpenGL에서는 이러한 작업을 쉽게 할 수 있도록 int gluBuild2DMipmaps(GLenum target, GLint internalFormat, GLint width, GLint height, GLenum format, GLenum type, void *texels); 함수를 제공하는데, 이 함수를 호출하면 자동적으로 texels에 저장되어 있는 width $\times$ height 해상도의 텍스춰 이미지에 대하여 밉맵을 만들어 텍스춰 메모리에 올려준다. 따라서 위의 코드는 다음과 같은 문장으로 대체할 수가 있다.

```
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 256, 256,
                  GL_RGBA, GL_UNSIGNED_BYTE, texture[0]);
```

이렇게 축소 필터를 밉맵 필터로 선택을 하고, 필요한 밉맵을 텍스춰 메모리에 올려주면 텍스춰 매핑 계산을 수행할 때 밉매핑 방법을 사용하여 텍스춰 색깔을 구해준다.

OpenGL에서는 밉맵 필터 외에도 밉맵을 사용하는 세 가지의 변형 필터를 제공한다. GL_NEAREST_MIPMAP_NEAREST 와 GL_LINEAR_MIPMAP_NEAREST 필터는 두 개의 인접한 레벨의 텍스춰를 사용하는 밉매핑과는 달리, $d$ 값에 따라 더 가까운 레벨의 텍스춰 이미지를 선택하여 텍스춰 색깔을 구한다. 전자는 그 텍스춰 안에서 최근 필터를 사용하는 필터이고, 후자는 이선형 필터를 사용하는 필터이다. 반면에 GL_NEAREST_MIPMAP_LINEAR 필터는 밉맵 필터와 같이 두 개의 텍스춰를 사용하지만 각 레벨의 텍스춰에 대하여 이선형 필터가 아니라 최근 필터를 사용하는 필터이다. 따라서 밉맵 필터와 함께 모두 네 가지의 조합이 가능한데, 각자 계산량과 필터 성능이 다르기 때문에 상황에 맞는 필터를 선택하여야 한다.

그림 5.18은 이 네 가지 필터를 사용할 때 어떠한 방식으로 필터링 계산이 수행



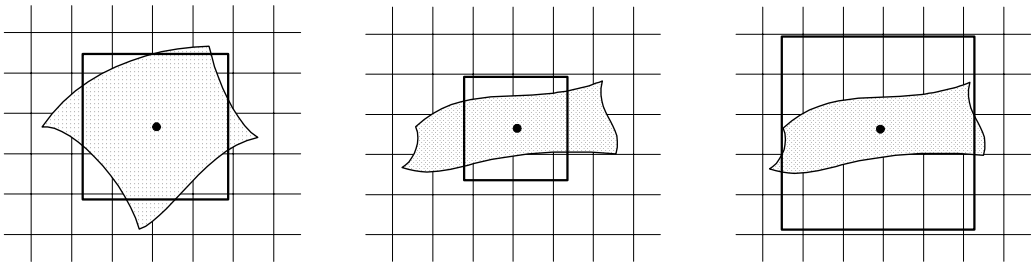
(a) NEAREST-NEAREST (b) LINEAR-NEAREST

(c) NEAREST-LINEAR (d) LINEAR-LINEAR

그림 5.18: 네 가지 밉맵 사용 필터의 비교(밉맵 레벨: 노란색 = 0, 초록색 = 1, 파란색 = 2, 빨간색 = 3, 하늘색 = 4)

이 되는가를 시각적으로 보여주고 있다. 이 이미지들은 예제 프로그램 5.B를 수행시켜 MULTILEVEL 텍스춰를 선택하여 렌더링을 하고 있는 모습이다. 여기서는 밉맵을 구성할 때 레벨이 0, 1, 2, 3, 그리고 4에 대하여 각각 색깔이 노란색, 초록색, 파란색, 빨간색, 하늘색인 텍스춰를 사용하였다. 축소 필터를 밉맵을 사용하는 네 가지 필터 중의 하나로 선택을 한 후, 물체를 크게 했다가 천천히 작게 해보면 물체의 크기에 따라 선택이 되는 텍스춰의 레벨을 파악할 수가 있다.

우선 한 개의 텍스춰만 사용하는 경우에 해당하는 그림 (a)와 (b)는 물주전자의



(a) 경우 1 (b) 경우 2 (c) 경우 3

그림 5.19: 정사각형을 통한 프리 이미지의 근사

표면 상에서 사용하는 밉맵의 레벨이 분명하게 구별이 됨을 알 수가 있다. 다만 각 레벨의 텍스춰 안에서 각각 최근 필터와 이선형 필터를 사용하는 것이 차이점인데, 그러한 효과를 분명히 관찰할 수가 있다. 물주전자의 옆으로 갈 수록 레벨이 증가하는데 이는 비스듬하게 볼수록 한 개의 픽셀 안으로 떨어지는 텍셀의 개수가 증가하기 때문이다. 그림 (c)와 (d)는 두 개의 인접한 텍스춰를 사용하는 경우로서 두 레벨으로부터의 색깔이 부드럽게 혼합이 되고 있음을 알 수가 있다.

## 5.3 밉맵 레벨의 계산

앞에서 밉매핑에 대하여 설명을 할 때 처리하려고 하는 픽셀의 프리 이미지가 텍스춰 공간에서 중심이 $(s,t)$이고 한 변의 길이가 $d$인 정사각형이라고 가정을 하였었다. 그러나 실제로 프리 이미지가 이렇게 텍스춰 좌표 공간의 좌표축에 정렬이 된 정사각형의 형태를 가지는 경우는 거의 없고, 일반적으로 프리 이미지는 임의의 형태의 임의의 방향성을 가지는 영역이 된다. 따라서 정사각형 형태의 프리 이미지를 가정하는 일반적인 밉매핑 계산 과정에서는 가능한 한 임의의 형태를 가지는 프리 이미지를 잘 근사화해주는 정사각형을 찾아, 그 크기에 따른 밉맵 레벨을 구해야 한다.

그림 5.19는 텍스춰 공간에서 프리 이미지를 근사화해주는 정사각형의 예를 보여주고 있다. 여기서 이 정사각형의 중심은 해당 픽셀의 중심에 대응이 되는 지점인데, 문제는 정사각형의 한 변의 길이 $d$를 얼마로 할 것인가 하는 것이다. 한 가지 방법은, 별로 실용적인 방법은 아니지만, 프리 이미지와 면적이 같은 정사각형을 사용하는 것이다. 즉 프리 이미지의 면적이 $A$라면 $d = \sqrt{A}$인 정사각형을 선택하는 것인데, 그림 (a)에서와 같이 프리 이미지가 정사각형에 의해 비교적 잘 근사화가 되는 경우라면 프리 이미지의 형태를 정사각형으로 가정을 해도 별 문제가 발생하지 않을 것이다. 반면에 그림 (b)의 경우에서처럼 프리 이미지의 모양이 정사각형의 모양과 동떨어진 경우라면 그러한 가정은 텍스춰 매핑의 결과에 문제를 야기할 것이다.

또 다른 방법은 그림 (c)에서와 같이 텍스춰 공간에서 $s$와 $t$축 각 방향으로 프리 이미지가 차지하는 영역의 폭 중 큰 쪽의 길이를 정사각형의 변의 길이로 사용하는 것이다. 두 가지 방법 중 어떤 것을 을 사용하건 텍스춰 매핑의 결과에 문제가 발생한다. 전자의 경우처럼 면적에 기반한 방법은 그림 (b)와 같은 경우에 정사각형이 프리 이미지의 많은 부분을 포함하지 못하기 때문에, 텍스춰 정보를 충분히 사용하지 못하게 되고, 따라서 최근 필터를 사용할 때와 같은 앨리어스가 발생한다. 반면에 그림 (c)에서와 같은 정사각형을 사용하여 밉맵 필터를 적용하면, 프리 이미지 영역 밖의 텍스춰 정보까지 사용를 하기 때문에 결과 이미지가 필요 이상으로 부드러워지는, 다시 말해서 선명도가 떨어지는 결과가 발생한다. $d$ 값을 어떠한 방식으로 추정할 지는 밉매핑의 구현 방법에 따라 다른데, 일반적으로 전자와 같은 앨리어스를 발생시키는 방법보다는 후자와 같은 방법을 택한다.

이제 $d$ 값을 구하는 방법의 한 가지 예를 통하여 이에 대한 이해를 높여보자. 그림 5.20의 왼쪽에는 스크린 공간인 윈도우 좌표계에서 좌표가 $(x, y)$인 픽셀이 주어

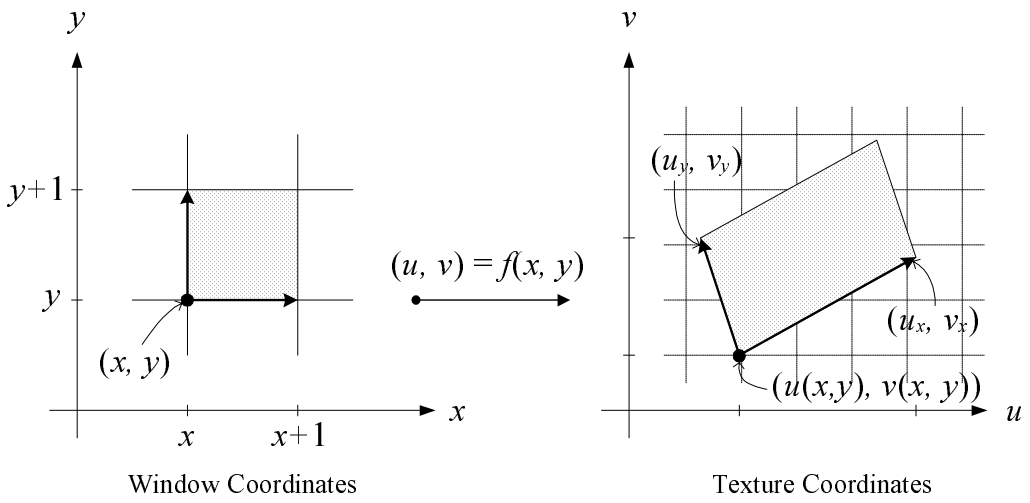



그림 5.20: 밉맵 레벨의 계산

져 있는데[11], OpenGL에서는 이 점에 대한 텍스춰 좌표 $(s,t) = (s(x,y), t(x,y))$가 텍스춰 공간의 정규화된 $[0,1] \times [0,1]$ 영역에 대하여 정의가 된다. 지금 여기서는 정규화된 텍스춰 공간에서가 아니라, 레벨 0 텍스춰 이미지 안에서 한 개의 텍셀의 폭을 단위 길이로 하는 새로운 텍스춰 공간에서 $d$ 값을 구하려 한다. 이 경우 $d$ 값은 레벨 0인 텍스춰 이미지 안에서 프리 이미지에 해당하는 정사각형이 몇 개의 텍셀의 폭을 가지는가에 대한 정보를, 그리고 $\log_2 d$ 값은 밉맵의 이미지 중 사용할 텍스춰의 레벨 값에 대한 정보를 제공한다. 이를 위하여 레벨 0의 텍스춰 이미지의 해상도가 $2^n \times 2^m$이라고 할 때, $u = u(x,y) \equiv 2^n \cdot s(x,y)$와 $v = v(x,y) \equiv 2^m t(x,y)$와 같이 단위 길이가 한 개의 텍셀에 해당하는 새로운 텍스춰 좌표 공간을 생각하자.

지금 윈도우 좌표와 새로운 텍스춰 좌표간에는 $(u,v) = f(x,y)$와 같은 함수 관계가 존재하는데, $u$와 $v$를 각각 $x$와 $y$에 대하여 편미분한 값을 $u_x = u_x(x,y) = \frac{\partial u}{\partial x}$, $u_y = u_y(x,y) = \frac{\partial u}{\partial y}$, $v_x = v_x(x,y) = \frac{\partial v}{\partial x}$, $v_y = v_y(x,y) = \frac{\partial v}{\partial y}$라 하자. 이 때 두 그래디언트 $(u_x, v_x)$와 $(u_y, v_y)$는 각각 윈도우 좌표계에서 $x$와 $y$ 좌표가 단위 길이만큼 변

[11] 여기서도 편의상 윈도우 좌표계에 대한 첨자 $wd$를 생략하자.

할 때의 텍스춰 좌표계에서의 변화량에 대한 정보를 제공한다. 윈도우 좌표계에서 한 픽셀은 길이가 1인 정사각형이므로, 이 값들로부터 픽셀에 대응되는 텍스춰 공간에서의 프리 이미지의 크기에 대한 정보를 얻을 수가 있다. 바로 앞에서도 언급한 바와 같이 비록 이미지가 좀 부예지더라도, 프리 이미지 전체를 포함하는 정사각형에 대한 밉맵 필터를 사용하는 것이 더 안전하기 때문에, $x$와 $y$의 함수인 $d$ 값은 보통 다음과 같이 계산을 한다.

$$d(x,y) = \max(\sqrt{u_x^2 + v_x^2}, \sqrt{u_y^2 + v_y^2}) \tag{5.1}$$

이 방법에서는 두 그래디언트 벡터 중 길이가 더 긴 것의 크기를 $d$ 값으로 사용을 하는데, 이는 자연스러운 선택이라 하겠다. 지금까지 윈도우 좌표와 텍스춰 좌표간에 수학적인 함수 $f$가 존재한다고 가정을 했는데, 문제는 밉매핑을 구현하기 위하여 그러한 함수에 대한 미분 값들을 효과적으로 계산해주어야 한다는 점이다. 어떻게 보면 이는 매우 어렵거나 불가능한 것처럼 보일지도 모르나, 실제로는 비교적 쉽게 미분 값 계산을 수행할 수가 있다. 여기서는 4장의 3.6절에서 설명한 원근 교정 시의 래스터화 과정의 문맥에서의 $d$ 값의 계산 방법에 대하여 이해하여 보자. 재차 강조하지만 원근 교정에 있어 중요한 사실은 원근 투영을 할 때 $\frac{1}{w_c}$과 보간을 하려하는 데이터 $f$에 대한 $\frac{f}{w_c}$가 윈도우 좌표계에서 선형적으로 변하기 때문에, 이 값들에 대하여 선형 보간을 한 후 $\frac{f}{w_c}$에 대한 보간 값을 $\frac{1}{w_c}$에 대한 보간 값으로 나누어 $f$에 대한 정확한 보간 값을 구한다는 점이다. 원근 교정의 효과가 가장 두드러지는 것은 텍스춰 좌표에 대한 교정인데, 텍스춰 좌표를 보간을 할 때에는 $f$가 $s$와 $t$인 경우이고, $u$, $v$와 $s$, $t$간에 선형적인 관계가 존재하므로 $\frac{u}{w_c}$와 $\frac{v}{w_c}$도 윈도우 공간에서 선형적으로 변하는 값이 된다.

원근 교정에 대한 설명을 할 때, 윈도우 공간의 변수 $x$와 $y$에 대한 $\frac{1}{w_c}$과 $\frac{f}{w_c}$의 미분 방법에 대하여 기술을 하였는데, 같은 방식으로 $\frac{u}{w_c}$와 $\frac{v}{w_c}$에 대한 편미분 값을 구할 수가 있다. 다시 한번 원근 교정을 할 때의 래스터화 계산 시의 상황에 대하여 살펴보자. 현재 한 삼각형을 래스터화하려 하는데, 각 꼭지점에는 꼭지점 좌표 $(x, y)$가 주어질 뿐만 아니라 보간을 하려고 하는 텍스춰 좌표 $(u, v)$가 붙어 있으며, 윈도우 좌표 공간에서 $\frac{1}{w_c}$, $\frac{u}{w_c}$, 그리고 $\frac{v}{w_c}$ 값이 선형적으로 변하고 있다. 앞에서 설명한 방식으로 이 세 개의 값들의 $x$와 $y$에 대한 편미분 값들을 계산할 수 있으므로, 한 꼭지점에 대한 값에 적절한 증분을 더하면 윈도우 좌표계의 원점에 대응이 되는 값 $(\frac{1}{w_c})_0$, $(\frac{u}{w_c})_0$, 그리고 $(\frac{v}{w_c})_0$을 구할 수가 있다.

이 때 임의의 $(x, y)$에 대한 $u$와 $v$는 다음과 같이 보간을 할 수 있고,

$$\begin{aligned} u &= u(x,y) = \frac{\frac{\partial \frac{u}{w_c}}{\partial x}\cdot x + \frac{\partial \frac{u}{w_c}}{\partial y}\cdot y + (\frac{u}{w_c})_0}{\frac{\partial \frac{1}{w_c}}{\partial x}\cdot x + \frac{\partial \frac{1}{w_c}}{\partial y}\cdot y + (\frac{1}{w_c})_0}, \\ v &= v(x,y) = \frac{\frac{\partial \frac{v}{w_c}}{\partial x}\cdot x + \frac{\partial \frac{v}{w_c}}{\partial y}\cdot y + (\frac{v}{w_c})_0}{\frac{\partial \frac{1}{w_c}}{\partial x}\cdot x + \frac{\partial \frac{1}{w_c}}{\partial y}\cdot y + (\frac{1}{w_c})_0}, \end{aligned}$$

분모에 있는 값을 $z(x, y)$라 하면 $u$와 $v$에 대한 편미분 값은 다음과 같이 구할 수가 있다.

$$\begin{aligned} u_x &= \frac{\partial u}{\partial x} = \frac{z(x,y)\cdot \frac{\partial \frac{u}{w_c}}{\partial x} - \frac{u}{w_c}\cdot \frac{\partial \frac{1}{w_c}}{\partial x}}{z^2(x,y)} = \frac{\gamma + \alpha y}{z^2(x,y)} \\ v_x &= \frac{\partial v}{\partial x} = \frac{z(x,y)\cdot \frac{\partial \frac{v}{w_c}}{\partial x} - \frac{v}{w_c}\cdot \frac{\partial \frac{1}{w_c}}{\partial x}}{z^2(x,y)} = \frac{\delta + \beta y}{z^2(x,y)} \\ u_y &= \frac{\partial u}{\partial y} = \frac{z(x,y)\cdot \frac{\partial \frac{u}{w_c}}{\partial y} - \frac{u}{w_c}\cdot \frac{\partial \frac{1}{w_c}}{\partial y}}{z^2(x,y)} = \frac{\epsilon - \alpha x}{z^2(x,y)} \\ v_y &= \frac{\partial v}{\partial y} = \frac{z(x,y)\cdot \frac{\partial \frac{v}{w_c}}{\partial y} - \frac{v}{w_c}\cdot \frac{\partial \frac{1}{w_c}}{\partial y}}{z^2(x,y)} = \frac{\varepsilon - \beta x}{z^2(x,y)} \end{aligned}$$

여기서 $\alpha$, $\beta$, $\cdots$, $\varepsilon$은 다음과 같이 정의되는 상수이다.

$$\begin{aligned}
\alpha &= \frac{\partial \frac{u}{w_c}}{\partial x} \cdot \frac{\partial \frac{1}{w_c}}{\partial y} - \frac{\partial \frac{1}{w_c}}{\partial x} \cdot \frac{\partial \frac{u}{w_c}}{\partial y}, \\
\beta &= \frac{\partial \frac{v}{w_c}}{\partial x} \cdot \frac{\partial \frac{1}{w_c}}{\partial y} - \frac{\partial \frac{1}{w_c}}{\partial x} \cdot \frac{\partial \frac{v}{w_c}}{\partial y}, \\
\gamma &= \frac{\partial \frac{u}{w_c}}{\partial x} \cdot (\frac{1}{w_c})_0 - \frac{\partial \frac{1}{w_c}}{\partial x} \cdot (\frac{u}{w_c})_0, \\
\delta &= \frac{\partial \frac{v}{w_c}}{\partial x} \cdot (\frac{1}{w_c})_0 - \frac{\partial \frac{1}{w_c}}{\partial x} \cdot (\frac{v}{w_c})_0, \\
\epsilon &= \frac{\partial \frac{u}{w_c}}{\partial y} \cdot (\frac{1}{w_c})_0 - \frac{\partial \frac{1}{w_c}}{\partial y} \cdot (\frac{v}{w_c})_0, \\
\varepsilon &= \frac{\partial \frac{v}{w_c}}{\partial y} \cdot (\frac{1}{w_c})_0 - \frac{\partial \frac{1}{w_c}}{\partial y} \cdot (\frac{v}{w_c})_0
\end{aligned}$$

이렇게 $u_x$, $v_x$, $u_y$, 그리고 $v_y$를 구한 후, $x_l = \sqrt{(\gamma + \alpha y)^2 + (\delta + \beta y)^2}$와 $y_l = \sqrt{(\epsilon - \alpha x)^2 + (\varepsilon - \beta x)^2}$라 하면 식 (5.1)의 $d$ 값은 다음과 같이 표현할 수가 있다.

$$d(x, y) = \frac{1}{z^2(x, y)} \cdot max(x_l, y_l) \tag{5.2}$$

언뜻 보면 삼각형이 투영되는 모든 픽셀$(x, y)$에 대하여 이러한 계산을 반복하는 것이 상당한 계산량을 요구하는 것처럼 보인다. 하지만 다음과 같은 점을 고려하면 생각만큼 많은 계산을 할 필요는 없다.

1. $\alpha$, $\beta$, $\cdots$, $\varepsilon$은 주어진 삼각형에 대하여 상수이므로, 각 픽셀마다 이 값들을 구할 필요가 없이 래스터화를 하려는 각 삼각형에 대하여 한 번만 계산한다.

2. $x_l$의 $\gamma + \alpha y$와 $\delta + \beta y$는 오로지 $y$에 대한 함수이고, $y_l$의 $\epsilon - \alpha x$와 $\varepsilon - \beta x$는 오로지 $x$에 대한 함수이다. 따라서 래스터화 과정에서 매 픽셀마다 반복해서 $x_l$과 $y_l$을 계산할 필요가 없이, 삼각형이 투영되는 각 스캔 라인에 대하여 $x_l$을 한

번씩만 계산을 하고, 마찬가지로 $y_l$은 삼각형이 차지하는 $x$축 방향의 최소, 최대 범위 안의 $x$ 값에 대해서만 한 번씩 계산을 하면 된다.

따라서 주어진 삼각형에 대하여 필요한 $x_l$과 $y_l$ 값을 미리 계산을 해놓으면, 각 픽셀 $(x, y)$에 대해서 $d(x, y)$를 계산할 때 해당 $x_l$과 $y_l$중 큰 값을 선택하여 $z^2(x, y)$로 나누어 주면 된다. 여러 번 언급한 바와 같이 나눗셈은 상대적으로 비용이 많이 드는 연산이다. 사실 여기서 $z(x, y)$는 픽셀 $(x, y)$에서의 $\frac{1}{w_c}$ 값에 해당한다. 4장의 3.6절에서 설명한 바와 같이 $\frac{1}{z(x,y)} = \frac{1}{\frac{1}{w_c}}$은 래스터화 과정에서 계산이 된다. 따라서 $d(x, y)$ 값을 계산하기 위하여 추가적으로 이 값을 계산할 필요가 없이 이 값의 제곱 값을 구해 $x_l$과 $y_l$ 중 큰 값에 곱해주기만 한다. 따라서 삼각형이 투영되는 각 픽셀에 대하여 $d(x, y)$를 구하기 위해서 비교 연산 한 번과 곱셈 두 번을 추가적으로 수행하면 된다. 물론 이것이 한 픽셀 당 필요한 계산량이 아니고, 앞의 1과 2번의 계산을 하는데 필요한 비용을 삼각형이 투영되는 모든 픽셀의 개수로 나누어 준 값을 이에 추가해준 것이 픽셀 당 $d(x, y)$를 계산하는데 드는 비용이 된다.

참고로 원근 교정은 무엇보다도 정확한 텍스춰 매핑에 필수적인 연산이라 하였는데, 여기서의 $d$ 값의 계산 과정 또한 양질의 텍스춰 매핑 결과를 산출하는데 있어 매우 중요한 요소라 할 수 있다. 3차원 게임이나 가상 현실과 같은 실시간 렌더링 소프트웨어에서의 텍스춰 매핑의 중요성을 생각해보면, 이 두 가지 계산 과정은 실시간 렌더링 시스템의 개발에 있어 핵심이 되는 요소임에 틀림 없다.

# 제 6 절 텍스춰 좌표의 자동 생성과 응용

## 6.1 OpenGL에서의 자동 좌표 생성

앞에서 살펴본 텍스춰 매핑의 예에서는 기하 물체에 대한 텍스춰 좌표를 미리 계산을 해놓고, 꼭지점 좌표를 기술할 때 glTexCoord2*(*) 함수를 사용하여 대응이 되는 텍스춰 좌표를 명시적으로 붙여주는 방식을 택하였다. 항상 그런 것은 아니지만 이러한 방식은 보편적으로 텍스춰가 기하 물체에 고정이 되어 있는 '정적인' 상황에 쓰인다. 반면에 이와는 달리 기하 물체의 각 꼭지점에 대한 텍스춰 좌표를 프로그램 실행 중에 '동적으로' 계산을 해야하는 경우가 종종 발생한다. 보통 실행 시에 기하 물체의 꼭지점 좌표나 법선 벡터의 값에 따라 텍스춰 좌표를 계산하는데, 계산 방법은 텍스춰 매핑 기법의 사용 목적에 따라 그에 맞는 방식을 취한다. OpenGL 시스템은 약간은 제한적이기는 하지만 기하 물체의 속성에 따라 텍스춰 좌표를 동적으로 자동 생성해주는 기능을 제공한다. 이 절에서는 OpenGL의 텍스춰 좌표의 자동 생성(automatic texture-coordinate generation) 기능에 대하여 간단하게 살펴보고, 실제 응용 예를 통하여 연습을 해보도록 하겠다.

OpenGL에서의 자동 생성은 크게 두 가지 부류로 나누어 생각을 할 수가 있다. 하나는 기하 물체의 꼭지점 좌표를 사용하여 텍스춰 좌표를 생성하는 것인데, 이 방법은 다시 모델링 좌표계나 세상 좌표계에 해당하는 물체 좌표계에서의 좌표 값 $(x_o\ y_o\ z_o\ 1)^t$을 사용하는지, 아니면 눈 좌표계에서의 좌표 값 $(x_e\ y_e\ z_e\ 1)^t$을 사용하는지에 따라 두 갈래로 나누어 진다. 다음 두 번째 부류는 꼭지점에 붙은 법선 벡터의 방향을 사용하여 텍스춰 좌표를 계산하는 것인데, 이 경우에는 눈 좌표계 공간에서의 법선 벡터 값을 사용한다.

좌표 생성을 위하여 어떤 기하 속성을 사용할지는 void glTexGenf(GLenum co-

ord, GLenum pname, GLfloat param); 함수와 void glTexGenfv(GLenum coord, GLenum pname, GLfloat *param); 함수를 사용하여 지정할 수가 있다[12]. OpenGL에서는 $(s, t, r, q)$의 각 좌표에 대한 계산 방식을 다르게 설정할 수 있는데, 첫 번째 인자 coord에는 GL_S, GL_T, GL_R, 그리고 GL_Q 중 생성 방법을 설정하려는 텍스춰 좌표에 대한 상수를 사용하면 된다. 다음 두 번째 인자 pname이 GL_TEXTURE_GEN_MODE이면 세 번째 인자 param으로 GL_OBJECT_LINEAR, GL_EYE_LINEAR, GL_SPHERE_MAP 중 하나를 선택하는데, 이는 텍스춰 좌표를 계산하기 위하여 각각 물체 좌표계에서의 꼭지점의 좌표, 눈 좌표계에서의 꼭지점의 좌표, 그리고 눈 좌표계에서의 법선 벡터를 사용하라는 것을 의미한다.

텍스춰 좌표를 동적으로 계산하는 방법에는 여러 가지가 있지만, OpenGL에서는 제한된 형태의 방식만 제공한다. 실제로 위의 세 가지 OpenGL 상수는 어떤 기하 속성을 사용하는가뿐만 아니라, 어떠한 방식을 사용할 지도 명시하고 있다. GL_OBJECT_LINEAR와 GL_EYE_LINEAR의 경우 그 이름이 암시하듯이 프로그래머가 해당 공간에서의 평면을 기술하면, 그 평면과 꼭지점과의 선형적인 관계에 의하여 텍스춰 좌표를 계산한다. 앞에서 사용자 설정 절단에 대하여 살펴볼 때 언급한 바와 같이 OpenGL에서는 하나의 평면을 네 개의 상수 $a$, $b$, $c$, 그리고 $d$를 사용하여 $a \cdot x + b \cdot y + c \cdot z + d = 0$과 같이 표현한다. 여기서 텍스춰 계산을 위한 평면을 기술하기 위해 각 텍스춰 좌표에 대하여 glTexGenfv(*) 함수를 사용하면 되는데, 두 번째 인자로 사용할 수 있는 GL_OBJECT_PLANE과 GL_EYE_PLANE은 각각 해당 공간에서의 평면을 의미한다. 이 때 포인터 값인 세 번째 인자는 평면의 네 개의 계수를 가리키도록 설정하면 된다.

GL_OBJECT_LINEAR 방식을 사용할 때, $a$, $b$, $c$, $d$를 GL_OBJECT_PLANE을

[12] f 버전 외에도 i 버전과 d 버전도 있음.

사용하여 기술한 평면의 계수라 하면, 해당 텍스춰 좌표 $g$는 $w_o$ 좌표까지 포함한 물체 좌표계에서의 좌표 $(x_o\ y_o\ z_o\ w_o)^t$를 사용하여 다음과 같이 계산을 한다.

$$g = a \cdot x_o + b \cdot y_o + c \cdot z_o + d \cdot w_o$$

위 식에서 $(a\ b\ c)^t$가 단위 벡터로 정규화가 되어 있다면 $g$는 바로 물체 공간에서의 평면 $a \cdot x + b \cdot y + c \cdot z + d = 0$과 꼭지점 $(x_o\ y_o\ z_o\ w_o)^t$ 간의 거리를 나타냄을 알 수가 있다.

반면 GL_EYE_LINEAR 방식을 사용할 경우에는, 약간 다른 방식을 사용하여 텍스춰 좌표를 계산한다. GL_EYE_PLANE을 사용하여 평면의 계수 $(a\ b\ c\ d)^t$를 설정하면, 바로 그 당시의 현재 모델뷰 행렬 $M_{MV}$를 사용하여, 이 평면을 눈 좌표계로 변환을 한다. 즉 2장의 3.17절에서 설명한데로 $(a'\ b'\ c'\ d')^t = (a\ b\ c\ d)^t \cdot M_{MV}^{-1}$와 같이 평면을 변환하는데, 이는 지금까지 살펴본 OpenGL에서의 기하 파이프라인의 원리를 이해하였다면 매우 자연스러운 것이라 하겠다. 일단 이렇게 평면을 변환을 한 후 눈 좌표계 공간에서의 꼭지점 $(x_e\ y_e\ z_e\ w_e)^t$와 평면간의 거리를 계산하여 텍스춰 좌표로 사용한다.

$$g = a' \cdot x_e + b' \cdot y_e + c' \cdot z_e + d' \cdot w_e$$

예를 들어 다음과 같은 코드를 살펴보자.

```
float s_plane[] = {1.0, 0.0, 0.0, 0.0 };
float t_plane[] = {0.0, 1.0, 0.0, 0.0 };
     ⋮
glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGenfv(GL_S, GL_EYE_PLANE, s_plane);
glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
```

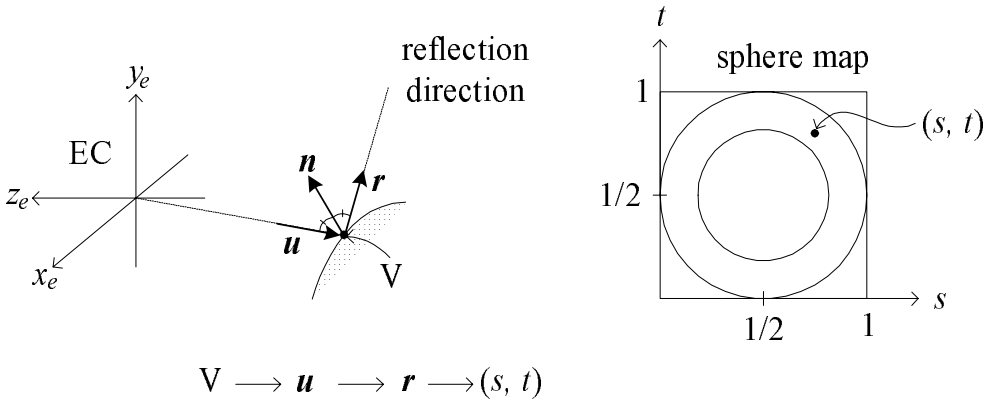



그림 5.21: 구 매핑 시의 텍스춰 좌표의 계산

```
glTexGenfv(GL_T, GL_EYE_PLANE, t_plane);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

이 예에서는 눈 좌표계 공간에서의 꼭지점 좌표를 사용하여 텍스춰 좌표를 계산하도록 프로그래밍을 하였는데, glTexGenfv(*) 함수를 호출할 때의 현재 모델뷰 행렬이 단위 행렬이라면 결과적으로 $s$와 $t$ 좌표로 각각 눈 좌표계에서의 꼭지점의 좌표 $x_e$와 $y_e$를 사용한다. 따라서 위에서와 같은 설정으로 텍스춰 매핑을 하면 물체에 입혀지는 텍스춰는 상대적으로 눈 좌표계에 대하여 고정이 된다. 그 결과 물체가 움직이면 텍스춰가 입혀지는 모양이 변하게 되지만, 화면을 기준으로 해서 보면 변하지가 않는다. 만약 GL_OBJECT_LINEAR 방식을 사용한다면 텍스춰가 물체에 고정이 되어 입혀지는데, 사용하려는 응용의 목적에 따라 적절한 방법을 선택하면 된다. OpenGL에서는 디폴트로 glTexCoord2*(*) 함수를 사용하여 설정하는 현재 텍스춰가 꼭지점에 붙게 되므로, 자동 생성을 원하는 경우에는 원하는 텍스춰 좌표마다 마지막 두 문장에서처럼 명시적으로 자동 생성 계산을 요청해야 한다.

꼭지점 좌표를 사용하는 이 두 가지 방법과는 달리 꼭지점의 법선 벡터를 사용하는 GL_SPHERE_MAP 방식을 선택하면 약간 복잡한 경로를 통하여 텍스춰 좌표를 생성한다. 그림 5.21에는 기하 물체의 한 꼭지점 V에 대하여 텍스춰 좌표 $(s, t)$가 자

동 생성되는 과정이 도시되어 있다. 여기서 계산은 눈 좌표계를 기준으로 하여 수행이 되는데, 우선 시점에 해당하는 원점에서 V를 향한 단위 벡터 **u**를 계산한 후, 이를 V에서의 단위 법선 벡터 **n**의 반대 방향으로 반사를 시킨 벡터 $\mathbf{r} \equiv (r_x\ r_y\ r_z)^t$를 계산한다. 이 벡터는 다음과 같이 구할 수 있음을 쉽게 유도할 수가 있는데,

$$\mathbf{r} = \mathbf{u} - 2(\mathbf{n} \cdot \mathbf{u})\mathbf{n}$$

**u** 벡터는 V를 바라보는 시선의 방향에 해당하므로, **r** 벡터는 V에서의 반사 방향(reflection direction)에 해당한다. 정확히 말하면 이 벡터는 물체가 거울과 같이 완전하게 정반사를 한다고 할 때 V를 통하여 보이는 빛이 들어오는 방향을 가리킨다. GL_SPHERE_MAP 방식을 선택하면 이렇게 구한 **r** 벡터가 2차원 텍스춰 이미지를 액세스하는데 사용이 되기 때문에, 이 방법은 주변의 모습이 물체에 반사가 되는 환경 매핑이나 하이라이트처럼 정반사와 관련된 렌더링 효과를 구현하는데 사용이 된다.

문제는 **r** 벡터를 이차원 평면 공간인 텍스춰 공간의 $[0,1] \times [0,1]$ 영역에 어떻게 매핑을 할 것인가이다. OpenGL에서는 $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$이라 할 때, 다음과 같이 꼭지점 V에 붙여줄 텍스춰 좌표를 계산한다.

$$s = \frac{r_x}{m} + \frac{1}{2}, \qquad t = \frac{r_y}{m} + \frac{1}{2}$$

이러한 방식은 3차원 공간의 임의의 방향, 다르게 표현하면 중심이 원점에 있고 반지름이 1인 구 상의 임의의 점과, 2차원 평면 상의 정사각형 안의 점과의 매핑 방법을 제공한다. 전체적인 매핑 구조를 이해하기 위하여 몇 가지 매핑 예를 들어보면, 만약 단위 벡터인 **r**이 $(0\ 0\ 1)^t$이라면, 즉 눈 좌표계에서 양의 $z_e$축 방향을 가리

키면, 이 벡터는 텍스춰 이미지의 중심 $(\frac{1}{2}, \frac{1}{2})$으로 매핑이 된다. 또한 $\mathbf{r} = (r_x\ r_y\ 0)^t$ $(\sqrt{r_x^2 + r_y^2} = 1)$과 같이 $z_e$축에 수직인 벡터들은 그림 5.21에서의 텍스춰 이미지의 안쪽에 있는 반지름이 $\frac{1}{2\sqrt{2}}$인 원 위로 매핑이 된다. 구의 중심을 카메라가 세상을 바라보는 방향인 음의 $z_e$축 상에 올려 놓고 바라볼 때, 보이는 쪽의 반구에 해당하는 방향들이 안쪽의 원에 매핑이 되고, 뒤쪽의 반구에 해당하는 방향은 큰 원과 작은 원과의 사이에 연속적으로 매핑이 된다. 이 방법은 비교적 쉽게 구현할 수 있는 장점이 있는 반면, 매핑 과정에서 생기는 두 공간간의 왜곡은 피할 수가 없다.

이와 같이 GL_SPHERE_MAP 방식을 사용하면 구에 기반하여 텍스춰 좌표를 생성하는데, 이러한 방식의 텍스춰 매핑을 구 매핑(sphere mapping)이라 하고, 이 때의 텍스춰 이미지를 구맵(sphere map)이라 부른다. 구 매핑은 여러 가지 상황에서 유용하게 쓰일 수가 있다. 한 가지 응용은 구를 렌더링을 하려고 하는 물체의 중심에 놓고, 각 방향에 대하여 광선 추적법에서와 같이 광선을 쏘아 그 방향으로 들어오는 빛의 색깔을 구하여 구맵을 구성하여 놓은 후, 구 매핑 기법을 적용하면 마치 물체에 주변 상황이 반사되어 보이는 것과 같은 모습을 실시간적으로 구현할 수가 있다. 이러한 방법을 환경 매핑이라 한다고 했는데, 지역 조명 모델을 사용하는 OpenGL과 같은 렌더링 시스템에서 반사라는 전역 조명의 특징를 근사적으로 묘사하는 기법 중의 하나이다.

지금까지 OpenGL에서의 텍스춰 좌표의 자동 생성 방법에 대하여 간략하게 살펴보았는데, 이러한 기능은 주로 고급 렌더링 기법에서 쓰이고 있다. 다음의 두 절에서는 자동 생성 기법의 응용 예를 두 가지 알아보도록 하겠다.

## 6.2 투영 텍스춰

투영 텍스춰(projective texture)는 전형적인 2차원 텍스춰 매핑의 기능을 확장시켜,

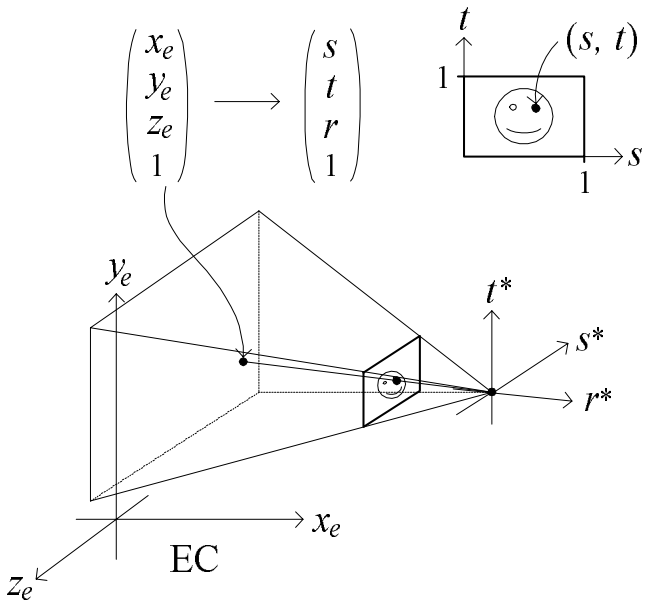



그림 5.22: 투영 텍스춰의 원리

OpenGL과 같은 실시간 렌더링 시스템에서 슬라이드 프로젝터, 스폿 광원, 그림자 등의 고급 기법을 자연스럽게 구현할 수 있도록 해주는 기법으로서 시갈 등에 의해 제안이 되었다[13]. 이 기법의 기본 원리는 마치 슬라이드 프로젝터처럼 2차원 텍스춰 이미지를 3차원 공간으로 투사를 하여, 기하 물체를 구성하는 각 꼭지점에서의 텍스춰 좌표를 동적으로 구하는 것으로서, 이를 어떻게 응용하는가에 따라 다양한 렌더링 효과를 산출할 수가 있다.

투영 텍스춰 기법을 이해하기 위하여 그림 5.22를 살펴보자. 편의상 눈 좌표계를 기준으로 하여 설명을 하려하는데, 여기에는 두 개의 좌표축이 있음을 알 수 있다. 하나는 눈 좌표계에 대한 것이고, 다른 하나는 2차원 텍스춰를 3차원 공간으로 뿌려주기 위한 것이다. 지금 $s^*$, $t^*$, $r^*$축 공간에서 음의 $r^*$축 방향으로 2차원 텍스춰 이미지를 좌표축에 수직으로 고정시키고, 원점에 전구를 키면, 이 이미지가 3차원

[13] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. "Fast Shadow and Lighting Effects Using Texture Mapping," *ACM SIGGRAPH* '92, pp. 1-11, 1992.

공간으로 투사가 된다. 마치 우리가 환등기라 부르는 슬라이드 프로젝터를 연상시키는데, 바로 이것이 투영 텍스춰의 기본 원리이다. 이럴 경우 텍스춰 이미지가 3차원 공간인 눈 좌표계에 존재하는 기하 물체에 자연스럽게 투영이 되는데, 문제는 임의의 꼭지점 좌표 $(x_e\ y_e\ z_e\ 1)^t$에 투사가 되는 텍스춰 이미지의 좌표 $(s,t)$를 어떻게 구하는가 하는 것이다.

OpenGL을 사용한 투영 텍스춰 기법에서는 구현의 편의상 텍스춰가 투사가 되는 영역의 깊이를 제한한다. 따라서 이 그림에서와 같이 마치 눈 좌표계 공간에서의 원근 투영시의 뷰잉 볼륨과 같은 영역 안에서만 투사가 된다고 가정을 한다. 즉 광원은 원점에, 그리고 슬라이드 필름에 해당하는 텍스춰 이미지는 앞 절단 평면 상의 사각형에 놓이는데, 슬라이드 프로젝터에 대한 이러한 기하 속성은 OpenGL 기하 파이프라인의 뷰잉 변환과 원근 투영 변환에 자연스럽게 대응이 된다. 따라서 눈 좌표계 공간에 슬라이드 프로젝터를 위치하는 작업은 기하 변환을 위한 OpenGL 함수들을 사용하여 쉽게 구현을 할 수 있다.

슬라이드 프로젝터에 대한 투사 영역을 결정하는 것은 슬라이드 프로젝터 입장에서는 이 그림에서의 눈 좌표계를 세상 좌표계로 생각을 하고, 그러한 세상 좌표계에서 카메라의 위치와 방향을 설정(뷰잉 변환)하고 뷰잉 볼륨을 결정(투영 변환)하는 작업과 동일하다. 한 가지 차이점은 슬라이드 프로젝터의 관점에서 볼 때 물체 좌표계의 점에 해당하는 꼭지점의 좌표 $(x_e\ y_e\ z_e\ 1)^t$가 뷰잉 변환과 투영 변환을 거친 후, 뷰폿 변환을 거쳐 윈도우 좌표계로 변환이 되는 것이 아니라, 꼭지점이 앞 절단 평면 상의 텍스춰로 투영이 되는 점에 해당하는 텍스춰 좌표로 변환이 되어야 한다는 점이다. 텍스춰 좌표 $(s,t)$는 $[0,1]\times[0,1]$의 범위 안에 들어오기 때문에 윈도우 좌표계로 변환을 시키는 뷰폿 변환하고 약간은 다르나, 변환의 형태, 즉 이동 변환과 크기 변환을 사용하는 기본 원리는 같다.

여기서는 슬라이드 프로젝터 및 스폿 광원 효과와 같이 $s$와 $t$ 좌표만 사용하는 예만 살펴보겠지만, 실제로 그림자 생성과 같은 응용에서는 $r$ 좌표의 값이 중요하게 쓰인다. 즉 원래의 기하 파이프라인에서 $z_{wd}$가 깊이 정보를 나타내듯이, 이 경우에서 $r$ 좌표는 텍스춰가 투사가 되는 점이 앞 절단 평면에서 얼마나 떨어져 있는가에 대한 정보를 제공한다. 좀 더 정확하게 말하면, 투영 텍스춰를 구현하기 위해서 $s$, $t$, 그리고 $r$ 좌표 외에 $q$ 좌표를 사용한다. 앞에서도 언급한 바와 같이 OpenGL에서는 텍스춰 좌표가 $(s\ t\ r\ q)^t$로 표현이 된다. 여기서 $r$는 3차원 텍스춰 매핑에서 세 번째 좌표를 나타내기 위한 것이고, $q$ 좌표는 꼭지점의 좌표 $(x\ y\ z\ w)^t$에서 $w$에 해당하는데, 투영 텍스춰 기법에서는 텍스춰 좌표를 구하기 위하여 원근 투영이 사용이 되므로 $q$의 사용은 필수적이라고 할 수가 있다.

그러면 투영 텍스춰의 구현에 있어 공통적으로 프로그래밍이 되어야 하는 부분에 대하여 살펴보자. 우선 처음에 슬라이드 프로젝터에 대한 기하 변환의 출발점은 눈 좌표계 공간에서의 기하 물체의 꼭지점의 좌표 $(x_e\ y_e\ z_e\ 1)^t$이므로, 다음과 같이 투영을 할 텍스춰 이미지에 대하여 이 좌표 값을 사용하여 텍스춰의 좌표가 자동 생성이 되도록 설정한다.

```
GLfloat s_plane[] = { 1.0, 0.0, 0.0, 0.0 };
GLfloat t_plane[] = { 0.0, 1.0, 0.0, 0.0 };
GLfloat r_plane[] = { 0.0, 0.0, 1.0, 0.0 };
GLfloat q_plane[] = { 0.0, 0.0, 0.0, 1.0 };
     ⋮
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGenfv(GL_S, GL_EYE_PLANE, s_plane);

glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGenfv(GL_T, GL_EYE_PLANE, t_plane);

glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
```

```
glTexGenfv(GL_R, GL_EYE_PLANE, r_plane);

glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGenfv(GL_Q, GL_EYE_PLANE, q_plane);
```

여기서 주의할 점은 GL_EYE_LINEAR 모드에서는 평면의 계수를 기술하면, 해당 평면이 현재 모델뷰 행렬에 의하여 변환이 된 후에 사용이 되므로, 위의 glTexGenfv(*) 함수들을 호출할 때 현재 모델뷰 행렬이 단위 행렬이 되도록 해주어야 한다는 사실이다. 그럴 경우 꼭지점의 좌표 $(x_e\ y_e\ z_e\ 1)^t$가 텍스춰 좌표 $(s\ t\ r\ q)^t$로 설정이 되는데, 이 텍스춰 좌표는 아직 실제로 텍스춰 이미지를 액세스할 때 사용하는 좌표가 아니고, 이 좌표 값에 대하여 앞에서 설명한 슬라이드 프로젝터의 설정을 위한 기하 변환을 통하여 적절하게 변환을 해주어야 한다.

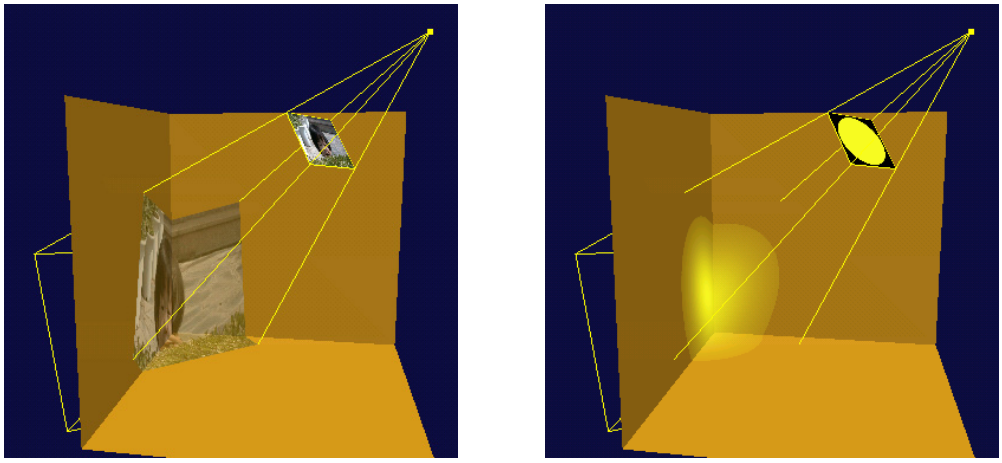
OpenGL에서는 텍스춰의 좌표를 glTexCoord*(*) 함수를 사용하여 기술을 하건, 아니면 지금과 같이 자동 생성을 하건, 일단 텍스춰 좌표를 텍스춰 행렬의 탑에 있는 현재 텍스춰 행렬에 곱해 변환을 한 후 꼭지점에 붙여준다. 바로 이러한 변환 기능이 OpenGL을 사용하여 투영 텍스춰를 구현하는데 있어 핵심적인 역할을 한다. 전체적인 구현 방식을 기술하면, 1. 텍스춰 자동 생성 기능을 통하여 일단 눈 좌표계 공간의 꼭지점의 좌표를 텍스춰 좌표로 만들어 준 후, 2. 슬라이드 프로젝터에 대한 기하 변환을 텍스춰 행렬 스택에 올려주면, 3. 기하 물체를 그릴 때 OpenGL 시스템에서 자동적으로 원하는 텍스춰 좌표를 계산해주게 된다. 슬라이드 프로젝터를 구현하기 위한 기하 변환은 다음과 같은 함수에 의하여 설정할 수가 있다.

```
void set_slide_projector(void) {
 glMatrixMode(GL_TEXTURE);
 glLoadIdentity();
 glTranslatef(0.5, 0.5, 0.5);
 glScalef(0.5, 0.5, 0.5);
```

```
  gluPerspective(25.0, 1.0, 7.0, 30.0);
  gluLookAt(8.0, 8.0, -22.0, 2.0, 2.0, -30.0, 0.0, 1.0,
0.0);
  glMatrixMode(GL_MODELVIEW);
}
```

여기서는 우선 `glMatrixMode(GL_TEXTURE);` 문장을 통하여 텍스춰 행렬 스택을 선택한다. 다음 기하 변환에 관한 OpenGL 함수를 적절히 호출하고 있는데, 이 함수의 수행 결과 스택에 올려지는 변환 행렬에 대하여 살펴보면, 우선 `gluLookAt(*)` 함수에 의한 뷰잉 변환 $M_{V_t}$가 적용이 되는데, 이 변환을 통하여 원래의 OpenGL 뷰잉에서 물체 좌표계를 기준으로 카메라의 위치와 방향을 설정하듯이, 눈 좌표계 공간을 기준으로 하여 슬라이드 프로젝터의 위치와 투영 방향을 결정할 수 있다. 다음 `gluPerspective(*)` 함수에 의한 원근 투영 변환 $M_{P_t}$는 피사계 범위, 앞 절단 평면, 그리고 뒤 절단 평면을 통하여 투사의 범위를 결정을 하는데 사용이 된다. 투영 텍스춰를 구현할 때, 처음 출발하는 세상 좌표계, 즉 OpenGL의 용어를 사용하면 물체 좌표계의 좌표 $(s_o\ t_o\ r_o\ 1)^t$는 $(x_e\ y_e\ z_e\ 1)^t$가 되고, 이 좌표가 $M_{V_t}$에 의하여 텍스춰의 눈 좌표계의 점 $(s_e\ t_e\ r_e\ 1)^t$로 변환이 된 후, $M_{P_t}$에 의해 텍스춰의 절단 좌표계의 점 $(s_c\ t_c\ r_c\ q_c)^t$를 거쳐 정규 디바이스 좌표계의 점 $(s_{nd}\ t_{nd}\ r_{nd}\ 1)^t$로 변환이 된다. 물론 여기서도 $(s_{nd}\ t_{nd}\ r_{nd}\ 1)^t = (\frac{s_c}{q_c}\ \frac{t_c}{q_c}\ \frac{r_c}{q_c}\ 1)^t$와 같이 $q_c$에 의한 원근 나눗셈이 수행이 된다.

정규 디바이스 좌표계 공간에서의 좌표는 -1과 1 사이의 값을 가지므로, 이 값을 우리가 원하는 텍스춰 좌표로 변환을 해주어야 하는데, 특히 $s$와 $t$값을 0과 1 사이의 값으로 만들어 주어야 한다. 이를 위하여 위의 코드에서 이동 변환과 크기 변환에 대한 OpenGL 함수를 호출하고 있는데, 그 결과 행렬 $T(0.5, 0.5, 0.5) \cdot S(0.5, 0.5, 0.5)$가 스택에 올라가고, 이 변환에 의하여 실제로 기하 물체의 꼭지점이 앞 절단 평면에 놓인 텍스춰 이미지에 투영되는 점의 텍스춰 좌표를 구할 수가 있게 된다. 이 예에



(a) 슬라이드 프로젝터 효과 (b) 부드러운 스폿 광원 효과

그림 5.23: 투영 텍스춰 기법의 사용 예

서 중요한 것은 2차원 텍스춰 이미지의 좌표 $(s,t)$이므로, 단순한 슬라이드 프로젝터 효과를 위해서는 $r$ 값은 무시해도 좋으나, 재차 강조하면, 투영 텍스춰 기법을 사용하여 그림자를 생성할 때는 0과 1 사이로 변환된 이 값이 슬라이드 프로젝터, 다시 말해서 광원을 기준으로 한 깊이 정보를 제공하기 때문에 중요한 역할을 한다.

요약을 하면, 눈 좌표계 공간의 꼭지점 좌표가 다음과 같이 텍스춰 좌표로 변환이 되는데,

$$\begin{pmatrix} s \\ t \\ r \\ 1 \end{pmatrix} = T(0.5,0.5,0.5)\cdot S(0.5,0.5,0.5)\cdot M_{P_t}\cdot M_{V_t}\cdot \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix}$$

여기서 $(s,t)$는 좌표가 $(x_e\ y_e\ z_e\ 1)^t$인 기하 물체의 지점으로 투영이 되는 텍스춰의 좌표가 된다.

그림 5.23은 투영 텍스춰 기법을 구현한 예제 프로그램 5.C를 수행한 모습을 보여준다. 이 프로그램에서 왼쪽 마우스 버튼을 누른 후 벽면을 움직여보면 텍스춰가 어떻게 투사가 되는지를 알 수가 있을 것이다. 참고로 't' 키는 텍스춰를 선택하기 위한 키이고, 'f' 키는 텍스춰의 뷰잉 볼륨에 대한 정보에 관한 키이다. 여기서는 GL_RGBA 타입의 텍스춰 이미지를 사용하여 GL_DECAL 모드로 텍스춰를 입히고 있다. 앞의 세 개의 텍스춰는 이미지 전체에 대하여 일정한 알파 값을 가지는 반면, 마지막 텍스춰는 스폿 광원처럼 중심으로부터 멀어질수록 알파 값이 감소하도록 설정을 하였다. 그 결과 마지막 텍스춰에 대해서는 중심 부분이 밝게 보이고, 중심에서 멀어질수록 광원의 효과가 감소한다. 스폿 광원의 효과를 위해서는 보통 좀 더 복잡한 방법이 사용이 되는데, 이에 대한 설명은 생략하기로 하겠다. 한 가지 주의를 할 것은 현재 슬라이드 프로젝터가 눈 좌표계를 기준으로 하여 설정이 되어 있으므로, 카메라를 움직이면 슬라이드 프로젝터도 카메라에 고정되어 따라 움직이게 된다. 종종 슬라이드 프로젝터를 세상 좌표계에서 설정하는 것이 더 자연스러울 수가 있는데, 이를 위해서는 약간의 추가 작업이 필요하다. 특히 현재 모델뷰 행렬의 역행렬을 계산해야 하는데, 이에 대한 구현은 연습 문제로 남기기로 한다.

투영 텍스춰에 대한 설명을 마치기 전에 OpenGL의 구현에 있어 중요한 사항 한 가지에 대하여 간략히 알아보자. 4장의 3.6절에서 설명한 원근 교정을 통한 선형 보간은, 특히 보간을 하려하는 데이터 값 $f$가 $s$, $t$, $r$과 같이 텍스춰 좌표인 경우 그 효과가 두드러진다고 하였다. 따라서 실시간 렌더링에 있어 텍스춰 매핑의 중요성을 생각해보면 텍스춰 좌표의 원근 교정은 매우 중요한 사항이라 할 수가 있다. 앞에서는 $q$ 좌표에 대하여 고려를 하지 않았었는데, 윈도우 좌표계에서 래스터화를 하려하는 삼각형의 꼭지점에 대하여 연관된 텍스춰 좌표가 $(s, t, r, q)$이라면, $s$, $t$, $r$ 좌표에 대하여 실제로는 $\frac{s}{q}$, $\frac{t}{q}$, $\frac{r}{q}$ 값이 사용이 되어야 한다.

여기서는 자세한 설명은 생략하고 요점만 간단히 말하겠다. 앞에서 각 텍스춰 좌표 $f(f = s, t, r)$에 대하여 $\frac{f}{w}$ 값이 윈도우 좌표계 공간에서 선형적으로 변한다고 했는데, 그뿐만 아니라, $\frac{q}{w}$ 값도 선형적으로 변하게 된다. 따라서 각 데이터 값에 대하여 점진적으로 선형 보간을 한 후, $\frac{\frac{f}{w}}{\frac{q}{w}} = \frac{f}{q}$와 같이 나눗셈을 하면 원근 교정뿐만 아니라 $q$ 좌표까지 고려한 정확한 텍스춰 좌표를 구할 수가 있다. 이를 위하여 앞에서 설명한 원근 교정 방법을 약간 수정을 해야하는데, $\frac{1}{w}$ 대신에 $\frac{q}{w}$ 값을 사용하여 선형 보간을 하면 된다. 여기서 모든 화소에 대하여 $\frac{q}{w}$ 값을 구하기 위하여 나눗셈을 한 번씩 하는 것이 아니라, 래스터화 계산에 들어가기 전에 삼각형의 각 꼭지점에 대하여 한 번씩만 해당 $q$ 값을 $\frac{1}{w}$에 곱해서 선형 보간을 하면 되므로, 래스터화 과정에 있어 추가적인 계산 부담은 거의 없다고 할 수가 있다.

결론적으로 말하면 텍스춰 좌표에 대해서는 431쪽의 식 (4.4)과 같은 방식이 아니라, 다음과 같은 형태의 선형 보간을 사용하는데 바로 이것이 OpenGL에서 사용하는 선형 보간의 기본식이다[14].

$$f_t = \frac{(1-t)\cdot\frac{f_a}{w_a} + t\cdot\frac{f_b}{w_b}}{(1-t)\cdot\frac{q_a}{w_a} + t\cdot\frac{q_b}{w_b}} \tag{5.3}$$

## 6.3 구 매핑을 통한 퐁 쉐이딩

3장에서는 뷰잉과 조명 인자들을 설정하였을 때, 눈 좌표계 공간에서 물체의 각 꼭지점에서 반사가 되어 화면에 투영이 되는 빛의 색깔을 계산하기 위하여 사용되는 퐁의 조명 모델에 대하여 살펴보았다. 이러한 조명 모델은 앞에서도 설명한 바와 같이 래스터화 과정에서 사용되는 다면체 모델을 위한 쉐이딩 모델과는 분명히 구별

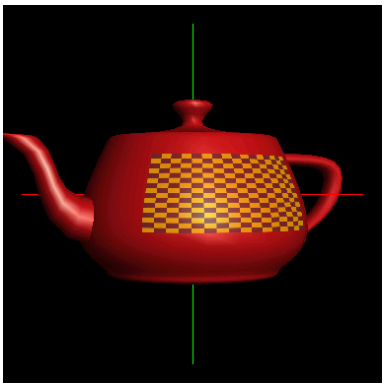
[14] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.1.1),* Silicon Graphics Inc., 1998의 66쪽의 식 (3.2)와 71쪽의 식 (3.4).

이 되어야 한다. 이 쉐이딩 모델은 선분과 다각형과 같은 기하 프리미티브들이 윈도우 좌표계로 변환이 된 후 래스터화가 될 때, 눈 좌표계에서 구한 꼭지점에 연관된 색깔을 사용하여 자신이 투영되는 영역 안의 화소의 색깔을 계산하는데 사용되는 모델이다.

앞에서도 기술한 바와 같이 OpenGL에서는 플랫 쉐이딩과 스므드 쉐이딩 등 래스터화 과정에서 두 가지의 쉐이딩 모델을 제공한다. 전자는 꼭지점에 연관된 색깔 중 하나를 선택하여 프리미티브 내부의 전체 화소를 칠하는 방법이고, 후자는 화소의 위치에 따른 비율에 따라 선형 보간을 통하여 꼭지점의 색깔을 부드럽게 혼합하는 것이다라고 하였다. 좀 더 정확하게 말하면 이 방법은 널리 쓰이는 스므드 쉐이딩 방법의 두 가지 중 하나로서, 보통 고우러드 쉐이딩(Gouraud shading)이라 부른다.

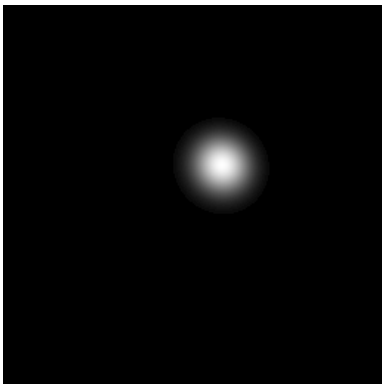
다른 하나는 퐁 쉐이딩(Phong shading)이라고 하는 방법인데, 고우러드 쉐이딩과 다른 점은 기하 프리미티브의 꼭지점에서 뿐만이 아니라 기하 프리미티브가 투영되는 각 화소에 대응되는 지점들에 대하여 퐁의 조명 모델을 적용한다는 점이다. 퐁 쉐이딩 방법을 사용하면 래스터화 과정에서 꼭지점의 색깔에 대하여 선형 보간을 하는 것이 아니라, 꼭지점에 대응이 되는 법선 벡터를 사용하여 각 화소에 대한 벡터를 선형 보간을 통하여 구한다. 이후 이 벡터를 눈 좌표계 공간으로 역변환을 한 뒤 퐁의 조명 모델을 적용하여 물체의 색깔을 구해, 이 색깔로 해당 화소를 칠하는 방식을 취한다. 퐁 쉐이딩은 일반적으로 고우러드 쉐이딩보다 훨씬 우수한 렌더링 결과를 산출한다. 반면에 고우러드 쉐이딩 방법보다 구현에 필요한 계산량이 많기 때문에, 실시간 렌더링 시스템에서는 보통 고우러드 쉐이딩을 채택하여 구현을 한다.

(a) OpenGL의 스므드 쉐이딩

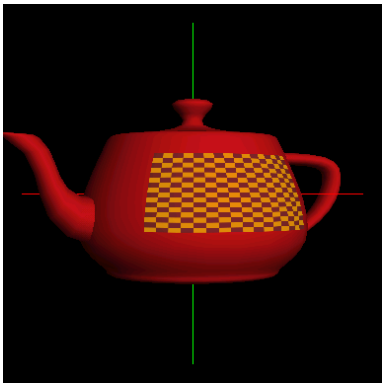


(b) OpenGL의 스므드 쉐이딩

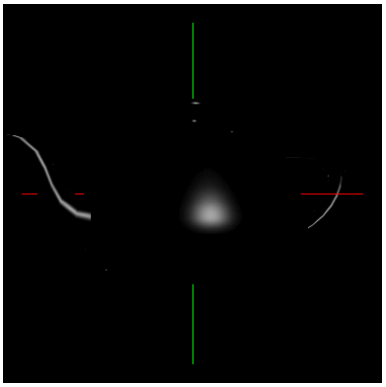
(c) 구 매핑에 의한 퐁 쉐이딩



(d) 구 맵



(e) 정반사를 제외한 렌더링



(f) 구맵에 의한 하이라이트

그림 5.24: OpenGL의 스므드 쉐이딩과 구 매핑에 의한 퐁 쉐이딩

하지만 고우러드 쉐이딩은 적지 않은 문제를 야기하기 때문에, 실시간 렌더링 시스템에서 퐁 쉐이딩 효과를 시뮬레이션 해주는 기법들에 대하여 지속적으로 연구가 되어 왔다. 그림 5.24의 (a)와 (b)는 고우러드 쉐이딩을 통하여 렌더링한 결과이다. 지금 물주전자에 자연스럽게 하이라이트가 생성이 되도록 하였는데, 이 두 이미지는 고우러드 쉐이딩의 문제를 잘 보여주고 있다. 우선 그림 (a)를 자세히 보면 많은 고우러드 쉐이딩의 결과에서 그러하듯이, 하이라이트가 지나치게 부드럽게 형성이 됨을 알 수가 있다. 일반적으로 금속성의 물체에서처럼 정반사를 많이 하는 물체를 보면 하이라이트가 적은 영역에 대하여 강하게 형성이 됨을 알 수가 있다. 고우러드 쉐이딩은 꼭지점의 색깔을 다각형 영역에 대하여 보간을 하므로, 상대적으로 하이라이트가 강하게 형성되는 꼭지점의 색깔이 그렇지 않은 지역으로 '번지므로' 그러한 현상이 발생하는 것이다.

또 다른 고우러드 쉐이딩의 문제는 선형 보간이 된 색깔이 물체 표면에서 부드럽게 변하는 것이 아니라, 종종 '파열이 되는 듯한' 결과를 생성한다는 점이다(그림 5.25). 특히 이러한 현상은 종종 다각형의 크기가 상대적으로 크고 쉐이딩이 된 색깔이 급격히 변하는 지역에서 다각형의 각 꼭지점에 대하여 계산된 색깔의 차이가 크기 때문에 발생한다. 이 문제를 줄이기 위한 한 가지 방법은 기하 물체를 더 많은 다각형을 사용하여 세밀하게 모델링을 하는 것이지만, 이는 실시간 렌더링 소프트웨어의 제작에 있어 바람직한 방법은 아니다.

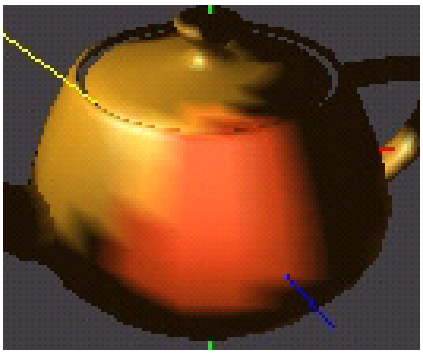


그림 5.25: 고우러드 쉐이딩의 문제

또 다른 문제는 엄격히 말하면 고우러드 쉐이딩의 문제라기 보다는 OpenGL의 렌더링 파이프라인의 구조에서 기인하는 문제라 할 수 있다. 그림 5.24(b)를 보면

텍스춰를 입힌 부분의 하이라이트가 상당히 약해진듯한 모습을 관찰할 수가 있다. 앞에서도 언급한 바와 같이 OpenGL에서는 퐁의 조명 모델을 통하여 구한 꼭지점 색깔을 고우러드 쉐이딩을 하여 보간한 화소의 색깔 위에 텍스춰 색깔을 입힌다. 따라서 어떠한 혼합 모드를 사용하건 텍스춰 색깔을 정반사 색깔인 하이라이트를 포함한 물체의 기반 색깔에 덧칠하는 결과가 되므로, 상대적으로 하이라이트가 잘 안 보이게 되는 것이다.

좀 더 낳은 방법은 우선 앰비언트 반사와 난반사만을 통하여 물체의 기반 색깔을 구하고, 이에 텍스춰를 입힌 후 여기에 하이라이트에 해당하는 정반사 색깔을 더해주는 것이다. OpenGL 버전 1.2부터는 정반사의 색깔의 분리가 가능하고, 실제로 정반사 색깔이 텍스춰 매핑 후 더해지게 할 수가 있는데, 이를 통하여 하이라이트가 약해지는 듯한 문제는 어는 정도 해결할 수가 있지만, 여전히 고우러드 쉐이딩에 기반을 두고 있기 때문에 하이라이트가 물체 표면 상에서 번지거나 파열이 되는 듯한 문제는 여전히 남는다.

이러한 문제를 극복하기 위한 방법 중의 하나는 구 매핑 기법을 통하여 하이라이트를 생성하는 것이다. 그림 5.24(c)는 지금 설명을 하고자 하는 예제 프로그램 5.D를 실행시켜, 퐁 쉐이딩을 사용한 하이라이트 효과를 시뮬레이션한 예를 보여주고 있다. 그림 (b)와 비교하여 하이라이트가 훨씬 강하고 자연스럽게 나타나고 있음을 알 수가 있다. 이 방법을 사용하려면 우선 하이라이트 생성에 필요한 구맵을 생성하여야 한다. OpenGL의 구 매핑 계산은 눈 좌표계에서 수행이 되므로, 구맵 또한 눈 좌표계를 기준으로 하여 미리 계산을 해놓아야 한다. 앞에서도 설명한 바와 같이 구맵의 한 지점 $(s,t)$는 눈 좌표계에서의 방향 $\mathbf{r} = (r_x\ r_y\ r_z)^t$에 대응이 된다. 따라서 구맵의 $(s,t)$ 좌표에 해당하는 각 텍셀에 대하여, 그와 매핑이 되는 $\mathbf{r}$ 벡터에 대한 하이라이트 색깔을 구해주어야 하는데, 여기서는 정반사 색깔 $I_{l\lambda} \cdot k_{s\lambda} \cdot (R \cdot V)^n$

중(293쪽의 설명 참조) $(R \cdot V)^n$에 해당하는 색깔이 저장되어야 한다.

보통 구 매핑을 통한 퐁 쉐이딩을 구현을 할 때, 뷰어는 지역 관찰자, 그리고 광원은 평행 광원으로 가정을 한다. 이러한 상황에서 빛이 들어오는 방향의 반대 방향에 해당하는 단위 벡터를 $L$이라 하면, $(R \cdot V)^n$ 값과 $(\mathbf{r} \cdot L)^n$ 값이 일치하기 때문에, 쉽게 구맵의 내용을 계산할 수가 있다. 다만 문제는 각 텍셀에 해당하는 $(s,t)$ 좌표에 대응이 되는 벡터 $\mathbf{r}$을 어떻게 구할 것인가 하는 것이다. 이는 구 매핑을 다룰 때 설명한 두 값간의 매핑의 역매핑에 해당하는 것으로서, 다음과 같이 유도할 수가 있는데, 그림 5.24(d)는 그림 (c)를 렌더링하는데 사용한 구맵의 이미지를 보여주고 있다.

$$\begin{aligned} r_x &= 2(2s-1)\sqrt{-4s^2+4s-1-4t^2+4t} \\ r_y &= 2(2t-1)\sqrt{-4s^2+4s-1-4t^2+4t} \\ r_z &= -8s^2+8s-8t^2+8t-3 \end{aligned}$$

일단 구맵을 만들어 놓으면, 아래의 프로그램 예 5.3에서와 같이 퐁 쉐이딩에 기반한 하이라이트를 생성을 할 수가 있다. 여기서의 기본 원리는 우선 앰비언트 반사와 난반사만을 사용하여 물체를 한 번 그린 후(그림 5.24(e)), 바로 그 결과 이미지에 구 매핑을 통하여 구한 하이라이트(그림 5.24(f))를 더하여 자연스러운 하이라이트를 생성하는 것이다. 이에 관하여 설명을 하면, 우선 Line (a)에서는 깊이 버퍼링에 대한 비교 함수를 GL_LESS로 설정을 한다. 깊이 버퍼링에 대해서는 자세히 설명을 하지는 않았는데, 이 함수는 현재 처리하려는 프래그먼트의 깊이 값 $z_{wd}$가 깊이 버퍼의 해당 화소의 깊이 값보다 작으면, 다시 말해서 지금 처리하는 프래그

먼트가 더 가까이 있으면, 이 프래그먼트의 색깔을 색깔 버퍼에, 그리고 깊이 값을 깊이 버퍼에 넣어주라는 것을 의미한다[15]. 따라서 매 순간 깊이 버퍼에는 현재까지 그린 기하 물체 중 가장 가까운 지점까지의 거리가 저장이 되는데, GL_LESS가 깊이 버퍼링의 디폴트 모드이다. 다음 Line (b)와 그 다음 문장에서 물체에 입힐 텍스춰를 바인딩 해준 후, Line (c)에서 광원의 정반사 색깔을 $(0, 0, 0, 1)$로 설정을 한다. 이런 상황에서 물주전자를 그리면(Line (d)), 물체의 기반이 되는 색깔, 즉 앰비언트 색깔과 난반사 색깔로 라이팅 계산을 한 후 텍스춰를 입혀 그 결과를 색깔 버퍼에 그려주게 된다. 즉 여기까지의 렌더링 결과가 그림 5.24(e)에 도시되어 있는데, 이제 그 위에 하이라이트를 적절히 입혀주어야 한다.

**프로그램 예 5.3** 퐁 쉐이딩에 기반한 하이라이트 생성 예.

```
void display (void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glDepthFunc(GL_LESS); // Line (a)
       ⋮       // set viewing and projection transforms
  glBindTexture(GL_TEXTURE_2D, tex_name[1]); // Line (b)
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
  glLightfv(GL_LIGHT0, GL_SPECULAR, null_color); // Line (c)
  draw_axes();
  draw_object(&teapot); // Line (d)

  glDisable(GL_LIGHTING); // Line (e)
  glColor3f(0.65, 0.65, 0.65); // Line (f)
  glDepthFunc(GL_EQUAL); // Line (g)
  glBindTexture(GL_TEXTURE_2D, tex_name[0]); // Line (h)
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);
  glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
  glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
  glEnable(GL_TEXTURE_GEN_S);
```

[15] 물론 프래그먼트별 연산의 다른 검사를 통과하면.

```
    glEnable(GL_TEXTURE_GEN_T);
    glBlendFunc(GL_ONE, GL_ONE); // Line (i)
    glEnable(GL_BLEND);
    draw_object(&teapot); // Line (j)
    glDisable(GL_BLEND);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    glEnable(GL_LIGHTING);
        ⋮
    glutSwapBuffers();
}
```

이를 위하여 물주전자를 한 번 더 그려주어야 하는데, 우선 Line (e)에서와 같이 조명 계산을 하지 않도록 설정한다. 대신 렌더링하려는 모든 다각형에 대하여 색깔을 $(0.65, 0.65, 0.65)$로 설정을 한다(Line (f)). 다음 깊이 버퍼링의 깊이 함수로 GL_EQUAL을 사용하는데(Line (g)), 이는 프래그먼트의 깊이 값이 현재 깊이 버퍼의 깊이 값과 같을 경우에만 색깔 버퍼에 그리라는 것을 의미한다. 다시 말해서 두 번째 물주전자를 그릴 때에는 물주전자의 어느 부분이 보이는지를 알 수가 있으므로, 그 부분에 대해서만 뒤에서 계산할 하이라이트 색깔을 더해주라고 설정을 하고 있는 것이다. 다음 Line (h)와 그 아래의 문장에서 미리 계산을 해놓은 구맵을 현재 텍스춰로 설정를 한 후, 텍스춰 매핑 과정에 필요한 인자들을 설정을 한다.

여기서 텍스춰 적용 모드는 GL_MODULATE를 사용을 한다. 앞에서 텍스춰 매핑 과정의 기본 목적은 눈 좌표계에서 꼭지점에 붙여준 물체의 기반 색깔과 텍스춰 이미지를 액세스하여 가져온 텍스춰 색깔을 혼합하는 것이라 하였다. 지금 라이팅 계산을 하지 않으므로, 물체의 기반 색깔은 현재 색깔로 지정이 된 $(0.65, 0.65, 0.65)$가 될 것이다. 한편 현재 구 매핑을 하기 때문에 전기한 바와 같이 텍스춰 색깔로 하이라이트의 강도를 나타내는 $(R \cdot V)^n$가 구해질 것이다. GL_MODULATE 모

드에서는 물체의 기반 색깔과 텍스춰 색깔의 각 채널끼리 곱하므로, 현재 색깔 $(0.65, 0.65, 0.65)$는 정반사가 되는 색깔 $I_{l\lambda} \cdot k_{s\lambda} \cdot (R \cdot V)^n$ 중 $I_{l\lambda} \cdot k_{s\lambda}$에 해당한다고 생각하면 된다. 즉 여기서 현재 색깔로 지정하는 색깔로 하이라이트의 색깔이나 강도를 조절할 수가 있다.

이러한 모드에서 물주전자를 다시 한 번 그리면, 바로 그림 5.24(f)의 내용이 색깔 버퍼에 그려지게 될 것이다. 여기서의 문제는 OpenGL에서는 렌더링을 할 때 디폴트로 현재 들어오는 프래그먼트의 색깔이 해당 화소에 이미 들어 있던 색깔에 덮어쓰므로, 단순히 한 번 더 렌더링을 하면 그 결과로 그림 5.24(f)와 같은 내용을 얻게 될 것이다. 지금 우리가 원하는 것은 그러한 것이 아니라, 첫 번째 렌더링의 결과 색깔 버퍼에 들어간 내용과 두 번째 렌더링의 결과 생성되는 내용을 그대로 더하는 것이다. 이를 위해서 OpenGL에서의 혼합(blending) 모드를 적절히 설정해주어야 한다. Line (i)와 그 다음 문장이 바로 이를 위한 것인데, 이를 통하여 두 번째 렌더링하는 물주전자의 색깔, 즉 하이라이트 색깔이 이미 색깔 버퍼에 있던 내용에 더해지게 된다.

이렇게 함으로써 원하는 하이라이트 효과를 시뮬레이션 할 수가 있는데, OpenGL에서 기본적으로 제공하는 고우러드 쉐이딩 방법에서 하이라이트의 계산을 위하여 퐁의 조명을 꼭지점에 대하여 적용하는 것과는 달리, 이 방법에서는 물체가 투영되는 모든 화소, 다시 말해서 프래그먼트에 대하여 계산이 됨을 명심하기 바란다. OpenGL 렌더링 파이프라인에서 전자의 방법을 꼭지점별 계산(per-vertex computation)이라 하고, 후자의 방법을 프래그먼트별 계산(per-fragment computation)이라 부르는데, 후자의 방법이 더 사실적인 결과를 얻게 됨은 당연하다고 할 수가 있다. 지금 살펴본 예에서는 구맵의 내용은 광원의 방향에 의존하므로, 광원의 기하속성이 바뀌면 다시 생성을 해주어야 한다. 반면 물체가 움직일 때에는 다시 구맵

을 계산할 필요가 없는데, 예제 프로그램 5.D를 수행시켜 오른쪽 마우스 버튼을 눌러 회전을 시킨 후, 기존의 방법과 이 방법의 결과를 비교해보기 바란다.

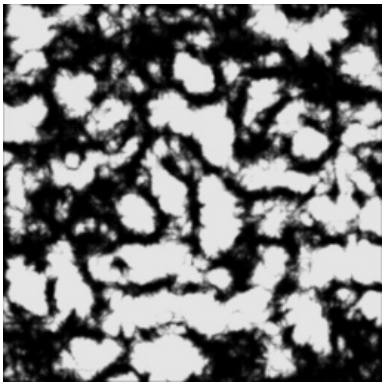
## 제 7 절 3차원 텍스춰 매핑

새로운 OpenGL 1.2 버전이 나오면서 기존의 1.1 버전과 다른 차이점 중의 하나는 3차원 텍스춰 매핑(3D texture mapping) 기법이 OpenGL의 기본 기능으로 채택이 되었다는 점이다. 이전까지만 해도 OpenGL 1.1의 특수한 확장 버전에서만 3차원 텍스춰 매핑이 가능했던 것과는 달리 1.2 버전을 지원하는 OpenGL 라이브러리는 이 기능을 반드시 제공하여야 한다.

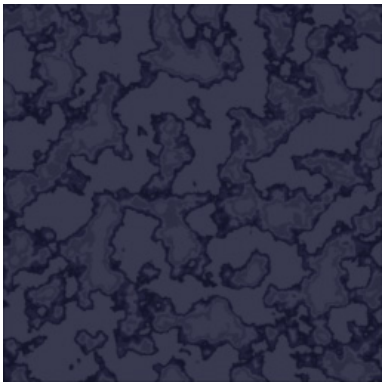
2차원 텍스춰 매핑은 특히 실시간 렌더링에 있어 사실적인 효과를 생성하는데 핵심적인 역할을 하는 반면, 2차원 평면의 이미지를 3차원 공간의 임의의 형태를 가지는 기하 물체에 입혀야 하는 계산 구조상 종종 텍스춰가 부자연스럽게 입혀지는 경우가 발생한다. 이러한 문제를 해결하기 위한 방편으로 솔리드 텍스춰 매핑(solid texture mapping)이라 하는 기법이 제안이 되었다. 여기서는 2차원이 아닌 3차원 텍스춰 이미지를 사용한다[16]. 간단한 예를 제시하면, 3차원 물체를 나무로 조각을 한 것과 같은 렌더링 효과를 생성하려고 한다고 가정하자. 이 때 나무에 대한 2차원 텍스춰를 사용하는 것이 아니라, 3차원 텍스춰 좌표의 영역 $[0,1]\times[0,1]\times[0,1]$에 대응이 되도록 육면체 형태의 원목에 해당하는 3차원 텍스춰 이미지를 생성해놓고, 기하 물체를 그 안에 집어넣는다고 가정을 한 상태에서 각 꼭지점에 대응이 되는 3차원 텍스춰 좌표 $(s,t,r)$을 설정을 하여 '3차원적으로' 텍스춰 매핑을 하면 된다.

이렇게 하면 마치 원목을 깍아 물체를 만든 것과 같은 자연스러운 텍스춰 매핑

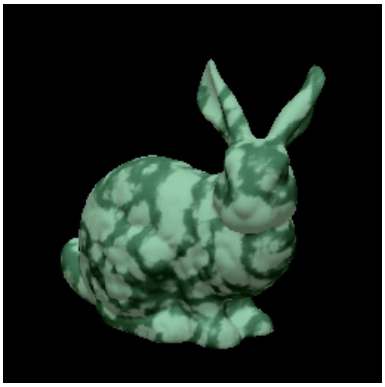
[16] 원래 솔리드 텍스춰 매핑에서는 꼭 3차원 텍스춰만 사용하는 것은 아니고, 응용 문제에 따라 그 이상의 차원의 텍스춰를 사용할 수도 있다.



(a) 3차원 텍스춰 단면 1



(b) 3차원 텍스춰 단면 2



(c) 렌더링 예 1

(d) 렌더링 예 2

그림 5.26: 3차원 텍스춰 매핑

효과를 생성할 수가 있다. 3차원 공간에서의 솔리드 매핑을 단순히 3차원 텍스춰 매핑이라고 하는데, 이러한 기법을 통하여 나무결, 대리석 무늬, 안개 등과 같은 자연 현상을 자연스럽게 렌더링 할 수 있다. 또한 CT나 MRI와 같은 3차원 의료 영상으로부터 실시간 볼륨 렌더링을 하는데 있어 3차원 텍스춰 매핑이 중요한 역할을 한다. 그림 5.26(a)와 (b)는 두 개의 3차원 텍스춰 이미지의 한 단면을 보여주고 있고, 그림 (c)와 (d)는 각각 그러한 텍스춰를 사용하여 3차원 텍스춰 매핑을 한 결과를 보여주고 있다.

OpenGL을 사용한 3차원 텍스춰 매핑은 사용하는 함수의 이름과 상수가 약간 다르다는 점을 제외하고는 기본적으로 2차원 텍스춰 매핑과 동일하다. 우선 glTexImage2D(*) 함수에 대응이 되는 함수는 void glTexImage3D(GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLsizei depth, GLint border, GLenum format, GLenum type, const GLvoid *texel);이다. 2차원의 경우와의 차이점은 첫 번째 인자로 GL_TEXTURE_3D를 사용해야하고, 텍스춰 이미지의 해상도를 설정할 때 차원이 하나 더 늘어나 depth 인자를 사용해야 한다는 점이다. 또한 3차원 텍스춰 좌표는 glTexCoord3f(GLfloat s, GLfloat t, GLfloat); 함수와 glTexCoord3fv(const GLfloat *v); 함수 등을 사용하면 된다. 또한 glEnable(GL_TEXTURE_3D);와 같이 2차원 텍스춰 매핑의 경우 GL_TEXTURE_2D 상수를 사용하는 OpenGL 함수에서 GL_TEXTURE_3D를 사용해야 한다.

3차원 텍스춰 매핑은 자연스러운 렌더링 효과를 내는데 있어 매우 유용하나 적지 않은 텍스춰 메모리를 요구한다는 점에서 주의를 해야한다. 예를 들어 GL_RGBA 타입의 $256 \times 256 \times 256$ 해상도를 가지는 텍스춰를 사용하려면 64 메가 바이트의 텍스춰 메모리가 필요하다. 물론 최근의 그래픽스 가속기들이 제공하는 텍스춰 메모리의 크기가 빠르게 증가하고는 있으나, 보통 렌더링을 하기 위해서 많은 개수의 텍스춰를 사용한다는 점을 고려할 때 이는 상당한 부담이 아닐 수가 없다. 따라서 3차원 텍스춰 매핑을 효과적으로 사용하기 위해서는 최근에 제안이 된 압축을 통한 3차원 텍스춰 매핑 방법과 같은 기법들이 고려가 되어야 할 것이다[17].

[17] C. Bajaj, I. Ihm, and S. Park. "Compression-Based 3D Texture Mapping for Real-Time Rendering," Graphical Models, Vol. 62, No. 6, pp. 391-410, November 2000.

# 제 6 장

# 나오는 말

## 제 1 절 텍스춰 매핑 이후의 계산

지금까지 OpenGL의 렌더링 파이프라인에서 텍스춰 매핑까지의 계산과정에 대하여 알아보았다. OpenGL과 같은 실시간 렌더링 시스템의 목표는 다면체 모델로 표현된 가상의 세상에 대하여 원하는 내용의 이미지를 생성하여 프레임 버퍼, 특히 색깔 버퍼에 저장을 하는 것이다. 텍스춰 매핑 계산을 마쳤을 때의 프래그먼트의 내용 $((x_{wd}, y_{wd}), z_{wd}, (r, g, b, a), (s, t, r, q))$는 최종 목적지인 프레임 버퍼에 저장할 데이터에 매우 근접한 상태이다. 여러 번 강조를 했듯이, 여기서 $(x_{wd}, y_{wd})$는 현재 프래그먼트에 해당하는 화소의 위치이고, $z_{wd}$는 이 화소를 통해서 보이는 기하 물체의 지점까지의 거리를 나타내며, $(r, g, b, a)$는 이 화소를 통해서 보이는 색깔이라 하였다. 마지막 $(s, t, r, q)$는 이 프래그먼트에 해당하는 지점의 텍스춰 좌표로서, 텍스춰 매핑 계산이 끝났으므로 현 단계에서는 무시하면 된다.

단순하게 생각하면 화소 $(x_{wd}, y_{wd})$에 해당하는 색깔 버퍼의 위치에 색깔 $(r, g, b, a)$를 저장하면 될 것 같지만, OpenGL에서는 아직 몇 가지의 계산 과정을

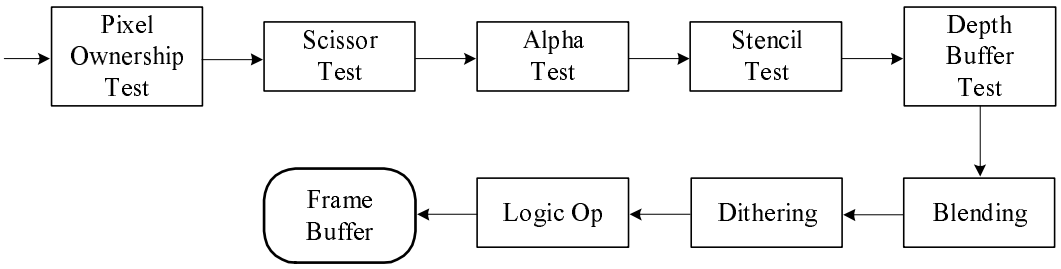



그림 6.1: 프래그먼트별 연산

더 거쳐야만 프래그먼트가 프레임 버퍼에 이르게 된다. 378쪽의 그림 4.1을 다시 보면, 텍스춰 매핑 이후의 나머지 렌더링 계산 과정이 간략히 도시가 되어 있다. 텍스춰 모듈에서 나온 프래그먼트들은 곧바로 안개(Fog) 모듈로 들어간다. 이 모듈은 안개 효과(fog effect)를 내기 위한 것인데, 계산을 하도록 설정을 하면 화면으로부터의 깊이 값 $z_{wd}$에 따라 안개 농도를 계산하여, 그에 맞게 프래그먼트의 색깔을 변경시켜 준다. 다음 눈 좌표계에서 조명 계산을 할 때 정반사 색깔을 분리하였다면, 색깔 합(Color Sum) 모듈에서 프래그먼트의 색깔과 라이팅 모듈에서 구한 정반사 색깔을 더해준다. 다음 포함율 적용(Coverage Application) 모듈에서 앤티앨리어싱을 위한 계산을 한 뒤, 프래그먼트별 연산(Per-Fragment Operations) 모듈로 들어가 마지막 계산을 수행한다.

프래그먼트별 연산 모듈은 그 이름이 의미하듯이 프래그먼트를 단위로 하여 계산을 수행한다. 사실 래스터화 계산을 통하여 프래그먼트가 생성이 되기 때문에 그 이후의 계산은 모두 프래그먼트 단위로 수행이 되지만, OpenGL에서는 이 부분을 따로 모아 프래그먼트별 연산이라 부른다. 그림 6.1에는 프래그먼트별 연산 모듈에서 차례대로 수행이 되는 여덟 가지 연산이 도시되어 있다. 앞쪽을 보면 다섯 개의 검사(test) 연산이 있는데, 이들은 현재 프래그먼트에 대하여 각각 고유한 검사를 해서 해당 조건을 만족시키면 다음 연산으로 통과를 시키고, 그렇지 않을 경우에는

프래그먼트를 제거한다. 따라서 여기의 다섯 개의 검사 연산을 모두 성공적으로 통과해야만 프래그먼트가 혼합(Blending) 모듈에 다다를 수가 있다.

혼합 모듈에서는 프로그래머가 OpenGL의 혼합 기능을 사용하도록 설정하였을 때, 그에 따라 프래그먼트의 색깔을 적절히 바꾸어 준다. 그 이후 디더링(Dithering) 모듈에서 필요에 따라 색깔을 변경한 후, 그 색깔을 논리 연산(Logical Operations) 모듈을 거쳐 색깔 버퍼의 해당 화소의 위치에 저장을 한다. 이렇게 함으로써 길고 긴 여정을 거쳐 프레임 버퍼에 적절한 이미지 데이터가 저장이 되는 것이다. 텍스춰 매핑 이후의 계산은 흘러 들어오는 각 프래그먼트를 프레임 버퍼에 저장할 지를 결정을 한 후, 프래그먼트에 대하여 마지막 손질을 하여 프레임 버퍼에 저장을 해주는 과정이라 보면 된다. 여기서는 이 책을 마무리하기 전에 프래그먼트별 연산 중 깊이 버퍼 검사와 혼합에 대하여 간략히 알아보겠다.

## 제 2 절 깊이 버퍼와 깊이 테스트

프래그먼트별 연산 모듈의 다섯 개의 검사 중 깊이 버퍼 검사(depth buffer test)는 은면 제거 계산을 수행하기 위한 검사이다. 이 책에서는 자세히 다루지는 않았지만, 은면 제거는 기하 변환, 조명, 래스터화, 텍스춰 매핑 등과 함께 렌더링 계산의 중요한 한 부분을 구성한다. 가상의 세상을 렌더링할 때, 가장 중요한 요구 사항 중의 하나는 카메라를 통하여 보이는 면만 그려주어야 한다는 점이다. 다시 말에서 앞쪽에 보이는 면에 의해 가리는 뒤쪽의 은면을 제거하여 그리지 않도록 해주어야 한다.

OpenGL과 같은 렌더링 시스템에서는 기하 물체를 그리기 위해서 물체를 구성하는 각 기하 프리미티브들을 하나씩 그려주어야 한다. 카메라를 임의의 위치에 배치를 한 후 기하 물체들을 그릴 경우, 기하 프리미티브를 그리는 순서에는 카메라를 기준으로 하여 어떠한 전후 관계도 존재하지 않는다. 따라서 적절한 처리가 없이

단순히 렌더링 파이프라인에 들어오는 순서대로 기하 프리미티브들을 프레임 버퍼에 그릴 경우, 은면 제거가 제대로 되지 않은 잘못된 이미지가 생성이 될 것이다.

어떻게 보면 장면이 복잡할 때 은면 제거를 하는 것은 그리 수월한 작업처럼 보이지는 않는다. 실제로 많은 사람들이 오랜 기간 동안 효과적인 은면 제거 알고리즘을 개발해왔다. 그 결과 적지 않은 수의 알고리즘이 존재하는데, 문제는 OpenGL 시스템과 같이 하드웨어로 구현이 되어야 하는 실시간 렌더링 시스템에서 어떤 방식의 은면 제거 알고리즘을 채택하여 구현할 것인가 하는 것이다. 대부분의 은면 제거 알고리즘들은 성능을 높이기 위하여 비교적 복잡한 방법을 사용하기 때문에, 하드웨어적으로 구현하는 것이 적절하지 못하다. 그러나 다행히도 아주 단순하면서도 실시간 렌더링 계산 구조에 맞도록 쉽게 구현할 수 있는 방법이 있는데, 바로 그것이 깊이 버퍼링(depth buffering) 방법이다. 앞에서도 간략히 설명을 하였듯이 Z 버퍼링(Z buffering)이라고 하는 이 방법은 프레임 버퍼의 한 요소인 깊이 버퍼를 사용하여 은면 제거를 한다.

깊이 버퍼 검사를 하도록 설정을 하면, 깊이 버퍼는 매 순간 그 때까지 그린 기하 프리티브들 중 각 화소에 대하여 가장 가까운 지점까지의 거리 정보를 제공한다. 래스터화의 결과 생성된 프래그먼트들이 흘러들어올 때, 깊이 버퍼 검사 모듈에서는 프래그먼트의 깊이 값 $z_{wd}$과 화소 $(x_{wd}, y_{wd})$에 해당하는 깊이 버퍼의 깊이 값과 크기 비교를 한다. 만약 현재 프래그먼트의 깊이 값이 더 크다면, 이는 현재 프래그먼트보다 카메라에 더 가까운 프래그먼트를 이미 그렸다는 것을 의미한다. 따라서 현재 프래그먼트는 다른 기하 프리미티브에 의해 가리므로 단순히 렌더링 파이프라인에서 제거를 하면 된다. 만약 현재 처리하는 프래그먼트의 깊이 값이 더 작다면, 이는 화소 $(x_{wd}, y_{wd})$에 대하여 이 프래그먼트에 해당하는 지점이 이미 그린 기하 프리미티브들보다 더 가깝다는 것을 의미하므로, 색깔 버퍼와 깊이 버퍼에 해

(a) 렌더링된 이미지 (b) 깊이 버퍼의 최종 내용

그림 6.2: 깊이 버퍼의 내용

당 색깔과 깊이 값을 넣어주면 된다. 이렇게 함으로써 깊이 버퍼 안의 깊이 정보를 올바르게 유지를 할 수가 있다. 모든 렌더링이 끝났을 때에는 각 화소에는 최종 이미지에 나타나는 물체까지의 거리가 저장이 된다. 그림 6.2(b)는 (a)의 이미지를 렌더링하였을 때 깊이 버퍼에 최종적으로 저장이 된 내용을 보여주고 있다. 여기서는 거리가 멀 수록 밝게 보이도록 했는데, 그림 (b)로부터 깊이감을 느낄 수 있을 것이다.

깊이 버퍼에는 보통 화소 당 24 비트 또는 32 비트가 필요하기 때문에, 메모리 가격이 비쌌던 과거에는 깊이 버퍼링을 하드웨어로 구현하는 것이 상당한 부담이었다. 그러나 메모리 가격의 하락으로 인하여 이제는 깊이 버퍼링은 실시간 렌더링 파이프라인에서 표준처럼 사용이 되고 있다. OpenGL에서 깊이 버퍼링을 사용하는 방법은 간단하다. 우선 깊이 버퍼링을 하기 위해서는 깊이 버퍼가 필요하므로, 시스템에 깊이 버퍼를 요청해야하는데, GLUT를 사용할 경우에는 334쪽에서 설명한 대로 다음과 같이 GL_DEPTH 상수를 추가하여 glutInitDisplayMode(*) 함수를 호출하면 된다.

```
glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
```

일단 깊이 버퍼를 할당을 받고 난 후, glEnable(GL_DPETH_TEST); 문장을 수행시켜, 렌더링 계산 중 깊이 버퍼링 계산을 하라고 명시적으로 설정을 해야 한다. 다음 깊이 버퍼를 적절하게 초기화를 해주어야 한다. 이는 다음과 같이 glClear(*) 함수를 사용하여 프레임 버퍼를 초기화할 때 깊이 버퍼도 같이 초기화를 해주라고 하면 되는데,

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

이 결과 깊이 버퍼의 각 화소에는 가장 먼 거리에 해당하는 값, 다시 말해서 깊이 버퍼가 저장할 수 있는 가장 큰 값이 저장이 된다.

이 정도가 전형적인 은면 제거를 하기 위하여 호출하는 OpenGL 관련 함수들이다. 실제로 OpenGL에서는, 다른 버퍼들에 대해서도 마찬가지이지만, glClearDepth(*) 함수, glDepthFunc(*) 함수, glDepthMask(*) 함수 등을 통하여 깊이 버퍼를 다양하게 사용을 할 수 있는 방법을 제공하는데, 그러한 사용법에 대한 이해는 연습 문제로 남기기로 하겠다.

## 제 3 절 혼 합

3장의 3절에서 RGBA 색깔 모드에 대하여 기술을 할 때, A 채널, 즉 알파 채널의 의미와 사용법에 대하여 설명을 하였다. 화소의 알파 값을 사용하면 다양한 방식으로 이미지들을 합성할 수가 있다고 했는데, OpenGL에서는 효과적인 이미지의 합성을 위하여 혼합(blending) 기능을 제공한다. 앞에서도 설명한 바와 같이 혼합 계산은 다섯 가지의 검사를 통과한 프래그먼트를 단위로, 즉 화소 별로 수행이 된다. 혼합을 하려면 섞으려고는 하는 RGBA 색깔이 최소한 두 개가 필요한데, OpenGL에서는 현재 처리하려고 하는 프래그먼트의 색깔 $C_S = (R_S, G_S, B_S, A_S)$와 색깔 버퍼에 이 프래그먼트에 해당하는 화소에 저장되어 있는 색깔 $C_D = (R_D, G_D, B_D, A_D)$를

사용하여 혼합을 한다. 다시 말해서 렌더링 파이프라인을 따라 흘러 들어오는 색깔 $C_S$와 이미 색깔 버퍼에 들어가 있는 색깔 $C_D$을 혼합을 하는데, 전자의 색깔을 원시 색깔(source color), 그리고 후자의 색깔을 목적 색깔(destination color)이라 한다.

OpenGL에서는 이 두 색깔을 다음과 같은 방식으로 혼합을 하여,

$$(R_S \cdot S_R + R_D \cdot D_R, G_S \cdot S_G + G_D \cdot D_G, B_S \cdot S_B + B_D \cdot D_B, A_S \cdot S_A + A_D \cdot D_A)$$

그 결과로 생성되는 색깔로 현재 프래그먼트의 색깔을 대치한다. 결과적으로 혼합한 색깔로 색깔 버퍼를 칠하게 되는데, 여기서 두 색깔을 어떻게 혼합을 할지는 원시 색깔과 목적 색깔에 대한 가중치인 $(S_R, S_G, S_B, S_A)$와 $(D_R, D_G, D_B, D_A)$에 의해 결정된다. 혼합 가중치는 glBlendFunc(GLenum sfactor, GLenum dfactor); 함수의 두 인자 sfactor와 dfactor에 적절한 OpenGL 상수를 사용하여 설정을 할 수가 있다. 이에 대한 자세한 설명은 생략하려고 하는데, 여기서는 그림 6.3의 두 가지 혼합 예에 대하여 간략하게 알아보고 이 절을 마치려 한다.

그림 6.3(a), (b), (c)는 혼합을 하려고 하는 RGBA 이미지들인데, 각 이미지를 그림이 그려져 있는 유리판으로 생각을 해보자. 이미지 1은 모든 화소의 A 채널 값이 1로 설정이 된, 완전히 불투명한 유리판이다. 이미지 2는 각 화소의 A 채널 값이 0.75로 설정이 되어 있어, 뒤에서 들어오는 빛의 25%만 통과시키는 약간 불투명한 유리판이다. 마지막으로 이미지 3은 약간 복잡한데, 그림에서 흰색이 나타나는 지역에 있는 화소의 A 채널 값은 0으로, 그리고 나머지 화소에 대해서는 0.4로 설정이 되어 있다. 따라서 이 이미지에 해당하는 유리판의 경우 흰색에 대응되는 영역은 뒤에서 들어오는 모든 빛을 통과시키고, 나머지 부분은 60%만 통과시킨다.

그림 (d)는 세 개의 유리판을 이미지 3, 2, 1의 순서대로 앞에서 뒤로 가면서 배치를 하고 보았을 때의 모습을 합성한 이미지인데, 다음과 같은 코드를 사용하여 생성할 수가 있다.

(a) 이미지 1

(b) 이미지 2

(c) 이미지 3

(d) 혼합 예 1

(e) 혼합 예 2

그림 6.3: 이미지의 혼합

```
glEnable(GL_BLEND); // Line (a)
glBlendFunc(GL_ONE, GL_ZERO);
draw_image_1(); // Line (b)

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
draw_image_2(); Line (c)
draw_image_3(); Line (d)
glDisable(GL_BLEND);
```

우선 OpenGL의 혼합 기능을 사용하려면, Line (a)에서와 같이 그러한 기능을 명시적으로 요청해야 한다. 다음 Line (b)에서 이미지 1을 그리기 직전에, 원시 색깔과 목적 색깔의 가중치를 각각 GL_ONE과 GL_ZERO로 설정한다. 이는 $(S_R, S_G, S_B, S_A)$와 $(D_R, D_G, D_B, D_A)$를 각각 $(1, 1, 1, 1)$과 $(0, 0, 0, 0)$으로 설정을 하겠다는 것을 의미하는 것으로서, 그 결과 원시 색깔, 다시 말해서 현재 들어오는 프래그먼트의 색깔이 색깔 버퍼에 저장이 되는데, 사실 이러한 방식은 혼합 기능을 사용하지 않을 때의 디폴트 방식임은 명백하다.

이제 이미지 1을 색깔 버퍼에 그린 상태에서 Line (c)와 (d)에서 이미지 2와 3을 그리기 전에, 원시와 목적 색깔의 가중치를 각각 GL_SRC_ALPHA와 GL_ONE_MINUS_SRC_ALPHA로 설정을 해준다. 이 경우 전자는 원시 색깔의 A 채널 값을, 그리고 후자는 1에서 그 값을 빼준 값을 가중치로 사용하겠다는 것이므로, $(S_R, S_G, S_B, S_A) = (A_S, A_S, A_S, A_S)$와 $(D_R, D_G, D_B, D_A) = (1 - A_S, 1 - A_S, 1 - A_S, 1 - A_S)$인 설정 상태에서 혼합을 한다. 이는 3장의 3절에서 설명한 **over** 연산에 해당하는 설정으로서(278쪽 참조), 설정된 불투명도에 따라 한 이미지를 다른 이미지 위에 덮어쓰는 결과를 낳는다. 따라서 이미지 2와 3을 그렸을 때, 그림 6.3(d)와 같은 이미지가 생성된다.

만약 다음과 같은 코드를 수행한다면, 각 화소의 A 채널 값에 상관 없이 그림 (e)에서처럼 단순히 이미지 1과 2의 화소끼리 색깔 값을 더한 결과를 얻을 것이다.

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ZERO);
draw_image_1();

glBlendFunc(GL_ONE, GL_ONE);
draw_image_2();
glDisable(GL_BLEND);
```

이 외에도 OpenGL에서는 여러 가지 가중치 설정 방법을 제공하는데, 이들을 어떻게 조합을 하는가에 따라 다양한 혼합 효과를 낼 수가 있다.

## 제 4 절 이 책을 마치면서

지금까지 OpenGL이라는 실시간 렌더링 시스템을 통하여 3차원 컴퓨터 그래픽스 프로그래밍에 필요한 기초 이론과 실제에 대하여 살펴보았다. 서문에서도 언급한 바와 같이 이 책은 단순한 OpenGL 프로그래밍 가이드가 아니기 때문에, OpenGL의 모든 기능을 상세히 다루지는 않았다. 사실 디스플레이 리스트(display list) 기능이나 이미징 서브셋(imaging subset)을 비롯한 래스터 데이터의 처리 기능, 그리고 다면체 모델, 이차 곡면(quadrics), NURBS(nonuniform rational B-spline) 등의 기하 표현 도구 기능과 같은 중요한 OpenGL 기능들을 다루지 못한 것이 아쉬운데, 이는 이러한 기능들이 중요하지가 않기 때문이 아니고, 지면의 한계상 실시간 렌더링 파이프라인을 이해하는데 가장 시급한 주제들만 먼저 선택을 한 결과이다. 본 기초편은 이 시점에서 멈추려 하는데, 여기서 빠진 OpenGL 기능들과 실시간 렌더링을 위한 고급 기법들을 모아 이 책의 속편에 해당하는 응용편에서 설명을 하도록 하겠다.

# 찾아보기