RayCore® Programming Guide

Version 1.0

SILICONARTS

목 차

제 1장 소개

3차원 그래픽이란 컴퓨터로 3차원의 이미지나 영상을 생성하는 것을 말한다. 이는 컴퓨팅 기술의 한 분야로서 그래픽을 처리함에 있어서 여러 가지 어려운 문제점이 있다. 기본적으로 3차원 그래픽을 처리하기 위해서는 많은 연산량을 필요로 한다. 컴퓨터의 집적도가 높아지면서 이러한 문제는 어는 정도 해결되고 있지만, 이미지나 영상을 보다 자연스럽고 사실적으로 표현하기에는 아직도 부족하다. 이런 문제를 해결하고자 시도된 방법 중 대표적인 것이 바로 레이트레이싱 방식의 3차원 그래픽 기술이다.

레이트레이싱 방식은 1970년대에 이미 그 이론이 확립되었으나, 실제로 이를 컴퓨터로 처리하기 위해서는 상당히 많은 어려움이 있었다. 이 방식은 알고리즘 자체가 매우 복잡하고, 많은 검색을 필요로 하기 때문에, 상당히 복잡한 메모리 구조를 필요로 한다. 이는 컴퓨터의 집적도가 높아지는 것만으로는 해결하기가 어렵다. 이런 문제를 해결하기 위해서는 랜더링 처리 성능을 향상시킬 수 있도록 고도의 병렬화 기술과 같은 다양한 기법을 필수적으로 사용해야만 한다. 따라서, 레이트레이싱 랜더링을 매우 빠른 속도로 처리할 수 있는 랜더링 하드웨어의 연구를 진행하게 되었다. 이렇게 실리콘아츠㈜가 개발한 레이트레이싱 하드웨어 랜더링 기술을 RayCore®라 한다.

1.1 레이트레이싱(Ray Tracing) 기법

레이트레이싱(ray tracing) 기법은 이미지 평면의 픽셀에 입사되는 광선의 경로를 추적하여 반사(reflection), 굴절(refraction), 투과(transmission), 그림자(shadow) 등과 같은 다양한 광학적인 효과를 시뮬레이션하여 이미지를 생성하는 기법이다. 이 기법은 빛의 효과를 다양하게 표현할 수 있어서 전통적인 스캔라인* 랜더링 방법보다 훨씬 뛰어난 사실감을

* Scanline Algorithm – 삼각형의 내부를 채우기 위해 사용하는 알고리즘이다. 삼각형의 내부를 선 단위로 순차적으로 채워가는 방식이다.

제공한다. 하지만, 연산량이 매우 많기 때문에 이를 실시간으로 처리한다는 것은 매우 어려웠다. 그래서, 레이트레이싱 기법은 컴퓨터 게임과 같은 즉각적인 응답이 필요한 실시간 응용 프로그램보다는 영화, TV 등의 특수 효과처럼 랜더링하는데 시간이 많이 소요되더라도 실감 영상을 제작할 필요가 있는 응용분야에서 주로 사용되어 왔다.

화면 상의 특정 좌표에 대한 화소를 결정하기 위해서는 시점에서 해당 방향으로 광선을 생성해야 한다. 시점에서 출발한 광선들은 한 장면을 구성하는 모든 오브젝트들과 교차 테스트가 이루어진다. 가장 가까운 객체를 찾게 되면, 레이트레이싱 알고리즘은 우선 교차점으로 들어오는 광원에 대한 조명(lighting*) 연산을 수행한 후 해당 객체에 대한 물성 검사를 통하여 초기 색상을 산출하게 된다. 이 때, 반사 혹은 굴절의 특성에 따라 추가적으로 2차 광선†들이 파생된다. 이렇게 파생된 각각의 광선들로 산출되는 색들은 계속 누적되어 해당 화소의 최종색을 결정하는데 반영된다. 레이트레이싱 기법은 실세계에서 광선이 광원으로부터 시작되는 것과는 반대로 광선을 카메라로부터 시작하여 역방향으로 진행시킨다. 이렇게 역방향으로 광선을 추적하는 것이 광원에서 빛을 추적하는 것보다 훨씬 효과적이다. 왜냐하면, 광원으로부터 나오는 대부분의 광선이 반드시 카메라 뷰로 들어오는 것은 아니므로, 광선의 경로를 계산하는 데 훨씬 더 많은 시간이 필요하기 때문이다. 광원으로부터 광선을 투사하는(ray casting) 컴퓨터 시뮬레이션 기법 중의 하나로 Photon mapping이라는 방식이 있다.

따라서, 레이트레이싱에서 사용할 간단한 방법은 광선이 뷰 프레임(view frame)을 교차한다고 가정하고, 최대 반사 회수만큼 광선을 탐색하여 화소를 반복 계산하거나, 혹은 교차없이 정해진 특정 거리까지 광선을 탐색하여 화소를 계산하는 것이다. 화소값은 다양한 알고리즘으로 계산될 수 있는데, 고전적인 랜더링 알고리즘과 조도계산(radiosity) 같은 알고리즘이 포함될 수 있다.

a. 광선

레이트레이싱 랜더링에서 말하는 광선이란 특정 점에서부터 임의의 방향으로 이동하는 가상의 선을 의미한다. 랜더링에 있어서 광선의 종류는 시점으로부터 발생되는 광선과 물체로부터 발생되는 광선으로 구분할 수 있다. 시점에서 생성되어 발생된 광선을 1차 광선(Primary Ray)이라고 정의한다. 이는 스크린에 있는 특정 지점의 초기 화소값을 결정하기 위해 시점에서 해당 화소를 향하여 처음 발생되는 광선이다. 이 광선을 제외한 나머지 광선들은 2차 광선 (Secondary Ray)이라고 정의한다.

b. 1 차 광선(Primary Ray)

1차 광선은 시점에서 발생되는 광선으로 화면에 투영되는 물체를 찾는 주요 광선이다. 이 광선과 처음 만나는 물체의 위치에서 그 물체의 속성을 이용하여 2차 광선을 생성한

* **Lighting** – 광원과 입사각 반사각을 이용해서 색을 계산하는 방식으로 illumination 알고리즘이라고도 한다. 가장 많이 사용 방식이 Phong illumination 알고리즘이다. 다양한 효과를 나타내기 위한 다양한 알고리즘이 있으나 Phong 방식이 실시간 처리에 적합하다.

† **2차 광선(Secondary Ray)** – 그림자, 반사, 굴절, 투과의 효과를 얻기 위해서 생성하는 추가 광선들을 통칭한다.

다. 물체의 윤곽은 기본적으로 1차 광선에 의해 보여지게 된다.

한 화소당 하나의 1차 광선을 생성하여, 그 화소의 초기 색상을 결정하기 때문에, 화면 해상도 개수만큼 1차 광선이 생성된다. 따라서, 이러한1차 광선들을 화면에 그려질 물체를 찾기 위한 광선들로 간주할 수도 있다. 추가로 생성되는 광선은 1차 광선과 만나는 물체의 속성과 교차점의 위치에 따라 결정된다. 이렇게 생성되는 광선을 2차 광선이라고 정의한다.

c. 2 차 광선(Secondary Ray)

1차 광선으로부터 파생되는 모든 새로운 광선을 2차 광선이라고 한다. 이 광선은 그림자 광선, 반사 및 굴절 광선으로 구분할 수 있다.

그림자 광선은 광원과 광선이 만난 위치 사이에 그림자를 발생시키는 물체가 있는지를 확인하여, 그림자 효과를 나타내는 데 사용되는 광선이다. 만약 그림자가 나타날 환경이 아니라면, 광원에 대한 쉐이딩 연산을 수행하기 위해 필요한 정보를 얻는 데 사용되는 광선이다.

반사/굴절 광선은 물체의 속성에 따라 반사/굴절/투과의 효과를 나타내기 위해 추가적으로 생성되는 광선이다.

1.2 RayCore®의 레이트레이싱(Ray Tracing)

RayCore®는 레이트레이싱(ray tracing) 알고리즘을 하드웨어 파이프라인 구조로 구현한 하드웨어 기반 레이트레이싱 가속기이다. 일반적인 레이트레이싱 알고리즘의 처리 흐름은 그림 1과 같이 나타낼 수 있다. 이 알고리즘은 각 단계를 순차적으로 처리하며, 생성되는 모든 광선(1차 광선과 2차 광선)에 대해서 각 단계를 반복적으로 처리한다. 따라서, 여기에 필요한 시간과 비용이 매우 크다. 각 단계는 스크린 평면을 기준으로 광선을 생성하는 광선 생성 단계(ray generation), 생성된 광선에 근접하는 물체의 표면을 찾는 탐색 단계(traversal), 물체의 표면과 광선과의 교차점을 검사하는 교차점 검사 단계(intersection test), 광선과 물체 표면의 정확한 교차점을 계산하는 교차점 계산 단계(hit-point calculation), 교차점의 색을 결정하는 쉐이딩 단계(shading), 그리고, 교차점의 텍스처 색을 결정하는 텍스처 매핑 단계(texture mapping)로 구분된다. 간혹 단계를 간략히 표현하기 위해 쉐이딩 단계에 텍스처 매핑 단계를 포함시키기도 한다.

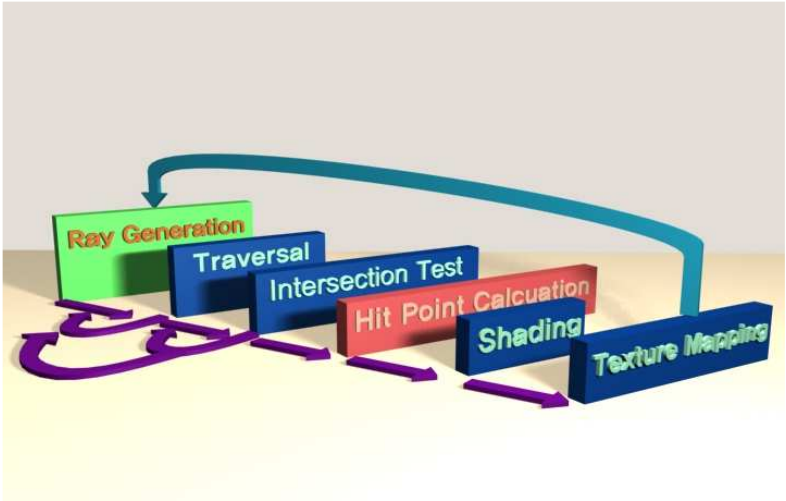

그림 1. 일반적인 레이트레이싱(ray tracing)의 처리 과정

앞서 말한 바와 같이, 스크린 평면상의 한 픽셀에서 생성된 주 광선(primary ray)이 그림자 광선(shadow ray), 반사 광선(reflection ray), 굴절 광선(refraction ray)을 파생시키고, 2차 광선인 반사광선과 굴절광선은 다른 물체의 표면과 부딪혀 또 다른 2차 광선을 파생시키기 때문에, 레이트레이싱 알고리즘은 소프트웨어적인 방식인 재귀적인 프로시저 호출로 구현된다. 하지만, 이는 랜더링의 성능을 저하시키는 주요 원인이 되기 때문에, 레이트레이싱 방식에서 영상 생성의 실시간 처리를 어렵게 만든다.

RayCore®는 레이트레이싱(ray tracing) 처리 절차의 모든 단계를 그림 2와 같이 하드웨어 파이프라인으로 설계하여 실시간 레이트레이싱(ray tracing)을 가능하게 하였다. 특히 탐색 단계(traverse)와 교차점 검사 단계(intersection test)를 하나로 통합한 T&I 유닛을 설계하여, 탐색과 교차점 검사의 효율성을 향상시켰다. 또한, RayCore®는 여러 개의 T&I 유닛이 병렬로 구성되어 있기 때문에, 서로 다른 데이터에 대하여 탐색과 교차점 검사를 MIMD 방식으로 동시에 처리할 수 있는 것이 특징이다.

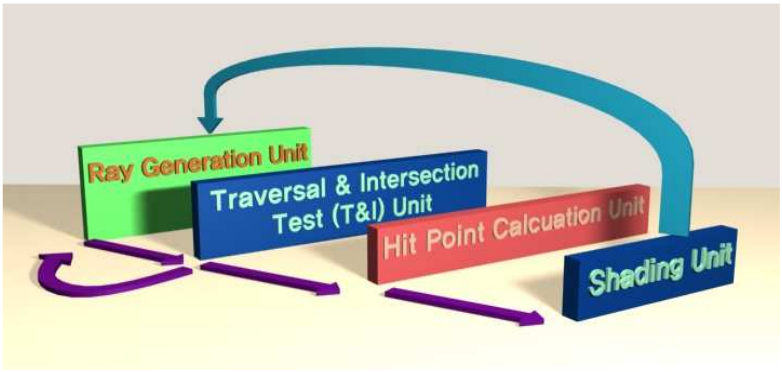

그림 2. RayCore®의 처리 과정

RayCore®는 광선 생성 유닛(ray generation unit), 탐색 및 교차점 검사 유닛(T&I unit), 교차점 계산 유닛(hit-point calculation unit), 그리고 쉐이딩 유닛(shading unit)으로 구성된다.

탐색 및 교차점 검사 유닛을 복수로 구성하는 것이 가능하며, 각각 다른 데이터를 병렬로 처리할 수 있다. 광선 생성 유닛은 할당된 스크린 블록 안에 있는 모든 화면좌표에 대한 1차 및 2차 광선을 생성한다. 탐색 및 교차점 검사 유닛은 생성된 광선에 대하여 모델링된 물체들을 탐색하여 교차점이 발생하는지를 검사한다. 만약 교차점이 발생하면 교차점 계산 유닛으로 교차되는 물체의 일부 데이터(primitive 정보)를 전송하여 교차점을 계산한다. 모델링된 모든 물체에 대해서 교차점이 발생하지 않으면, 다음 화면좌표에 대한 처리를 계속한다. 교차점을 계산한 후에는 해당 교차점의 물체 특성에 따라 추가 광선을 생성하고, 이 광선들에 대한 탐색과 교차점 검사를 계속한다. 쉐이딩 유닛은 계산된 교차점에 대한 Phong illumination 모델을 이용하여 그 점의 색을 결정하며, 텍스처 매핑이 사용된 경우에는 그 점의 텍스처 좌표와 텍스처 데이터를 이용하여 택스쳐 매핑을 처리한다.

또한, RayCore®는 스크린 분할 방식의 처리가 가능하기 때문에, 복수의 RayCore®를 병렬로 구성하여 Full HD급 이상의 고해상도를 지원할 수 있다.

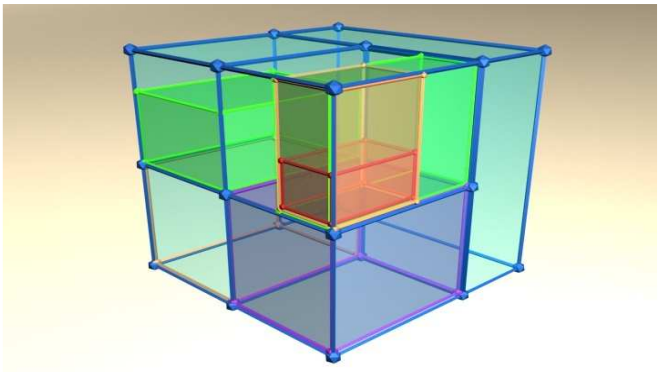

그림 3. 3차원 공간에서의 KD-Tree

a. 가속 구조(Acceleration Structure)

가속 구조의 사용은 실시간 레이트레이싱 랜더링을 위한 필요조건이다. 앞에서도 언급한 바와 같이 레이트레이싱 랜더링은 적합한 삼각형을 찾기 위한 검색이 중요한 작업으로 구성되어 있으며, 처리 속도에 가장 많은 영향을 미친다. 결국, 랜더링 처리 속도를 향상시키기 위해서는 삼각형에 대한 검색이 빨리 처리되어야 한다. 이를 해결하기 위해서 사용하는 방식이 가속 구조이다.

가속 구조는 KD-Tree혹은 BVH의 형태가 가장 많이 사용되고 있으며, 이들 구조가 처리 속도와 처리 효율면에서 가장 적합한 것으로 알려져 있다. 이 방식들은 공간을 이분할

하여 검색 효율을 향상시킨다. 특히 KD-Tree가 랜더링 속도면에서 보다 효율적이기 때문에 이를 기본 가속 구조로 사용한다.

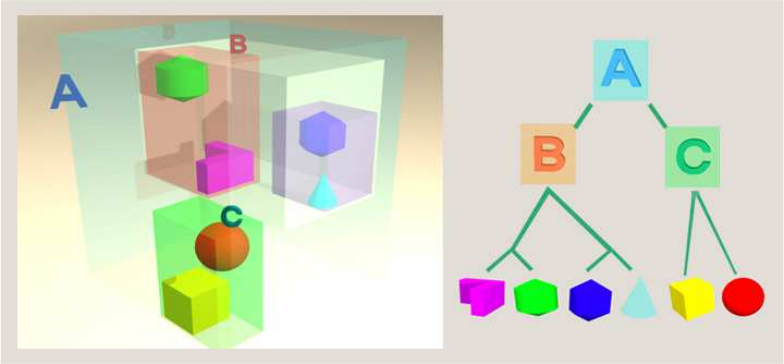

그림 4. BVH 개념도

KD-Tree는 이진 공간 분할 방식*을 통해서 구성된다. 공간을 나누는 분할 평면이 결정되면, 이 평면을 기준으로 공간이 분할되면서 동시에 삼각형도 분할될 수 있다. 이 경우 삼각형이 양쪽 공간에 모두 공유될 수도 있다. 이와 달리 BVH는 객체 분할† 방식을 이용한다. 이러한 특성의 차이로 인하여 랜더링 속도면에서 KD-Tree가 더 많은 장점을 가진다. KD-Tree의 단점은 가속 구조를 만드는 데 있어서 BVH보다 시간이 더 오래 걸린다는 것이다.

래스터 랜더링 방식에서는 이런 가속 구조를 사용하지 않고, 삼각형을 랜더러로 바로 보내어 처리한다. 따라서, 래스터 랜더러에는 가속 구조를 생성하기 위한 함수가 존재하지 않는다. 하지만, 레이트레이싱 랜더링 방식에서는 랜더링을 수행하기 전에 가속 구조를 생성해야하는 시점이 존재한다. 이 시점을 API단계에서 자동으로 알 수는 없다. 이 시점은 ***rcFinish()***를 통하여 지시해 주어야만 한다. 이 함수에 의해 처리해야 할 삼각형이 모두 적재되었음을 알려주고, 이 때 가속 구조 생성을 시작한다.

1.3 RayCore® API

레이트레이싱 기법의 영상을 RayCore®로 프로그램하기 위한 방법으로 API(Application Programming Interface)를 지원한다. 이렇게 정의된 인터페이스 함수가 RayCore® API이다. 이 API를 이용해서 3차원 데이터를 렌더러‡로 보내고, RayCore®는 레이트레이싱 방식으로 영상을 생성한다.

RayCore® API는 OpenGL ES 프로그램에 익숙한 사용자가 쉽게 사용할 수 있도록 하기

* Binary Space Partitioning – 공간을 나누었을 때 분할된 공간안에 있는 객체를 구분한다.
† Binary Object Partitioning – 물체를 나누었을 때 분할된 물체로 공간을 구분한다.
‡ 랜더러(Renderer) – 소프트웨어 혹은 하드웨어로 동작하여 영상을 생성하는 방법 혹은 장치를 말한다.

위해서 OpenGL ES 1.1과 유사하게 구현되어 있으며, 함수나 파라미터 또한 유사하게 정의되어 있다. 그러나 렌더링 방식의 차이로 인해 다소 상이한 내용들도 존재한다.

RayCore® API는 OpenGL ES와 비교하여 다음과 같은 주요 특징을 가지고 있다.

- OpenGL ES 1.1 의 함수와 기본 사용법은 유사하다.
- 투영 변환은 사용하지 않는다.
- 객체에 정적/동적 개념을 지정하여 사용할 수 있다.
- 데이터 로드가 완료되면, 랜더링을 시작하는 시점을 명시해야 한다.

텍스쳐*와 물성† 은 각각 객체로 생성하고, 각 객체에 사용할 데이터를 설정하여 사용한다. 이 때 객체 생성시에 부여 받은 이름을 연결(binding)하여 사용할 객체를 지시한다.

RayCore® API에서는 접두사로 RC 혹은 rc를 사용하는데 이는 RayCore®에서 사용됨을 나타내는 것이다. 데이터 타입에는 RC를 사용하며, 함수에는 rc를 사용한다. 모델 데이터를 설정하는 방식은 OpenGL ES 1.1과 동일하다.

OpenGL에서는 시점을 유틸리티 함수를 이용해서 설정할 수 있는데, 이는 모델 전체를 기준 시점에 맞게 변환‡ 한다는 것을 의미한다. 하지만, RayCore®에서는 물체를 그대로 두고 시점을 바꾸면서 영상을 생성하게 된다. 물체를 매번 적재하는 수고를 덜기 위해서 투영 변환을 사용하지 않는다. 이 방식을 통해 정적/동적 객체가 구분되어 사용이 된다. 정적 객체는 프로그램 실행 초기에 한 번만 적재하고, 매번 적재하는 과정없이 재사용할 수 있다. 그러나 동적 객체는 매 프레임마다 가속 구조를 생성하여 적재해 주어야 한다. 이 부분은 OpenGL에서는 사용하지 않는 개념이며, 레이트레이싱 방식을 사용하기 때문에 좀더 효율적인 처리가 가능하게 해 준다.

모델에서 사용할 물성과 텍스쳐를 설정하고, 이를 사용하는 객체를 모아둔다. 모든 데이터가 적재되면 랜더링을 시작하도록 지시해 주어야 한다. 이 지시는 랜더링 시작을 알리는 명령어로 작동한다.

a. 프로그래밍 방식(Programming Model)

RayCore® API를 이용해서 응용프로그램을 작성하기 위해서는 다음과 같은 단계를 기억해야 한다.

- 영상의 전역 데이터 설정
- 랜더링 모델 데이터 설정

---

* Texture – Texture mapping을 이용한 실감 처리 기법에서 사용하는 이미지 데이터 이다.
† 물성(Material Property) – 퐁 쉐이딩에서 사용하는 ambient, diffuse, specular의 기본 요소에 반사 굴절 투과 등의 요소가 포함된 물체의 속성을 의미한다.
‡ 투영 변환

영상의 전역 데이터는 프레임이 바뀌어도 변화되지 않는 데이터 및 설정을 의미한다. 절두체와 배경색 등이 이에 속한다. 경우에 따라 광원의 설정도 변경되지 않을 경우 전역 데이터로 간주하여 설정할 수 있다. 이 설정은 랜더링 데이터 설정에서 사용해도 상관없다. 왜냐하면, 이런 설정으로 인한 처리 속도의 감소가 미미하기 때문이다. 하지만, 꼭 전역 데이터로 설정해야하는 것이 바로 텍스쳐 데이터이다. 텍스쳐의 경우 데이터를 교체해서 사용하는 것은 가능하지만, 교체하는 동안 텍스쳐 데이터의 이동에 많은 시간이 필요하여 성능을 감소시킨다. 이를 없애기 위해서 텍스쳐는 프로그램 실행 초기에 적재한 후 다시 적재하지 않고 재사용한다. 이렇게 하는 이유는 텍스쳐 데이터의 적재 프로세스로 인한 성능 저하가 상당히 발생하기 때문이다.

이와 더불어 사용할 물성 정보를 함께 설정하여 사용하면, 보다 쉬운 물성 및 텍스쳐 사용이 가능해진다. 즉, 텍스쳐와 물성을 따로따로 설정하여 관리하기 보다는 물성에 대한 하나의 속성으로 텍스쳐를 사용하는 것이 보다 효율적이다. 따라서, 텍스쳐 데이터를 적재한 후 물성 설정을 전역 데이터 설정 단계에서 사용하기를 권장한다.

랜더링 모델 데이터(동적 객체 포함)는 프레임이 바뀔 때마다 변경되는 데이터를 의미한다. 모델 데이터는 사용자의 입력 또는 정해진 순서에 따라 움직이기 때문에 매번 변경된다. 이 데이터는 움직임이 없더라도 그 상태로 랜더링해 주어야 하기 때문에, 지속적으로 입력이 이루어져야만 한다. 이 과정은 데이터를 랜더링하는 프로그램이 종료하기 전까지 계속 반복 수행된다. 모델의 좌표 데이터는 랜더링 데이터로 간주된다. 여기에 추가적으로 광원과 시점도 변경될 수 있는데, 이럴 경우 매번 새롭게 관련 정보를 설정해 주어야 한다.

b. Object Naming

특정 데이터를 객체에 설정하고, 각 객체에 이름을 부여하여 사용하는 방식으로 텍스쳐와 물성을 설정한다. 먼저 텍스쳐 객체와 물성 객체를 생성한다. 객체가 올바로 생성되면 이름이 숫자로 지정되어 응용 프로그램에 전달된다. 이 이름으로 객체의 사용을 지시하여 해당 객체의 데이터를 설정한다. 따라서 다음과 같은 순서로 텍스쳐와 물성에 대한 객체를 설정한다.

1. 각 객체를 생성한다.
2. 생성된 객체 중 데이터를 설정할 객체를 생성시 부여 받은 이름으로 지정*한다.
3. 지정된 객체에 데이터를 설정†한다.
4. 객체를 이름으로 지정하면, 그 객체에 설정된 데이터를 사용할 수 있다.

* **지정** – 객체를 선택한다.
† **설정** – 지정된 객체에 데이터를 적재 혹은 설정한다.

1.4 용어

RayCore®를 통해서 사용하는 용어에 대해서 정의한다.

- **모델(Model)** - 영상으로 그리려는 전체 3 차원 데이터.
- **객체(Object)** - 삼각형들의 모음. 객체와 삼각형 또는 객체와 객체의 모음도 객체로 정의함. 객체의 최소 단위는 하나의 물성을 포함하는 삼각형의 집합임. 최소 단위의 객체에는 하나의 물성만을 사용함. 객체가 하나 이상의 물성을 가질 경우 해당 객체를 물성의 수만큼 새로운 객체로 구분하여 상위 객체로 구성함.
- **삼각형(Triangle)** - 세개의 정점으로 구성된 면적이 있는 도형. RayCore®에서 보내주는 데이터 형태는 삼각형임.
- **프리미티브(Primitive)** - RayCore® 내부에서 사용할 가공된 혹은 랜더링이 수행될 원시 도형. 삼각형과 동일하나 RayCore® 내부에서는 프리미티브라고 구분하여 명명함.
- **텍스쳐(Texture)** - 질감을 이미지로 구현해둔 데이터.
- **물성(Material Properties)** - 물체의 속성. 색에 대한 속성과 텍스쳐에 대한 속성 및 반사/굴절/투과에 대한 속성, 즉 빛에 반응하는 속성의 집합.

제 2장 시야

물체를 보이기 위해서는 다양한 정보가 필요하다. 대표적인 정보가 바로 시점(view point) 또는 시야범위(viewing volume)이다. 시점은 물체를 보기 위해 필요한 기본 요소로써 사람이나 카메라처럼 물체를 바라보는 주체(viewer)의 정보를 의미한다. 시야범위는 물체가 보이는 범위를 정량적으로 표시한 값이다. 이들의 3차원 공간상에서의 역할과 설정하는 방식, 그리고 각각에 대한 처리 결과에 대해서 본 장에서 다룬다.

2.1 시점(view point)

시점은 3차원 공간상의 물체를 보기 위해 필요한 시각 정보를 포함한다. 관찰자(viewer)의 위치, 물체를 바라보는 관찰 방향, 물체가 보이는 시야범위로 구성된다. 이런 기본 정보를 통해서 공간상에 있는 물체를 보이게 할 수 있다. 시점을 설정하기 위해서는 다음 함수를 사용한다.

void **rcuLookAt** (RCfloat *eyex*, RCfloat *eyey*, RCfloat *eyez*, RCfloat *centerx*, RCfloat *centery*, RCfloat *centerz*, RCfloat *upx*, RCfloat *upy*, RCfloat *upz*);
시점의 좌표와 바라보는 점 그리고 시점의 위쪽을 알려주는 업(*UP*) 벡터를 지정하는데 사용한다. 시점의 위치는 3 차원 공간상의 좌표를 의미한다. 바라보는 점은 화면 중심을 나타내며, 이를 이용하여 시점의 위치로부터 바라보는 벡터를 계산한다. 업(*UP*) 벡터는 시점의 위쪽이 어디인지 지정하는 값으로 실제 벡터 값을 지정한다.

예를 들어

rcuLookAt(0,0,0, 0,0,-1, 0,1,0);

와 같이 정의를 하면 시점은 (0, 0, 0)의 위치에 있고, z축으로 -1의 점을 바라보고 있다. 업(*UP*) 벡터는 y축으로 양의 방향을 향하며, 크기가 1인 벡터이다. RayCore® API에는 이 값이 기본 값으로 설정되어 있다. 설정을 하지 않고 사용할 경우 위의 값을 그대로 사용한다.

다음 예

rcuLookAt(1,1,1, 0,0,0, -1,1,-1);

는 (1, 1, 1) 위치에서 원점 (0, 0, 0)을 업(*UP*) 벡터가 (-1, 1, -1) 상태로 바라보는 설정이다. 시점이 결정이되면 3차원 공간상에서 물체가 보이는 범위를 설정해야 한다.

2.2 시야범위(Viewing Volume)

시야범위란 화면에 보이고자 하는 물체의 범위를 말한다. 시점을 통해서는 관찰 위치와 관찰 방향이 결정되고, 시야범위에 따라서는 물체가 보여지는 공간이 달라지게 된다. 영어로는 viewing volume 또는 view frustum 이라 정의된다. 시야는 물체가 보여지는 범위와 그렇지 않은 범위로 나누어 진다.

a. 해상도 설정

영상은 시야범위 안에서 발생한 광선을 통하여 생성된다. 화면 해상도는 화면을 구성하는 화소의 개수로 표현되는데, 이 해상도에 의해 생성될 광선의 개수가 결정된다. 화면 해상도는 rcViewport()함수를 통해서 설정한다.

void **rcViewport** (RCint *x*, RCint *y*, RCsizei *width*, RCsizei *height*);
화면 해상도를 설정한다. 이렇게 설정된 해상도에 따라서 결과 영상의 화소 개수가 결정된다. 이를 통해 생성될 1차 광선의 개수가 결정된다. RayCore® API에서는 *x*, *y*의 값은 사용하지 않는다. 결과 영상의 너비와 높이는 *width*와 *height*로 설정한다.

화면 해상도의 기본값은 (800, 480)이다. ***rcViewport()***로 별도 설정을 하지 않으면, 기본 해상도로 영상을 생성한다. 이 값을 통하여 발생될 광선의 개수가 결정되고, 광선의 최초 발생 방향과, 다음 발생시킬 변화량을 구하게 된다. 최초 발생 방향과 변화량은 다음 함수로 시야 범위를 설정하여 최종적으로 지정할 수 있다.

b. 절두체 설정

시야 범위, 즉 절두체(Frustum)는 관찰 방향을 기준으로 물체를 얼마만큼 보여줄 것인지를 나타낸다. 절두체 정보를 설정하는 함수는 ***rcFrustum()***과 ***rcuPerspective()***이 있다.

i. 절두체 직접 설정

rcFrustum()은 공간 정보를 직접 설정하는 함수이다.

void **rcFrustum** (RCdouble *left*, RCdouble *right*, RCdouble *bottom*, RCdouble *top*, RCdouble *zNear*, RCdouble *zFar*);
절단(clipping) 평면의 크기(상하좌우 좌표)와 근거리 및 원거리에 있는 깊이 절단 평면까지의 거리를 설정한다. 상하좌우 좌표는 근거리에 있는 깊이 절단 평면의 크기이다.

원거리 값을 제외한 근거리 값과 상하좌우 좌표는 광선 시작점과 인접한 광선과의 변화량을 계산하는데 사용된다. 이는 일반적으로 최초 시야 범위를 설정할 때 한번만 수행되고, 시야 범위가 변화하지 않을 경우 기존의 값을 그대로 사용한다. 만약 시야 범위가 바뀌게 되면 다시 계산한다.

```
rcFrustum (-1, 1, -1, 1, 1, 1000);
```

위의 코드는 근거리 1에 있는 깊이 절단 평면의 크기를 설정하는 예이다. 해당 평면은 시점에서 바라보는 방향을 중심으로 상하좌우의 거리가 1인 평면이다. 위의 파라미터 중 원거리 값은 레이트레이싱에서 사용되지 않는다.

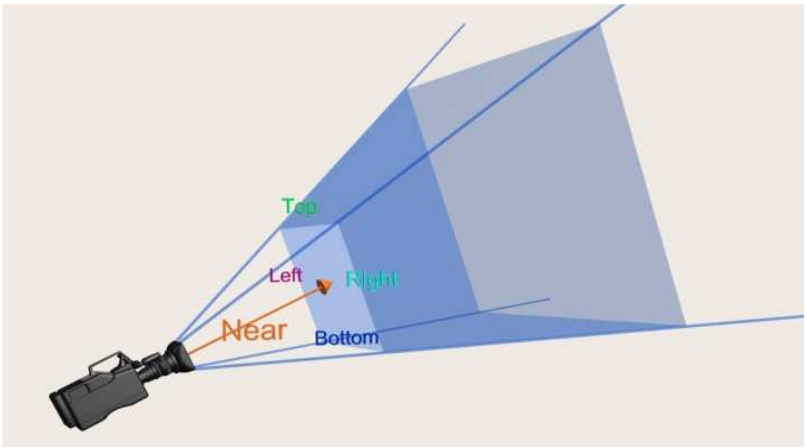

그림 5. 절두체

ii. 절두체 비율 설정

rcFrustum()은 실제 값으로 범위를 설정하기 때문에 비율이 달라져 영상이 왜곡되어 보일 수 있다. 이런 이유로 width와 height의 비율을 유지할 수 있는 ***rcuPerspective()*** 함수가 사용된다.

void **rcuPerspective** (RCfloat *fov*, RCdouble *aspect*, RCfloat *zNear*, RCdouble *zFar*);
절두체에 대한 세로 방향의 시야각(*fov*)을 설정한다. 이 시야각과 종횡비(*aspect*)를 통해서 절두체의 너비를 추출한다. 근거리 *zNear*와 원거리*zFar*는 rcFrustum 함수 파라미터와 동일한 의미이다.

시야각*fov* (field of view)는 물체를 바라보는 범위를 각도로 표현한 것이다. 범위는 (0, 180) 이며, 각도의 기준은 세로 방향이다. 이 시야각을 통해 계산되는 절두체의 높이와 종횡비(*aspect*)를 이용해서 절두체의 너비를 계산할 수 있다. 여기서, 종횡비(*aspect*)는 높이에 대한 너비의 비율을 나타낸다. 이렇게 지정된 절두체는 언제나 화면과 물체간의 비율을 일정하게 유지할 수 있다.

RayCore® API에서는 이 값을 설정할 때 투영 행렬을 생성하지 않는다. 일반적으로 래스터 방식에서는 전체 물체를 투영 행렬을 통해서 정규화된 공간으로 투영하여 처리한다. 이 투영 행렬은 모델 좌표계에서 월드 좌표계로 변환하기 위한 변환 행렬과 함께 사용된다. 하지만 이는 레이트레이싱 기법에서는 사용하지 않는 방식이다.

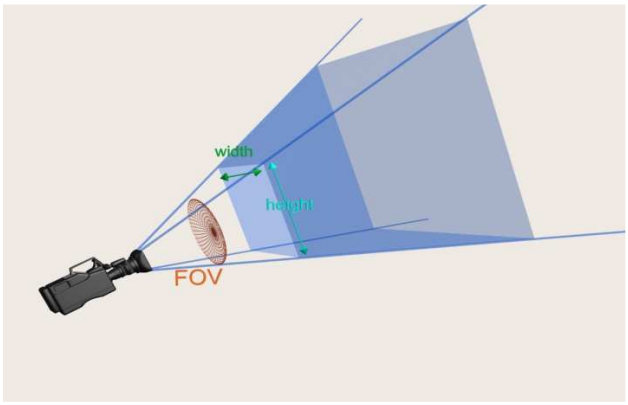

그림 6. 비율을 이용한 절두체 생성

2.3 초기화

레이트레이싱 랜더링에서 초기화해야 하는 가장 중요한 것이 바로 시점 및 절두체 설정이다. 초기화는 앞에서 설명된 함수들을 이용한다.

2.4 불필요한 투영 매트릭스

일반적으로 3D 그래픽 이미지를 생성하기 위해서는 3차원 공간상의 삼각형에 대한 정점 정보가 필요하다. 이 정점 정보를 효과적으로 관리하기 위해서 행렬이 사용된다. 행렬을 사용하여 정점의 좌표를 변환시키면, 물체가 이동된다. 이 좌표 변환 행렬은 3차원 공간에도 동일하게 적용될 수 있다.

물체에 대한 월드 좌표계로의 변환은 래스터 방식과 레이트레이싱 방식에서 동일하게 수행된다. 하지만 투영 변환에는 차이가 있다. 래스터 방식에서는 투영 변환을 처리하기 위한 **PROJECTION_MATRIX** 모드가 존재한다. 이 모드로 설정되는 투영 행렬에 의해 모든 물체들은 정규화된 정육면체로 이동된다. 정육면체 내부에 있는 물체들은 랜더링에 포함되어 나타나지만, 정육면체 외부에 있는 물체들은 보여지지 않기 때문에 랜더링할 필요가 없다. 물체가 보여지는 공간은 시야범위에 의해 결정된다.

레이트레이싱 기법에서는 투영 변환을 수행하지 않는다. 대신 투영 효과는 광선을 생성하는 방식으로 표현된다. 화면의 각 화소에 대한 광선을 발생시켜서 물체를 투영시킨다. 이러한 광선 발생 처리를 통하여 투영 행렬을 사용하는 래스터 방식과 동일한 결과를 나타낼 수 있다.

2.5 예제 코드

a. 비율을 이용한 절두체

기본 설정을 다음과 같이 한다.

- 배경색을 설정한다.
- 랜더링을 수행할 영상의 크기를 설정한다.
- 비율을 이용하여 절두체를 설정한다.
- 시점의 위치와 바라보는 방향을 설정한다.

코드는 다음과 같다.

```
{
  rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
  rcViewport(0, 0, 800, 480);
  rcuPerspective(60.0f, 800.0f / 480.0f, 0.1f, 100.0f);
  rcuLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);
}
```

배경색은 적색을 0.73으로, 녹색을 0.2로, 청색을 0.23으로 설정한다. 화면의 시야각(fov)은 60°로 설정하고, 화면 비율은 스크린의 종횡비로 설정한다. 스크린은 시점으로부터 0.1 떨어진 곳에 있고, 최대 범위는 100을 설정한다.

결과 영상은 다음과 같다. 삼각형을 그리는 방식에 대해서는 다음에 설명한다. 비율을 기반으로 절두체를 설정하였음으로 삼각형의 모양이 왜곡되지 않는다.

그림 7. 비율을 이용한 절두체 설정으로 그려지는 기본 삼각형

b. 값을 이용한 절두체

다음은 값을 기반으로 설정을 하는 경우이다.

- 값을 이용하여 절두체를 설정한다.

코드는 다음과 같다.

```
{
  rcClearColor( 0.73f, 0.2f, 0.23f, 1.0f );
  rcViewport(0, 0, 800, 480);
  rcFrustumf(-0.1, 0.1, -0.1, 0.1, 0.1, 100);
  rcuLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);
}
```

스크린이 시점으로부터 0.1 떨어진 곳에 있다. 좌우 수직 절단 평면의 좌표는 -0.1과 0.1이고, 상하 수평 절단 평명의 좌표는 0.1과 -0.1이다. 설정된 스크린의 높이와 너비에 대한 비율을 이용하면, 비율을 이용한 절두체와 동일한 결과가 나타난다.

다음은 값을 이용해서 절두체를 설정한 결과 영상이다. 영상의 너비와 높이에 대한 비율을 무시하고 영상을 생성하기 때문에 앞서 ***rcPerspective***를 이용했을 경우에 비해 왜곡된 듯한 느낌을 받을 수 있다.

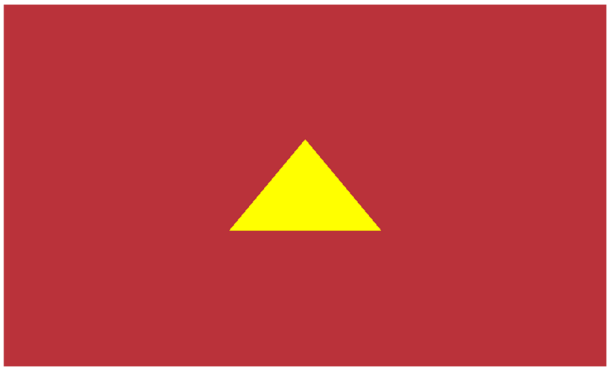

그림 8. 값을 이용한 절두체 설정으로 그려지는 기본 삼각형

c. 시점 위치 변화

다음은 시점을 이동하였을 경우를 살펴본다. 시점은 ***rcuLookAt()*** 을 이용해서 바꿀 수 있다.

- 시점의 위치를 변화시킨다.

코드는 다음과 같다.

```
{
  rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
  rcViewport(0, 0, 800, 480);
  rcuPerspective(60.0f, 800.0f / 480.0f, 0.1f, 100.0f);
  rcuLookAt(0.3, 0, 1, 0, 0, 0, 0, 1, 0);
}
```

시점을 x 축으로 0.3만큼 이동시킨다. 바라보는 위치는 동일하고, 시점의 위치만 바뀐다. 결과는 다음과 같다.

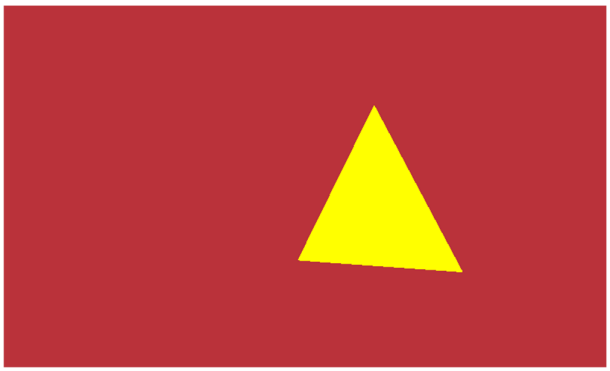

그림 9. 시점의 위치 변화로 그려지는 기본 삼각형

제 3장 변환

물체는 각각의 모델 좌표계에서 구성된다. 이렇게 구성된 물체들은 하나의 월드 좌표계로 변환된다. 이런 변환은 최종적으로 정점의 좌표를 변환하는 방식으로 구현된다. 또한 사용자 입력에 의해 물체가 이동되는 경우에도 변환이 적용된다.

물체의 좌표를 변환하기 위한 함수가 지원된다. 좌표 변환은 3가지로 구분할 수 있다. 이동, 회전, 크기 변환을 통해서 좌표 변환을 수행한다. 각각의 변환은 행렬연산을 통해서 수행된다. 행렬은 아핀(affine) 변환이 가능하도록 4×4행렬로 구성한다.

3.1 모델 변환

좌표 변환을 위해 4×4의 2차원 행렬을 사용한다. 이 행렬은 열기준으로 저장된 16개의 배열로 구성된다.

$$\mathrm{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

a. 이동 변환

이동 변환은 물체를 어느 방향으로 얼마만큼 이동할 지를 나타내는 것이다. 좌표 이동을 수행하기 위한 방향은 축을 선택하여 설정하고, 이동량은 크기값으로 설정한다.

이동 변환을 수행하는 행렬은 다음과 같이 생성된다.

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

이동 변환을 수행하기 위해 다음과 같은 함수가 지원된다.

void **rcTranslatef** (RCfloat *x*, RCfloat *y*, RCfloat *z*);
각 축을 기준으로 설정된 값 (x, y, z) 만큼 이동한다.

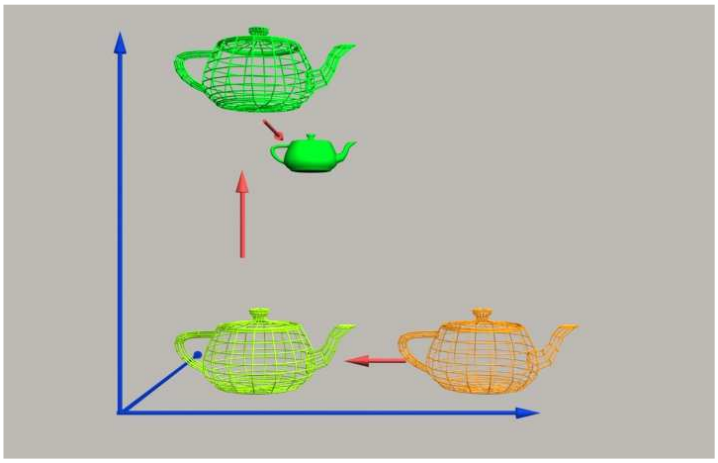

그림 10. 물체의 이동

b. 회전 변환

회전 변환은 물체를 어느 축을 기준으로 얼마만큼 회전할 지를 나타내는 것이다. 회전 축은 임의의 축으로 설정할 수 없다. 좌표계의 기본축인 x, y, z 축 중 하나를 기준으로 회전을 수행한다.

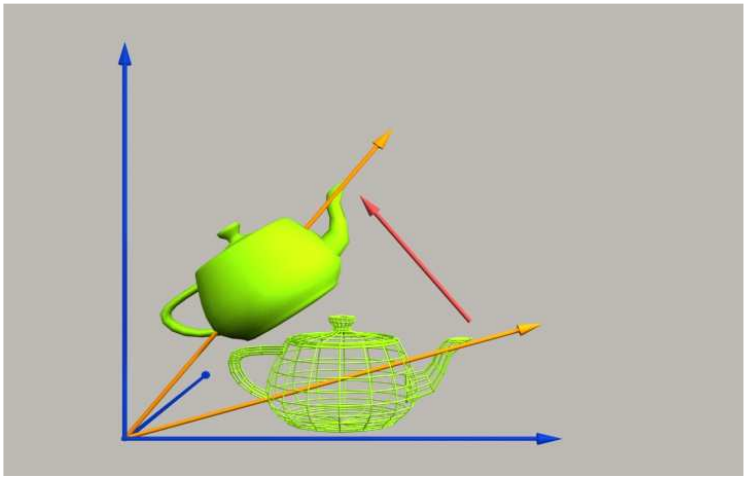

그림 11. 물체의 회전

회전 변환을 수행하는 행렬은 다음과 같이 생성된다. R_x 은 x축 기준의 회전행렬이고, R_y 은 y축 기준의 회전행렬이고, R_z 은 z축 기준의 회전행렬이다.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

회전 변환을 수행하기 위해 다음과 같은 함수가 지원된다.

void **rcRotatef** (RCfloat *angle*, RCfloat *x*, RCfloat *y,* RCfloat *z*);
회전을 수행할 기준축을 *x, y, z* 로 설정한다. 설정된 축을 기반으로 반시계 방향으로 *angle*° 만큼 회전변환을 수행한다.

c. 크기 변환

크기 변환은 물체의 크기를 얼마나 변환할 지를 나타낸다. 크기를 변환 시키는 방식은 원점을 기준으로 해당 좌표에 입력된 값을 곱하는 것이다.

크기 변환을 수행하는 행렬은 다음과 같이 생성된다.

$$s = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

크기 변환을 수행하기 위해 다음과 같은 함수가 지원된다.

void **rcScalef** (RCfloat *x*, RCfloat *y,* RCfloat *z*);
값의 크기에 따라서 커지거나 줄어드는 효과를 볼 수 있다. 입력되는 (*x, y, z*)는 크기를 변화시키고자 하는 배율의 값이다. 좌표계의 원점을 기준으로 변환을 수행한다.

물체가 원점에서 벗어나 있는 경우에는 실제 물체의 크기가 입력된 배율로 변환되지 않고, 원점에서의 거리에 대한 배율로 크기가 변환되므로 이를 염두하여 사용해야 한다.

d. 행렬 변환

프로그램 작성시 모델 행렬을 직접 만들어 사용할 경우 해당 행렬 데이터를 입력하여 모델 변환 행렬을 새롭게 바꿀 수 있다. 또한, 행렬 곱을 이용해서 새로운 행렬을 생성할 수도 있다. 이 행렬은 앞의 세가지 변환 함수를 조합하여 내부에서 생성된다. 생성된 행렬은 기존의 행렬에 지정되거나 기존의 모델 변환 행렬과 곱하여 새로운 모델 변환 행렬로 만들어진다. 이렇게 연산한 모델 행렬은 ***Get***함수를 이용해서 확인할 수 있다.

void **rcLoadMatrixf** (const RCfloat **m*);
행렬 데이터를 적재하는 기능을 수행한다. 입력되는 행렬 데이터 *m은* 열기준으로 저장된 행렬 데이터를 가지는 포인터 값이다. 데이터 크기는 16개로 되어 있어야 한다. 이 데이터는 기존에 모델 변환 행렬의 데이터를 대치한다.
void **rcMultMatrixf** (const RCfloat **m*);
입력되는 행렬 데이터 *m*은 위의 함수와 동일하다. 이 데이터는 기존의 모델 변환 행렬에 저장된 값과 행렬 곱셈을 수행하여 새로운 행렬을 생성한다.

e. 행렬의 푸쉬(Push) / 팝(Pop)

최종 변환 행렬은 여러번의 변환에 의해 생성되는 여러 행렬의 곱으로 이루어진다. 이전 행렬로 되돌리기 위해서는 역행렬을 곱하면 된다. 4x4의 역행렬을 구하고, 다시 곱하는 방식은 매우 비효율적이다. 따라서 되돌리고자 하는 행렬을 메모리에 저장하고 있으

면 바로 가져다 쓸 수 있다. 이는 연산을 다시해야 하는 단점을 피할 수 있다. 메모리에 행렬을 저장하고, 다시 가져와 사용할 수 있도록 행렬 스택(stack)을 사용하고, 이를 ***rcPushMatrix()***와 ***rcPopMatrix()***를 통해서 제어한다.

f. 행렬 스택

계층으로 구성된 모델의 경우 이 방법을 사용하면 변환을 쉽게 적용할 수 있다. 예를 들어 태양계를 생각해 보자. 지구는 태양을 중심으로 공전을 하고, 달은 지구를 중심으로 공전을 한다. 태양이 좌표계의 원점에 있지 않을 경우 태양이 원점이 되도록 지구를 이동시키고, 지구를 원점 기준으로 회전시킨 후 다시 원래의 위치로 이동시킨다. 이때 달은 지구를 중심으로 회전해야 하기 때문에, 지구가 원점이 되도록 달을 이동시켜서 똑같이 원점을 기준으로 회전시킨 후 다시 원래의 위치로 이동시켜야 한다. 여기까지는 아무 문제가 없다. 이후에 화성을 그린다면 어떻게 해야 할 것인지가 문제이다. 두가지 방식이 있을 것이다. 한가지는 태양이의 위치를 바꾸고 화성을 회전시키고, 다시 원래의 위치로 이동하는 행렬을 만든 뒤, 이 행렬에 화성이 공전하는 회전 행렬을 적용하는 방식이다. 다른 방법은 중간의 행렬 데이터를 스택(stack) 메모리에 저장해 두었다가 다시 이용하는 방식이다.

첫번째 방식은 앞서 설명한 함수들을 이용해서 변환 행렬을 다시 만들면 된다. 두번째 방식은 아래에 설명할 함수들을 이용해서 스택(stack) 메모리에 저장하는 시점과 원래의 함수를 되돌리는 시점을 지정하여 사용한다.

void **rcPushMatrix** (void);
현재 사용하고 있는 변환 행렬을 스택 메모리에 저장한다.

이 함수는 현재 설정된 행렬을 기억해 두었다가 이후에 다시 사용한다.

void **rcPopMatrix** (void);
스택 메모리에 저장되어 있는 행렬을 변환 행렬로 가져온다.

이전 행렬로 돌아가서 다시 변환을 시작한다.

3.2 예제 코드

a. 이동

이동을 수행하는 예제이다.

```
#include "RCFramework.h"

// Programming Guide - Translate

struct Pos {
  float x, y, z;
};

RCubyte g_cubeIndices[] = {
  0, 1, 2,
};

struct Pos g_TrianglePos[] = {
  {0.0f, 1.0f, 0.0f},      // 0
  {-1.0f, -1.0f, 0.0f},    // 1
  {1.0f, -1.0f, 0.0f},     // 2
};

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Translate", 800, 480){}
  virtual ~Tutorial(void){}

  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcTranslatef(1.0f, 1.0f, 0.0f);
        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1,1,0,0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }

  void DynamicSceneDraw(void){}

protected:
  virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
              rcClearColor(0.73f, 0.2f, 0.23f, 1.0f );
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();
              rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
              rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

              StaticSceneDraw();
        }

        return TRUE;
  }
```

```
  virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        return TRUE;
  }
};

Tutorial      g_Tutorial;
```

x축과 y축으로 각각 1만큼 이동한다. 중심에 있던 삼각형이 이동된 결과를 확인 할 수 있다.

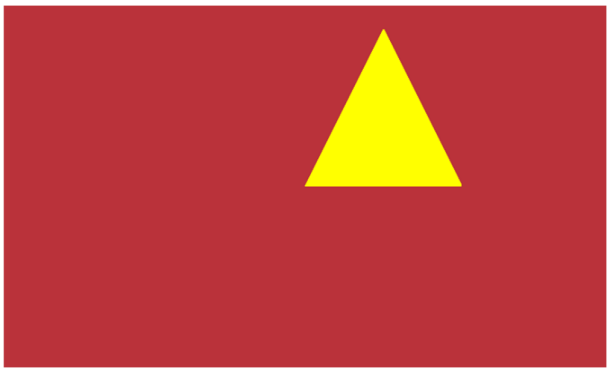

그림 12. 이동을 통해 그려지는 삼각형

b. 회전

회전을 수행하는 예제이다.

```
#include "RCFramework.h"

// Programming Guide - Rotate

struct Pos {
  float x, y, z;
};

RCubyte g_cubeIndices[] = {
  0, 1, 2,
};

struct Pos g_TrianglePos[] = {
  {0.0f, 1.0f, 0.0f},     // 0
  {-1.0f, -1.0f, 0.0f},   // 1
  {1.0f, -1.0f, 0.0f},    // 2
};

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Rotate", 800, 480){}
  virtual ~Tutorial(void){}
```

```
  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcRotatef(45, 0, 0, 1);

        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1, 1, 0, 0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }

  void DynamicSceneDraw(void){}

protected:
  virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
              rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();
              rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(),0.1f, 100.0f);
              rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

              StaticSceneDraw();
        }

        return TRUE;
  }

  virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        return TRUE;
  }
};

Tutorial      g_Tutorial;
```

z축을 기준으로 45°만큼 회전한다. 그 결과는 다음과 같다.

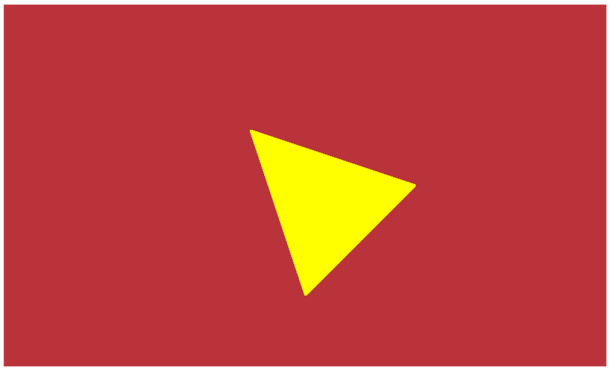

그림 13. 회전을 통해 그려지는 삼각형

c. 크기 변환

크기변환을 수행하는 예제이다.

```
#include "RCFramework.h"

// Programming Guide - Scale

struct Pos {
  float x, y, z;
};

RCubyte g_cubeIndices[] = {
  0, 1, 2,
};

struct Pos g_TrianglePos[] = {
  {0.0f, 1.0f, 0.0f},     // 0
  {-1.0f, -1.0f, 0.0f},   // 1
  {1.0f, -1.0f, 0.0f},    // 2
};

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Scale", 800, 480){}
  virtual ~Tutorial(void){}

  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcScalef(2.0f, 2.0f, 1.0f);

        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1, 1, 0, 0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }
```

```
    void DynamicSceneDraw(void){}

protected:
    virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
            rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
            rcViewport(0, 0, Width(), Height());

            rcMatrixMode(RC_PROJECTION);
            rcLoadIdentity();
            rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
            rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

            StaticSceneDraw();
        }

        return TRUE;
    }

    virtual BOOL OnDraw(void){
        static int i = 0;
        i++; // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        return TRUE;
    }
};

Tutorial    g_Tutorial;
```

크기를 x 축과 y축의 방향으로 2배 확대한다. 확대한 결과는 다음과 같다.

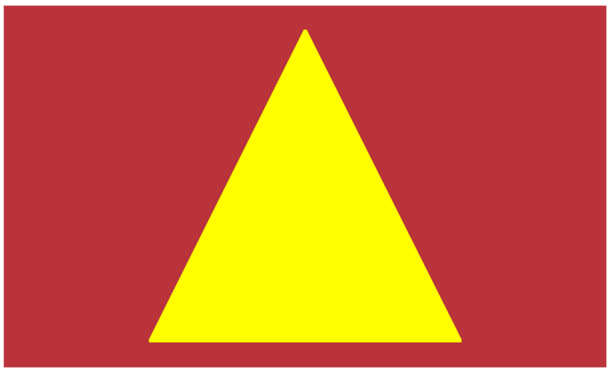

그림 14. 크기 변환을 통해 그려지는 삼각형

1보다 작은 값일 경우는 축소된다. 0의 값을 사용할 경우 정점이 각 축의 원점으로 이동하기 때문에 결과를 볼 수 없다.

d. 회전 후 이동

회전 후 이동과 이동 후 회전의 경우 결과는 달라진다. 다음의 예는 회전 후 이동을 했을 때의 예제이다.

z축을 기준으로 90°회전한 후 x축으로 1만큼 이동한다. 프로그램 코드에서는 이동한 다음에 회전을 한다. 변환 행렬이 오른쪽에서 곱해지기 때문이다.

```
#include "RCFramework.h"

// Programming Guide - Transform(Rotate, Translate)

struct Pos {
  float x, y, z;
};

RCubyte g_cubeIndices[] = {
  0, 1, 2,
};

struct Pos g_TrianglePos[] = {
  {0.0f, 1.0f, 0.0f},     // 0
  {-1.0f, -1.0f, 0.0f},   // 1
  {1.0f, -1.0f, 0.0f},    // 2
};

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Transform(Rotate, Translate)", 800, 480){}
  virtual ~Tutorial(void){}

  void StaticSceneDraw(void){
        rcStaticSceneBegin();

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();


        rcTranslatef(1, 0, 0);
        rcRotatef(90, 0, 0, 1);

        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1, 1, 0, 0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
  }

  void DynamicSceneDraw(void){}

protected:
  virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
              rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();
              rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
              rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

              StaticSceneDraw();
        }

        return TRUE;
  }

  virtual BOOL OnDraw(void){
```

```
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        return TRUE;
    }
};

Tutorial      g_Tutorial;
```

그 결과는 다음과 같다.

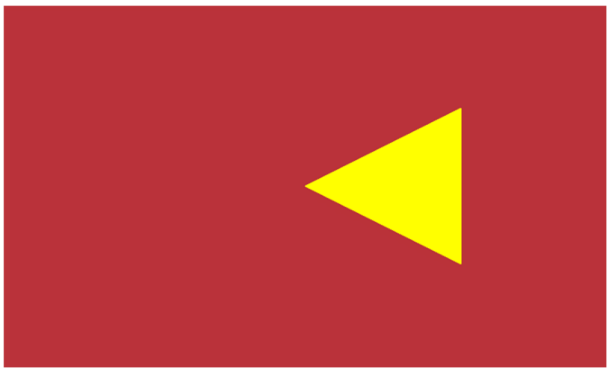

그림 15. 회전 후 이동을 통해 그려지는 삼각형

e. 이동 후 회전

이동 후 회전이 실시된 결과이다. 앞의 예제와는 다른 결과를 얻을 수 있다. 마찬 가지로 프로그램 코드에서는 회전을 먼저 삽입하고, 이동을 수행한다.

```
#include "RCFramework.h"

// Programming Guide - Transform(Translate, Rotate)

struct Pos {
    float x, y, z;
};

RCubyte g_cubeIndices[] = {
    0, 1, 2,
};

struct Pos g_TrianglePos[] = {
    {0.0f, 1.0f, 0.0f},      // 0
    {-1.0f, -1.0f, 0.0f},    // 1
    {1.0f, -1.0f, 0.0f},     // 2
};

class Tutorial : public RCFramework
{
public:
    Tutorial(void) : RCFramework("Programming Guide - Transform(Translate, Rotate)", 800, 480){}
    virtual ~Tutorial(void){}

    void StaticSceneDraw(void){
        rcStaticSceneBegin();
```

```
        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();


        rcRotatef(90, 0, 0, 1);
        rcTranslatef(1, 0, 0);


        rcEnableClientState(RC_VERTEX_ARRAY);

        float color[4]={1, 1, 0, 0};

        rcDisable(RC_TEXTURE_2D);

        rcVertexPointer(3, RC_FLOAT, 0, g_TrianglePos);
        rcBindMaterial(1);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, color);
        rcDrawElements(RC_TRIANGLE_FAN, 3, RC_UNSIGNED_BYTE, g_cubeIndices);

        rcStaticSceneEnd();
    }

    void DynamicSceneDraw(void){}

protected:
    virtual BOOL OnIntialize(void){
        rcSceneAllInit();

        {
            rcClearColor(0.73f, 0.2f, 0.23f, 1.0f );
            rcViewport(0, 0, Width(), Height());

            rcMatrixMode(RC_PROJECTION);
            rcLoadIdentity();
            rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
            rcuLookAt(0, 0, 4, 0, 0, 0, 0, 1, 0);

            StaticSceneDraw();
        }

        return TRUE;
    }

    virtual BOOL OnDraw(void){
        static int i = 0;
        i++; // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        return TRUE;
    }
};

Tutorial    g_Tutorial;
```

먼저 x축으로 1만큼 이동한 후 z축으로 90° 만큼 회전을 하였기 때문에 마치 z축을 기준으로 90° 회전한 후 y축으로 1만큼 이동한 것과 같은 결과를 얻을 수 있다.

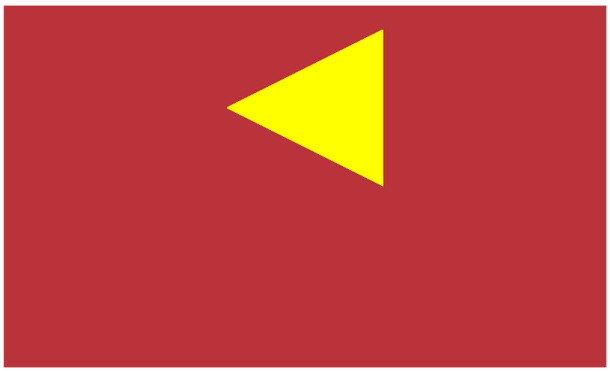

그림 16. 이동 후 회전을 통하여 그려지는 삼각형

두가지 방법이 서로 다른 결과를 보이기 때문에, 원하는 결과를 얻기 위해서는 변환의 순서를 꼭 지켜야 한다.

제 4장 광원

모든 물체는 광원이 존재해야만 그 실체를 확인 할 수 있다. 따라서 영상을 생성하기 위해서는 광원의 설정은 필수적이다. 본 장에서는 광원의 설정과 그에 따른 결과에 대해서 알아본다.

4.1 광원 속성

광원의 속성은 다음과 같다.

- 광원의 위치
- 광원의 형태
- 광원의 주변광 속성
- 광원의 확산광 속성
- 광원의 반사광 속성
- 광원 방향벡터 속성
- 광원 집중광*의 속성

a. 광원의 위치와 종류

해당 광원의 위치를 (x, y, z, w)로 지정한다. (x, y, z)는 3차원 상의 위치를 의미한다. w는 광원의 형태를 나타낸다.

광원은 특정 위치에 고정될 수도 있고, 매 프레임마다 바뀔 수도 있다. 매 프레임마다 변경이 될 경우에는 광원의 위치를 바꿔주면 된다. 입력되는 값을 변경하여 이동시킬 수도 있고, 좌표 변환 함수를 이용해서 바꾸는 것도 가능하다. 광원도 랜더링 객체로 보고 변환 행렬이 적용된다.

* Spot light – 특정 위치만 광원 처리하여 집중도를 높이는 광원이다.

광원은 크게 2가지로 구분이 가능하다. 점 광원과 방향성 광원이다. 점 광원은 태양을 생각할 수 있다. 특정 방향으로 방향성이 있는 광원이 아니라, 점에서 모든 방향으로 발산되는 광원을 의미한다. 광원의 위치에서 발산되는 여러 방향의 벡터로 조명 연산을 수행한다.

그에 비해 방향성 광원은 특정 방향에 대한 방향성만 존재하는 광원을 의미한다. 광원의 위치보다는 방향 벡터가 조명 연산에서 사용된다.

b. 광원 속성

광원에도 조명 연산 처리를 위한 속성을 지정한다. 이는 주변광, 확산광, 반사광의 속성으로 지정되어 있다. 각각의 속성은 빛에서 생기는 물성을 지정한다. 자세한 내용은 물성을 다룬 다음 장에서 다시 언급한다.

c. 집중광

점 광원 중 특이한 경우에 해당한다. 집중광은 점 광원과 같이 위치에서 정해진 방향 중 일부 집중된 곳에서 조명 연산을 처리하는 형태이다.

그림 17. 집중광이 적용된 물체

d. 감쇠(Attenuation)

광원이 멀리 있어 효과를 적용하기 어려운 경우를 표현하기 위해 광원의 감쇠 정도를 지정할 수 있다. 감쇠 요소로 3가지 계수를 설정한다.

- 상수 감쇠 인자(Constant attenuation)
- 선형 감쇠 인자(Linear attenuation)
- 2 차항 감쇠 인자(Quadratic attenuation)

e. 프로그래밍

광원 설정에는 다음과 같은 특징이 있다.

- 조명을 활성화시켜야만 광원을 사용할 수 있다.
- 조명을 활성화시키지 않거나 개별 광원을 설정하지 않는 경우 기본 광원을 사용한다.
- 개별 광원은 8 개까지 지정할 수 있다.
- 개별 광원에 대한 활성화 상태를 설정한다.

광원을 설정하기 위한 함수는 다음과 같다.

void **rcLightv** (RCenum *light*, RCenum *pname*, const RCfloat **params*);
void **rcLight** (RCenum *light*, RCenum *pname*, const RCfloat *param*);
광원을 설정하는 함수이다. light에는 설정하고자 하는 광원을 지정한다. pname을 통해서 설정하고자 하는 속성을 지정한다. 3가지 색의 속성을 지정하기 위해서 RC_POSITION, RC_AMBIENT, RC_DIFFUSE, RC_SPECULAR, RC_SPOT_DIRECTION, RC_SPOT_EXPONENT, RC_INNER_CONE, RC_OUTER_CONE, RC_ATTENUATION_RANGE, RC_START_ATTENUATION, RC_END_ATTENUATION, RC_CONSTANT_ATTENUATION, RC_LINEAR_ATTENUATION, RC_QUADRIC_ATTENUATION를 사용한다. 입력 데이터는 params 혹은 param에 입력한다.

4.2 예제 코드

부록 C의 광원이 지구를 중심으로 돌아가는 예이다. 지구에는 텍스쳐를 사용하였으며, 광원을 회전시켰다.

a. 광원 초기화

광원을 초기화 한다.

```
rcEnable(RC_LIGHTING);

i = 0;

while(i < sceneData.m_lightCount) {

      RCenum lightNumber = RC_LIGHT0 + i;

      if(i < sceneData.m_lightCount) {
            rcEnable(lightNumber);

      sceneData.m_light[i].pos.w = 1;
      sceneData.m_light[i].pos.x -= 15;

      rcLightfv(lightNumber, RC_POSITION, &sceneData.m_light[i].pos.x);
      rcLightfv(lightNumber, RC_AMBIENT, &sceneData.m_light[i].ambient.r);
      rcLightfv(lightNumber, RC_DIFFUSE, &sceneData.m_light[i].diffuse.r);
      rcLightfv(lightNumber, RC_SPECULAR, &sceneData.m_light[i].specular.r);
      i++;
}
```

조명 사용 유무를 설정한 후 개별 광원을 활성화시킨다. 그 다음으로 각 광원의 물성을 설정한다.

b. 광원 회전

변환 행렬을 이용해서 광원을 회전시키는 방식이다.

```
rcMatrixMode(RC_MODELVIEW);
rcLoadIdentity();

rcPushMatrix();
rcRotatef(g_turn, 0, 1, 0);
rcLightfv(RC_LIGHT0, RC_POSITION, &sceneData.m_light[i].pos.x);
rcPopMatrix();
```

변환 행렬이 설정되면, 광원의 위치값과 변환 행렬이 곱해져 광원의 새로운 위치을 얻을 수 있다.

제 5장 텍스쳐

텍스쳐 매핑은 보다 사실적인 영상을 효과적으로 생성할 수 있는 방식으로 3차원 그래픽에서 필수적으로 사용되고 있다. 이를 사용하기 위해 지원하는 함수와 사용 방법을 살펴 본다.

5.1 텍스쳐 매핑

폴리곤만으로 물체의 모델링을 극사실적으로 표현하는 것은 거의 불가능하다. 필요한 폴리곤의 수가 컴퓨터로 처리할 수 있는 능력 이상이 되기 때문이다. 더욱이 실시간 처리를 위해서는 데이터의 크기를 줄여야만 사용이 가능한데, 각각의 형태를 기하 형태로 모두 구성하는 대신 일부를 영상으로 덧씌움으로써 그 효과를 충분히 나타낼 수 있다. 또한, 물체의 질감을 표현할 수 있어 더욱 사실감을 부여할 수 있다. 현재 3차원 그래픽에서의 텍스쳐 매핑은 필수적인 요소이다.

텍스쳐를 사용하기 위해서는 정점에 대한 텍스쳐 좌표와 함께 그와 관련된 특정 이미지를 지정하여 사용한다. 이미지는 질감을 표현할 수도 있고, 특정 형태를 대신하여 표현할 수도 있다.

텍스쳐 이미지의 경우 사용하는 개수를 한정하고, 텍스쳐 사용을 위해서는 객체이름(ID)으로 객체를 지정하여 사용하도록 한다. 텍스쳐 데이터를 매번 적재하여 사용하는 경우 실시간 랜더링 성능에 매우 큰 영향을 미치기 때문에 3차원 데이터 초기화 단계에서 적재한다.

a. 텍스쳐 데이터

텍스쳐 데이터는 2차원 이미지로 구성된다. 텍스쳐 이미지 데이터를 이용해서 삼각형 혹은 특정 물체의 재질을 나타낼 수 있다. 이런 재질은 기하 데이터, 즉 삼각형으로는

구성하기가 거의 불가능하다. 실시간 랜더링을 위해 데이터를 줄여야 하는 이유뿐만 아니라, 재질표현에 있어서 텍스쳐는 필수적이다.

그림 18. 재질을 표현하는 텍스쳐 이미지

위의 재질 표현은 모델링의 간략화를 이룰 수 있다. 사진이나 사실적인 이미지를 사용하여 영상을 보다 실감나게 표현할 수 있다.

b. 텍스쳐 좌표

텍스쳐를 사용하기 위해서는 정점당 텍스쳐 좌표를 설정해야 한다. 이 좌표는 모델링 단계에서 지정한다. 지정된 좌표는 정점 데이터와 매우 유사한 방식을 이용해서 적재한다. 관련된 함수는 7장(객체 그리기)에서 설명할 것이다.

다음은 텍스쳐를 간단히 설정하는 방법이다. 기본적으로 사각형의 물체에 기본 이미지를 덧씌우는 방식이다. 예를들어 각 정점을 다음과 같이 설정할 수 있다. 텍스쳐 좌표 설정에 따라 랜더링 결과가 달라진다.

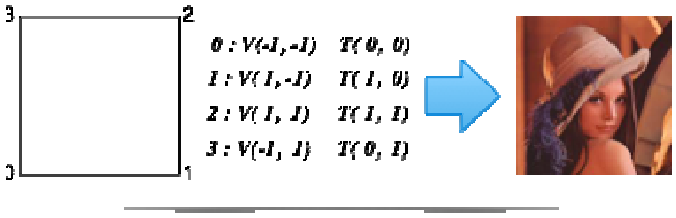

그림 19. 사각형에 적용한 텍스쳐 매핑

5.2 텍스쳐 필터링

광선이 발생되어 특정 삼각형과 만나면 그 만나는 점에 해당하는 텍스쳐 화소를 덧씌운다. 이런 과정을 텍스쳐 매핑이라고 한다. 이 과정에서 하나의 단위 화소를 텍셀*이라고 한다. 삼각형의 깊이(혹은 거리)에 따라서 이 텍셀을 여러 개 사용해야 하는 경우가 발생한다. 또는 그 반대로 하나의 텍셀이 여러 개의 화소에 적용되는 경우도 있다. 이 때 단순히 하나의 텍셀만을 사용하면 생성되는 이미지에 계단 현상†이 발생할 수 있다. 이런 현상은 생성되는 이미지를 부자연스럽게 만든다. 이런 문제를 해결하기 위해서 사용하는 것이 텍스쳐 필터링이다.

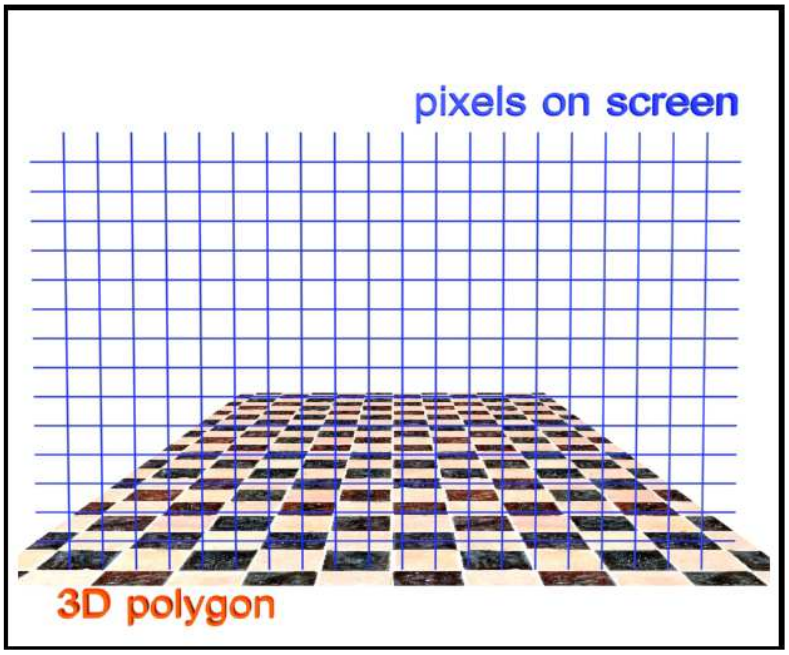

그림 20. 픽셀당 여러 개의 텍셀이 적용된 경우

텍스쳐 필터링을 이용하면 보다 부드러운 결과 이미지를 얻을 수 있다. 다음 그림은 필터링을 수행했을 때와 그렇지 않은 경우를 비교한 것이다. 필터링을 수행하면 전체적으로 매핑 결과가 보다 부드러우면서 자연스러워진다. 텍스쳐 밉맵(mipmap)은 대표적인 필터링 기법으로 RayCore® API에서는 기본적으로 지원된다. 현재는 밉맵 방식으로 공일차내삽법(bilinear) 방식을 지원하지만 추후에 다양한 기법과 설정을 지원할 예정이다.

* **Texel** – Texture Pixel의 줄임말.
† Aliasing effects

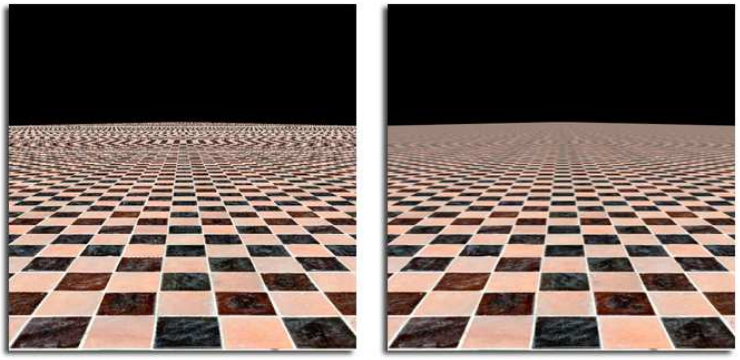

그림 21. 필터링을 적용하지 않은 이미지(좌)와 적용한 이미지(우)

a. 밉맵 텍스쳐

텍스쳐를 덧씌우는 삼각형의 크기는 매번 변화될 수 있다. 이에 텍스쳐 매핑의 결과가 매우 비사실적으로 나타날 수도 있다. 이는 작은 프리미티브(primitive)에 큰 텍스쳐를 사용했을 경우 각 화소간의 연결성이 떨어지기 때문이다. 이를 해결하기 위해서 필터링을 수행하는데 가장 대표적인 방식이 밉맵 텍스쳐를 사용하는 방식이다.

밉맵 텍스쳐란 텍스쳐 이미지를 1/4의 크기로 줄여 새로운 상위 순위로 구성하고, 이를 이미지 데이터로 미리 저장한 후 사용하는 방식이다. 텍스쳐를 매번 줄여서 생성하거나 연결성 해소를 위해서 많은 텍셀을 가져오게 되는 경우 그 처리 속도가 느려져 매우 비효율적이다. 이런 비효율성을 개선하기 위하여 상위 순위 이미지를 만들어 저장한 뒤 사용하여 처리 속도를 높일 수 있다. 물론 텍스쳐를 밉맵화하는 작업에 상당한 시간이 소요되지만, 이런 처리는 랜더링이 시작되기 전의 전처리 단계에서 수행되기 때문에 랜더링 속도에는 영향이 없다.

그림 22. 텍스쳐 밉맵

텍스쳐 필터링 역시 전처리로 수행하게 된다. 비록 정확한 영상을 생성하지는 못하지만 처리 성능에서는 많은 효율을 얻을 수 있다. 화질 면에서도 정확한 값이 사용되지는 않는다. 하지만 좋은 성능을 보이는 방식이라는 것에는 이견이 없다. 따라서 이 방식의 사용은 선택이 아닌 필수라고 할 수 있다.

5.3 텍스쳐 객체

텍스쳐는 객체 단위로 처리한다. 텍스쳐 객체를 생성하고, 각 객체의 이름을 받아 온다. 이렇게 받아 온 이름은, 색인 목록으로 필요에 따라서 텍스쳐로 묶어서 매핑을 수행한다.

void **rcGenTextures** (RCsizei *n*, RCint **textures*);
텍스쳐 객체를 생성한다. 생성하려는 객체의 개수를 *n*으로 지정한다. 객체가 생성되면 각 텍스쳐 객체의 이름이 *textures*에 배열로 반환된다. 여기에서 반환된 객체 이름은 삭제하기 전까지 계속 사용할 수 있다. 객체 이름은 1 에서부터 순차적으로 생성되지만, 연속적인 값은 아니다. 객체 이름을 저장하기 위한 *textures*에 이름을 저장하기 위한 공간은 이 함수를 호출하기 전에 미리 할당을 해 두어야 한다.

텍스쳐 객체의 이름은 RayCore® API 내부와 그 상위 응용프로그램에서 사용하는 텍스쳐 객체를 동일하게 유지할 수 있도록 가상으로 지정하는 값이다. 이 이름은 ***rcBindTexture()*** 를 통해서 적재 또는 사용할 텍스쳐 객체를 나타낸다.

void **rcBindTexture** (RCenum *target*, RCint *texture*);
RayCore® API에서는 2차원 텍스쳐만을 지원하기 때문에 *target*에는 RC_TEXTURE_2D만 지원한다. 사용하려는 텍스쳐의 객체 이름을 *texture*를 통해서 지정한다. 텍스쳐 객체를 사용하기 위해서는 그 이름을 반드시 설정해야 한다.

텍스쳐 데이터를 적재하기 위한 함수는 ***rcTexImage2D()***이다. 2차원 텍스쳐만 지원한다. 이 함수를 이용하여 텍스쳐 데이터를 RayCore®에 적재하여 사용한다. 밉맵 텍스쳐를 자동으로 생성한다. 내부적으로 공일차내삽법(bilinear) 필터링을 사용하기 때문에 메모리 접근 패턴이 미리 결정이 되어 있다. 즉 텍스쳐 좌표 (u, v)가 가르키는 텍셀은 근접한 4개의 텍셀로부터 가져오는 방식이다.

텍스쳐 메모리는 매우 높은 지역성을 가지고 있다. 따라서 텍스쳐 매핑 유닛에서의 캐쉬(cache) 메모리를 사용하여 메모리 대역폭을 줄일 수 있다. 여기에 텍스쳐 필터링에서

처리하는 방식에 맞게 텍스쳐 데이터를 재배열하면 보다 좋은 성능을 얻을 수 있다. 처리 성능을 높이기 위해서 전처리 단계에서 텍스쳐 데이터를 재배열한다. 이는 특정 함수로 지정하지 않고, 데이터를 적재하는 기본 함수내에서 처리한다.

void **rcTexImage2D** (RCenum *target*, RCint *level*, RCint *internalformat*, RCsizei *width*, RCsizei *height*, RCint *border*, RCenum *format*, RCenum *type*, const RCvoid **pixels*);
텍스쳐 이미지를 적재하기 위한 함수로 사용한다. 텍스쳐는 2 차원만을 지원하기 때문에 *target* 은 RC_TEXTURE_2D 만 지원한다. 이 함수에는 텍스쳐 밉맵을 생성하는 기능이 추가되어 있다. 따라서 밉맵 순위를 0 으로만 입력받아 자동으로 상위 순위를 생성한다. OpenGL 에서 사용하는 함수일 경우에는 각 순위를 매번 지정했지만, 이 함수는 0 만을 사용한다. 텍스쳐의 크기인 *width* 와 *height* 를 지정한다. 텍스쳐의 크기는 2 의 승수로 구성되어야 한다. 크기는 최소 16 이고, 최대 1024 이다. 이 범위와 요건을 만족시키지 못하면 해당 텍스쳐를 적재하지 않고 오류를 발생시킨다. 입력되는 텍스쳐 데이터는 RGB 또는 RGBA 를 지원한다. 내부 텍스쳐는 항상 RGBA 의 4 개의 요소로 구성되어 있다. 따라서 *format* 은 입력 데이터의 요소를 지정하는 RC_RGB 와 RC_RGBA 를 지원한다. *internalformat* 은 입력받는 데이터의 요소인 *format* 과 동일한 값을 사용한다. 각 요소의 데이터 형을 *type* 에서 지정한다. 지원하는 데이터 형은 RC_BYTE, RC_UNSIGNED_BYTE, RC_SHORT, RC_UNSIGNED_SHORT 이다. 실제 데이터의 주소는 *pixels* 를 통해서 전달한다.

RayCore® 랜더러에서 지원하는 텍스쳐 매핑 유닛은 RGBA의 4개의 요소로 구성되어 사용한다. 그런데 RGB로 입력되는 데이터가 있을 경우 알파값을 임의로 설정할 수 있도록 하기 위해 다음의 함수를 사용한다. 이를 설정하지 않을 경우 알파값은 기본 불투명 속성값(16)으로 지정된다. (0은 투명의 속성이다.)

void **rcTextureAlpha** (unsigned char *value*);
텍스쳐가 3개의 요소로 구성되어 있을 경우 설정되는 *value*값을 알파값에 자동으로 적용한다. 텍스쳐가 4개의 요소로 구성되어 있을 경우에는 원본 텍스쳐의 값을 설정한다.

5.4 프로그래밍

텍스쳐를 사용하기 위해서는 초기 설정단계에서 사용할 모든 텍스쳐 데이터를 적재하고, 텍스쳐 사용 단계에서는 적재된 텍스쳐를 연결하여 사용한다. 이렇게 사용하기 위해서 정해진 규약에 따라야 텍스쳐를 올바로 사용할 수 있다.

[초기 설정 단계]
1. 텍스쳐 객체를 생성한다. 생성한 텍스쳐 객체의 이름을 받아온다.
2. 텍스쳐 이름으로 적재하려는 텍스쳐를 활성화시킨다.
3. 텍스쳐를 적재한다.

[물성 설정시 텍스쳐 사용을 명시]
1. 사용하려는 텍스쳐를 해당 이름으로 활성화시킨다.
2. 텍스쳐에 대한 활성화 상태를 명시한다.

텍스쳐는 물성의 일부로 보고 물성 설정 단계에서 함께 지정하여야 한다. 랜더링 단계에서도 변경이 가능하지만, 물성과 함께 묶어서 사용함을 염두에 두고 프로그램을 작성해야 의도한 결과를 얻을 수 있다. 따라서 텍스쳐 및 물성 객체의 설정은 초기에 수행하고 랜더링에서는 변경하지 않도록 하는 것을 권장한다.

5.5 예제 코드

a. 텍스쳐 적재

텍스쳐를 적재하는 방법은 다음과 같다.

1. 텍스쳐 객체를 생성한다.
2. 사용하려는 텍스쳐 객체를 이름으로 지정한다.
3. 텍스쳐 데이터를 적재한다.

```
rcGenTextures(1, &m_texture[index].bindName);
rcBindTexture(RC_TEXTURE_2D, m_texture[index].bindName);


if(m_texture[index].pImage->type == 3){

      rcTextureAlpha(0x10);
      rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                        m_texture[index].pImage->sizeX,
                        m_texture[index].pImage->sizeY,
                        0, RC_RGB, RC_UNSIGNED_BYTE,
                        m_texture[index].pImage->data);
}
else if(m_texture[index].pImage->type == 4) {

      rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGBA,
                        m_texture[index].pImage->sizeX,
                        m_texture[index].pImage->sizeY,
                        0, RC_RGBA, RC_UNSIGNED_BYTE,
                        m_texture[index].pImage->data);
}
```

내부적으로 텍스쳐는 RGBA의 형태를 사용한다. 텍스쳐가 RGB로 되어 있을 경우 알파값(A)은 ***rcTextureAlpha*** 함수를 이용해서 지정하여 사용한다. 원래의 텍스쳐 데이터가 RGBA의 형태일 경우 그냥 사용하면 된다.

b. 텍스쳐 사용 명시

텍스쳐 사용 여부의 명시는 물성 설정에서 한다. 물성 설정에 대해서는 다음 장에서 설명한다.

```
{
  rcGenMaterials(1, &sceneData.m_material[i].bindName);
  rcBindMaterial(sceneData.m_material[i].bindName);

  rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, &sceneData.m_material[i].ambient.r);
  rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, &sceneData.m_material[i].diffuse.r);
  rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, &sceneData.m_material[i].specular.r);
  rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, sceneData.m_material[i].exponent);
  rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, &sceneData.m_material[i].reflection.r);
  rcMaterialfv(RC_FRONT_AND_BACK, RC_TRANSMITTANCE,
&sceneData.m_material[i].refraction.r);
  rcMaterialf( RC_FRONT_AND_BACK, RC_REFRACTION_INDEX,
sceneData.m_material[i].refractionIndex);
  rcEnable(RC_TEXTURE_2D);
  rcBindTexture(RC_TEXTURE_2D, sceneData.m_texture[0].bindName);
}
```

물성 설정 단계에서 텍스쳐를 활성화시키고, 적재한 텍스쳐 객체를 이름을 이용해서 지정한다.

제 6장 물성

물성은 랜더링에서 사용하는 데이터로 물체에 지정되는 특성값이다. 이 값은 적색, 녹색, 청색의 RGB 3개의 요소로 구분된다. 이 장에서는 이 요소에 대한 각각의 의미와 설정 방식에 대해서 알아본다.

6.1 물체의 속성(Material Property)

물체의 속성에는 색을 결정하기 위한 속성과 2차 광선을 생성하기 위한 속성으로 구분이 된다. 여기에 질감을 나타내는 텍스쳐를 지정하여 색의 요소와 더불어 사용한다. 텍스쳐는 색의 요소에 적용되어 사용된다. 반사, 굴절, 투과는 2차 광선을 어떻게 발생시킬지 결정하는 요소이다. 2차 광선에 필요한 이런 속성도 3개의 요소로 나뉘는데 이는 RGB의 반사, 투과의 정도를 지정하는 값으로 이해할 수 있다.

6.2 색 속성

색 속성은 랜더링시 조명 연산을 수행하기 위한 요소들이다. 주변광, 확산광, 반사광의 3가지로 구분하여 사용한다.

a. 주변광 (Ambient)

주변에 존재하는 빛으로 방향성이 없고, 일정하게 빛나는 속성을 의미한다. 광원으로부터 시작된 광선은 물체와 무수히 많은 반사 및 굴절이 이루어진 뒤 방향성을 잃어버리게 된다.

낮에 햇빛이 직접 집안으로 들어오지 않아도 환하게 사물을 볼 수 있는 것은 바로 햇빛에서 파생되어 반사 또는 굴절된 광선이 주변에 산재하여 사물을 볼 수 있기 때문이다.

동일한 환경에서 밤에 주위를 분간할 수 없는 이유는 바로 주변광이 사라지기 때문이다. 이런 속성을 색의 속성으로 표현한 것이 바로 주변광이다.

b. 확산광 (Diffuse)

확산광은 3차원 그래픽에서 가장 일반적으로 생각할 수 있는 광선의 속성이다. 광원의 방향성에 의한 물체로의 입사각에 따라 최종 결과색에 영향을 미치는 광선의 속성이다. 광원의 위치에 따라서 밝고 어둠이 구별된다.

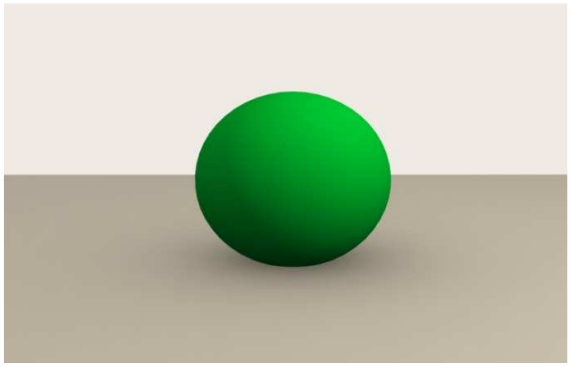

그림 23. 확산광을 적용한 구체

c. 반사광 (Specular)

물체에 빛을 집중해서 비추면 일부분이 강하게 반응하여 환하게 보여진다. 이런 것을 표현하는 것이 반사광이다. 이 속성으로 인해 빛이 특정 위치에 있을 때 물체의 일부분이 환하게 표현된다.

물체의 일부가 다른 곳보다 더욱 밝게 표현되는 것은 해당 위치의 법선 벡터값에 따라서 상대적으로 밝은 값을 얻기 때문이다. 광원의 위치에 따른 하이라이트 부분을 표현할 수 있다.

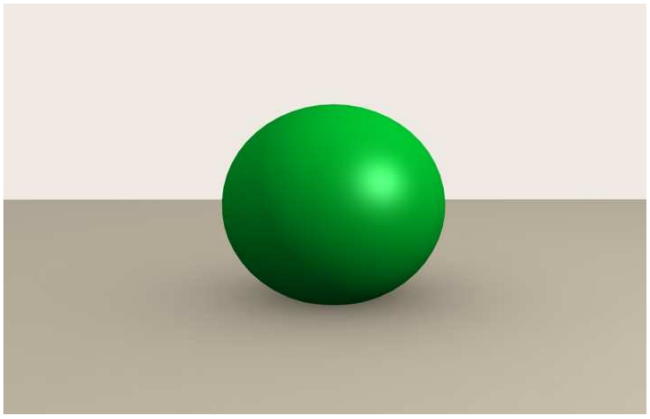

그림 24. 반사광을 적용한 구체

6.3 빛의 속성

빛의 속성은 2차 광선을 생성하는 데 사용된다. 빛의 속성을 설정할 때는 다음을 주의해야 한다.

- 반사율과 투과율의 총합이 1 을 넘지 않아야 한다.

a. 반사 (Reflection)

물체에 반사의 속성을 설정한다. 반사가 적용된 물체는 주변의 물체가 반사되어 표현된다.

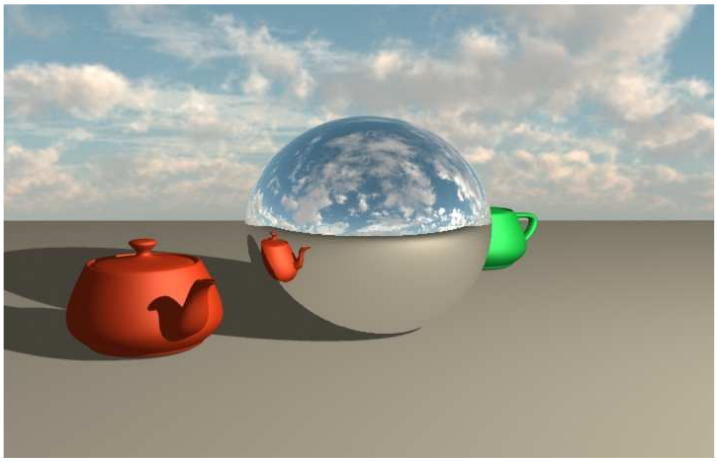

그림 25. 반사 속성을 적용한 구체

b. 투과 (Transmission)

광선이 투과되는 비율을 설정한다. 이 값을 통해서 광선이 얼마나 투과되는지를 지정할 수 있다. 이 속성 또한 RGB의 형태로 값을 지정한다.

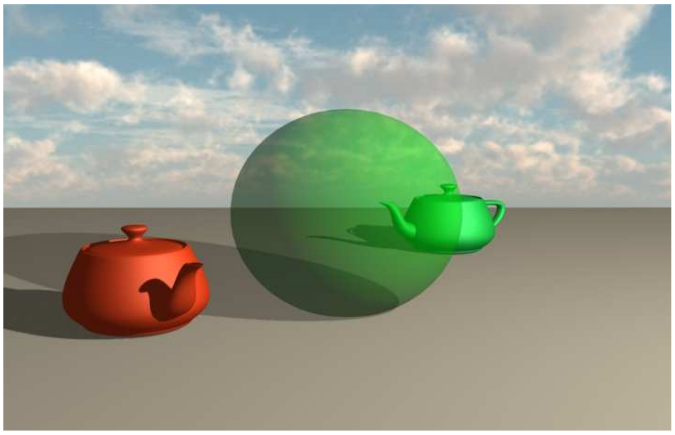

그림 26. 투과 속성을 적용한 구체

c. 굴절 (Refraction)

광선이 굴절되는 정도를 하나의 수치인 굴절률로 표현한다. 굴절률에 따라서 투과되는 빛의 방향이 달라진다. 그림 27은 그림 26과 비교하여 그림자 및 주전자의 표현이 달라짐을 확인할 수 있다.

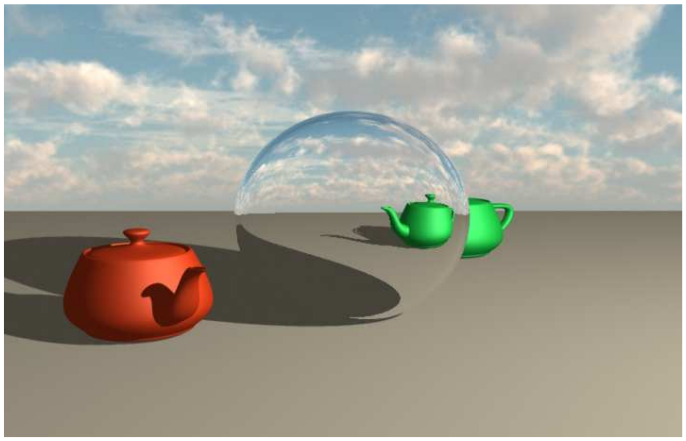

그림 27. 굴절 속성을 적용한 구체

6.4 프로그래밍

물성 객체는 512개로 제한된다. 0번 물성은 사용하지 않음을 지정하기 위해서 비워두기 때문에, 실제로 사용자가 설정할 수 있는 개수는 511개이다. 이 한도에서 자유롭게 사용할 수 있다.

OpenGL에서는 이런 개수의 제한은 없지만, 매번 속성값을 설정해야 하는 번거로움이 있었다. RayCore® API에서는 속성값을 한번만 설정한 후, 추가적인 속성값 지정없이 필요에 따라 물성 개체만 지정하면 물성이 적용될 수 있도록 함으로써 프로그램을 보다 명확히 하고, 반복 사용하는 물성에 대해서 재사용이 가능하도록 하였다.

다음의 함수를 이용해서 물성을 설정한다.

void **rcGenMaterials** (RCsizei *n*, RCuint **materials*);
물성 객체를 생성한다. 생성하려는 객체의 개수를 *n*에 지정한다. 객체가 생성되면 그 객체의 이름을 *materials*에서 받아온다. 최소 얻어지는 이름은 1부터이다. 0은 물성을 초기화 할 경우에 사용할 수 있도록 비워둔다.
void **rcBindMaterial** (RCuint *materials*);
생성될 물성 객체를 지정한다. 지정된 물성 객체가 활성화된다. 물성 객체를 설정하기 위해서는 해당 객체의 활성화가 꼭 필요하다. 활성화가 이루어진 물성 객체는 속성 설정 혹은 랜더링 객체에서 사용할 물성 객체임을 의미한다.
void **rcMaterialf** (RCenum *face*, RCenum *pname*, RCfloat *param*); void **rcMaterialfv** (RCenum *face*, RCenum *pname*, const RCfloat **params*);
물성을 설정하기 위한 함수로 *rcDraw*함수 단위로 값을 설정한다. *face*는 RC_FRONT_AND_BACK만 지원한다. *pname*을 통해서 설정하고자 하는 속성을 지정한다. 3가지 색의 속성을 지정하기 위해서 RC_AMBIENT, RC_DIFFUSE, RC_SPECULAR, RC_REFLECTION, RC_TRANSMITTANCE, RC_AMBIENT_AND_DIFFUSE를 사용한다. 입력 데이터는 *params*에 벡터 형태로 입력된다. RC_SHINIESS, RC_REFRACTION_INDEX를 통해서는 단일값을 입력하기 위해서 사용한다. 입력 값은 *param*에 지정된다.

- RC_AMBIENT 는 주변광에 대한 3 가지 색 정보를 입력한다.
- RC_DIFFUSE 는 확산광에 대한 3 가지 색 정보를 입력한다.
- RC_SPECULAR 는 반사광에 대한 3 가지 색 정보를 입력한다.
- RC_REFLECTION 은 반사량에 대한 3 가지 색의 속성 정보를 입력한다.
- RC_TRANSMITTANCE 은 투과에 대한 3 가지 색의 속성 정보를 입력한다.
- RC_AMBIENT_AND_DIFFUSE 는 주변광과 확산광에 색 정보를 동시에 입력한다.
- RC_SHININESS 는 반사광의 지수값을 설정한다.
- RC_REFRACTION_INDEX 는 굴절률을 설정한다.

a. 프로그램 방식

물성을 설정하고 사용하는 방식은 다음과 같다.

[초기 설정 단계]
1. 물성 객체를 생성한다. 생성한 물성 객체의 이름을 받아온다.
2. 물성 객체의 이름으로 적재하려는 물성 객체를 활성화시킨다.
3. 물성 데이터를 설정한다.

[랜더링 단계]
1. 사용하려는 물성 객체를 이름으로 활성화시킨다.
2. 랜더링 객체를 적재한다.

b. 주의점

물성을 설정하고 사용하는데 있어서 주의해야 하는 점은 다음과 같다.

- 물성 설정 데이터는 전역 데이터임을 기억한다.
- 텍스쳐도 물성 설정의 일부임을 기억한다.

물성 객체는 전체 랜더링 수행 단계에서 사용하는 전역 데이터이다. OpenGL에서 사용하듯이 빈번하게 변경할 경우, 동일한 물성을 사용하던 다른 랜더링 객체도 변경된 물성을 이용하여 랜더링하게 된다. 하나의 물성을 재사용할 수 있음으로 인해 이런 경우가 발생하는 것이다. 물성 데이터를 바꾸면서 사용하려면 해당 물성 객체를 다른 랜더링 객체에서 재사용되지 않도록 독립적으로 물성 객체를 지정해서 사용해야 한다.

텍스쳐도 물성 데이터 설정에서 활성화시킨다. 랜더링에서는 텍스쳐를 설정하지 않는 것이 프로그램 작성시 혼란을 줄일 수 있다. 물성 객체의 지정만으로 텍스쳐도 함께 사용하거나 사용하지 않을 수 있기 때문에, 프로그램을 간략화할 수 있다.

6.5 예제 코드

a. Phong shading 물성 설정

기본 물성을 지정한다. OpenGL에서 사용하는 속성을 그대로 사용한다. 설정한 값에 따라서 결과는 다음과 같이 달라진다.

```
#include "RCFramework.h"

// Programming Guide - Phong Shading

#include "sphere.h"

class Tutorial : public RCFramework
{
public:
```

```
  Tutorial(void) : RCFramework("Programming Guide - Phong Shading", 800, 480){}
  virtual ~Tutorial(void){}

  RCuint    m_staticMaterial;
  RCuint    m_bindName[4];

protected:
  virtual BOOL OnIntialize(void){

        rcSceneAllInit();

        {
              rcDepthBounce(14);

              rcClearColor(0.5f, 0.5f, 0.5f, 1.0f);
              rcViewport(0, 0, Width(), Height());

              rcMatrixMode(RC_PROJECTION);
              rcLoadIdentity();

              rcuPerspective(30, (float)Width() /(float)Height(), 10, 10000);

              rcuLookAt(0, 0, 8, 0, 0, 0, 0, 1, 0);


                 /*
                  * Light Setting
                  *
                  */

              rcEnable(RC_LIGHTING);

              {
                    rcEnable(RC_LIGHT0);

                    {
                          float pos[4] = {-5, 5, 8, 1};
                          rcLightfv(RC_LIGHT0, RC_POSITION, pos);
                    }
                    {
                          float ambient[4] = {0, 0, 0, 0};
                          rcLightfv(RC_LIGHT0, RC_AMBIENT, ambient);
                    }
                    {
                          float diffuse[4] = {0.2f, 0.3f, 0.4f, 0};
                          rcLightfv(RC_LIGHT0, RC_DIFFUSE, diffuse);
                    }
                    {
                          float specular[4] = {1, 1, 1, 0};
                          rcLightfv(RC_LIGHT0, RC_SPECULAR, specular);
                    }
              }


               /*
                  * Material Setting
                  *
                  */

              {
                    rcGenMaterials(1, &m_bindName[0]);

                    rcBindMaterial(m_bindName[0]);

                    {
                          float ambient[4] = {0, 0, 0, 0};
                          rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
                    }
                    {
                          float diffuse[4] = {0, 1, 1, 0};
                          rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
                    }
                    {
                          float specular[4] = {1, 0, 0, 0};
                          rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
                    }
```

```
        {
            float exponent = 10;
            rcMaterialf (RC_FRONT_AND_BACK, RC_SHININESS, exponent);
        }
        rcDisable(RC_TEXTURE_2D);
    }

    {
        rcGenMaterials(1, &m_bindName[1]);

        rcBindMaterial(m_bindName[1]);

        {
            float ambient[4] = {0, 0, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
        }
        {
            float diffuse[4] = {1, 0, 1, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
        }
        {
            float specular[4] = {0, 1, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
        }
        {
            float exponent = 20;
            rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, exponent);
        }

        rcDisable(RC_TEXTURE_2D);
    }

    {
        rcGenMaterials(1, &m_bindName[2]);

        rcBindMaterial(m_bindName[2]);

        {
            float ambient[4] = {0, 0, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
        }
        {
            float diffuse[4] = {1, 1, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
        }
        {
            float specular[4] = {0, 0, 1, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
        }
        {
            float exponent = 30;
            rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, exponent);
        }

        rcDisable(RC_TEXTURE_2D);
    }

    {
        rcGenMaterials(1, &m_bindName[3]);

        rcBindMaterial(m_bindName[3]);
        {
            float ambient[4] = {0, 0, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
        }
        {
            float diffuse[4] = {0, 1, 1, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
        }
        {
            float specular[4] = {0, 0, 0, 0};
            rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
        }
        {
            float exponent = 20;
            rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, exponent);
```

```
            }

            rcDisable(RC_TEXTURE_2D);
        }


        /*
         * Static Scene
         *
         */

        {

            float galaxyV[12] = {
                -1, 0, -1,
                1, 0, -1,
                1, 0, 1,
                -1, 0, 1
            };

            float galaxyT[8] = {
                0, 0,
                1, 0,
                1, 1,
                0, 1
            };

            float diffuse[3] = {0.6f, 0.8f, 0.5f};

            rcGenMaterials(1, &m_staticMaterial);
            rcBindMaterial(m_staticMaterial);

            {
                float ambient[4] = {0, 0, 0, 0};
                rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
            }
            {
                float diffuse[4] = {1, 1, 0, 0};
                rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);
            }
            {
                float specular[4] = {0, 0, 0, 0};
                rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, specular);
            }
            {
                float exponent = 10;
                rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, exponent);
            }

        rcDisable(RC_TEXTURE_2D);

        rcStaticSceneBegin();

        rcEnableClientState(RC_VERTEX_ARRAY);
        rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

        rcVertexPointer(3, RC_FLOAT, 0, galaxyV);
        rcTexCoordPointer(2, RC_FLOAT, 0, galaxyT);

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();
        rcTranslatef(0.0, -1, -10);
        rcScalef(8, 1, 10);

        rcDrawArrays(RC_QUADS, 0, 4);

        rcDisableClientState(RC_VERTEX_ARRAY);
        rcDisableClientState(RC_TEXTURE_COORD_ARRAY);

        rcStaticSceneEnd();
            }
    }

    return TRUE;
}
```

```
virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        {
                rcClear(RC_COLOR_BUFFER_BIT | RC_DEPTH_BUFFER_BIT);

                rcMatrixMode(RC_MODELVIEW);
                rcLoadIdentity();
                rcTranslatef(0.0, 0, -7.0);
                rcEnableClientState(RC_VERTEX_ARRAY);
                rcEnableClientState(RC_NORMAL_ARRAY);

                rcVertexPointer(3, RC_FLOAT, sizeof(struct Pos), g_SphereVertices);
                rcNormalPointer(RC_FLOAT, 0, g_SphereNormals);

                rcPushMatrix();
                rcBindMaterial(m_bindName[0]);
                rcTranslatef(2, 0, 0);
                rcDrawArrays(RC_TRIANGLES, 0, sizeof(g_SphereVertices)/sizeof(Pos));
                rcPopMatrix();

                rcPushMatrix();
                rcBindMaterial(m_bindName[1]);
                rcTranslatef(-2, 0, 0);
                rcDrawArrays(RC_TRIANGLES, 0, sizeof(g_SphereVertices)/sizeof(Pos));
                rcPopMatrix();

                rcBindMaterial(m_bindName[2]);
                rcDrawArrays(RC_TRIANGLES, 0, sizeof(g_SphereVertices)/sizeof(Pos));

                rcDisableClientState(RC_VERTEX_ARRAY);
                rcDisableClientState(RC_NORMAL_ARRAY);
        }

        return TRUE;
  }
};

Tutorial      g_Tutorial;
```

물성의 값을 각 객체당 달리 설정하였으며, 그 결과에 따라서 다음과 같이 다른 색을 얻을 수 있음을 확인할 수 있다.

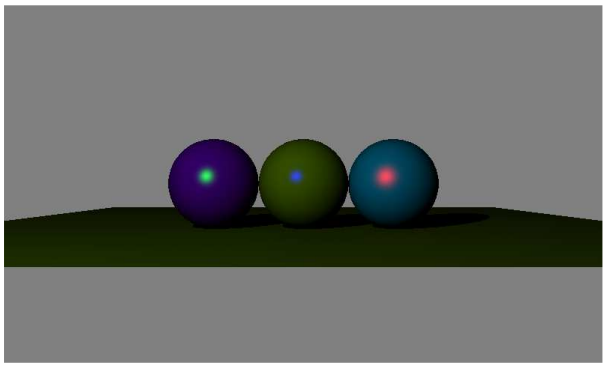

그림 28. 물체의 속성값이 서로 다른 구체들

b. 반사

거울과 같은 반사 예제이다. 두 개의 사각형 중 큰 사각형은 프레임이고, 작은 사각형이 거울이다. 따라서 작은 사각형에는 물성의 반사값을 설정한다. 그 앞에 있는 삼각형이 반사된 결과를 얻을 수 있다.

```
#include "RCFramework.h"

// Programming Guide - Reflection

typedef struct RGBImageRec {
  int sizeX, sizeY;
  unsigned char *data;
} RGBImageRec;

typedef struct bmpBITMAPFILEHEADER{
  WORD    bfType;
  DWORD  bfSize;
  WORD    bfReserved1;
  WORD    bfReserved2;
  DWORD  bfOffBits;
} BITMAPFILEHEADER1;

typedef struct bmpBITMAPINFOHEADER{
  DWORD  biSize;
  DWORD  biWidth;
  DWORD  biHeight;
  WORD    biPlanes;
  WORD    biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  DWORD  biXPelsPerMeter;
  DWORD  biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPFILEHEADER2;

RGBImageRec *DIBImageLoad(char* path, int channel) {
  RGBImageRec* pImage=NULL;
  FILE *f=NULL;
  unsigned char *pBuf=NULL;
  int dataSize=0;
  int index=0;
  DWORD x=0;
  DWORD y=0;
  int bpp=0;

  if(channel !=3 && channel !=4) return pImage;

  f = fopen(path, "rb");
  if(f != NULL) {
        BITMAPFILEHEADER1 HD1;
        BITMAPFILEHEADER2 HD2;

        fseek(f, 0, SEEK_SET);
        fread(&HD1.bfType, sizeof(WORD), 1, f);
        fread(&HD1.bfSize, sizeof(WORD), 1, f);
        fseek(f, 10, SEEK_SET);
        fread(&HD1.bfOffBits, sizeof(int),1, f);

        fread(&HD2.biSize, sizeof(int), 1, f);
        fread(&HD2.biWidth, sizeof(int), 1, f);
        fread(&HD2.biHeight, sizeof(int), 1, f);
        fread(&HD2.biPlanes, sizeof(WORD), 1, f);
        fread(&HD2.biBitCount, sizeof(WORD), 1, f);
        fread(&HD2.biCompression, sizeof(int), 1, f);

        fseek(f,HD1.bfOffBits,SEEK_SET);

        bpp = HD2.biBitCount/8;
        if(bpp == 1 || bpp == channel)
        {
              pBuf = (unsigned char*) malloc(channel);
```

```
            pImage = (RGBImageRec*) malloc(sizeof(RGBImageRec));
            pImage->sizeX = HD2.biWidth;
            pImage->sizeY = HD2.biHeight;

            dataSize = HD2.biWidth*HD2.biHeight*channel;
            pImage->data = (unsigned char*) malloc(dataSize);

            for(y=0; y<HD2.biHeight; y++) {
                for(x=0; x<HD2.biWidth; x++) {
                    fread(pBuf, bpp, 1, f);
                    if(bpp == 1) {
                        pBuf[1] = pBuf[2] = pBuf[0];
                        if(channel == 4)    pBuf[3] = 0;
                    }

                    index = (y*HD2.biWidth + x)*channel;
                    pImage->data[index]     = pBuf[2];
                    pImage->data[index + 1]  = pBuf[1];
                    pImage->data[index + 2]  = pBuf[0];
                    if(channel == 4)
                        pImage->data[index + 3] = pBuf[3];
                }
            }

            if(pBuf) free(pBuf);
            pBuf = NULL;
        }

        fclose(f);
    }

    return pImage;
}

RCfloat         m_Angle=0.0f;
RGBImageRec     *g_texture;
RCuint          *g_textureName;

#define MATERIAL_FRAME          0
#define MATERIAL_MIRROR         1
#define MATERIAL_TRIANGLE       2
unsigned int g_MaterialArray[3];

class Tutorial : public RCFramework
{
public:
    Tutorial(void) : RCFramework("Programming Guide - Reflection", 800, 480){}
    virtual ~Tutorial(void){}

    void GenTexture(void){
        g_texture = (RGBImageRec *)DIBImageLoad("./scenedata/Guide_Reflection/triangle.bmp", 3);

        if(g_texture){
            g_textureName = new RCuint[1];

            rcGenTextures(1, g_textureName);
            rcBindTexture(RC_TEXTURE_2D, g_textureName[0]);
            rcTextureAlpha(128);
            rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                        g_texture[0].sizeX, g_texture[0].sizeY, 0,
                        RC_RGB, RC_UNSIGNED_BYTE,
                        g_texture[0].data);

            if(g_texture->data)   free(g_texture->data);
            free(g_texture);
        }
    }

    void GenMaterial(void){
        RCfloat Reflectance[] = {1.0, 1.0, 1.0};
        RCfloat NotReflectance[] = {0, 0, 0};
        RCfloat color[3][4]={
            {0.2f, 0.4f, 0.8f, 0.0f},
            {0.0f, 0.8f, 0.4f, 0.0f},
            {0.0f, 1.0f, 0.0f, 0.0f},
        };
```

```
        rcGenMaterials(3, g_MaterialArray);

        rcBindMaterial(g_MaterialArray[MATERIAL_FRAME]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE,
&color[MATERIAL_FRAME][0]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, NotReflectance);

        rcBindMaterial(g_MaterialArray[MATERIAL_MIRROR]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE,
&color[MATERIAL_MIRROR][0]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, Reflectance);

        rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK,
                        RC_AMBIENT_AND_DIFFUSE,
                        &color[MATERIAL_TRIANGLE][0]);

        RCfloat ambient[] = {0.8,0.8,0.8,0};
        RCfloat diffuse[] = {0.2,0.2,0,0};
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);

        rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, NotReflectance);
    }

    void RenderMirror(void){
        RCfloat QuadVertices[4][3];

        QuadVertices[0][0]=2.0f;
        QuadVertices[0][1]=-2.0f;
        QuadVertices[0][2]=0.0f;

        QuadVertices[1][0]=2.0f;
        QuadVertices[1][1]=2.0f;
        QuadVertices[1][2]=0.0f;

        QuadVertices[2][0]=-2.0f;
        QuadVertices[2][1]=2.0f;
        QuadVertices[2][2]=0.0f;

        QuadVertices[3][0]=-2.0f;
        QuadVertices[3][1]=-2.0f;
        QuadVertices[3][2]=0.0f;

        rcDisable(RC_TEXTURE_2D);

        rcEnableClientState(RC_VERTEX_ARRAY);

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcVertexPointer(3, RC_FLOAT, 0, QuadVertices);

        rcPushMatrix();
        rcTranslatef(0, 0, -10);
        rcScalef(2, 2, 2);
        rcBindMaterial(g_MaterialArray[MATERIAL_FRAME]);
        rcDrawArrays(RC_QUADS,0,4);
        rcPopMatrix();

        //mirror
        rcPushMatrix();
        rcTranslatef(0, 0, -9.999f);
        rcScalef(1.7f, 1.7f, 1.7f);
        rcBindMaterial(g_MaterialArray[MATERIAL_MIRROR]);
        rcDrawArrays(RC_QUADS, 0, 4);
        rcPopMatrix();

        rcDisableClientState(RC_VERTEX_ARRAY);
    }

    void RenderTriangle(void){
```

```
        RCfloat TriangleVertices[3][3];

        TriangleVertices[0][0] = 0.0f;
        TriangleVertices[0][1] = 0.707f;
        TriangleVertices[0][2] = 0.0f;

        TriangleVertices[1][0] = -1.0f;
        TriangleVertices[1][1] = -0.707f;
        TriangleVertices[1][2] = 0.0f;

        TriangleVertices[2][0] = 1.0f;
        TriangleVertices[2][1] = -0.707f;
        TriangleVertices[2][2] = 0.0f;

        rcEnableClientState(RC_VERTEX_ARRAY);

        rcVertexPointer(3, RC_FLOAT, 0, TriangleVertices);

        rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

        float TexIndex[3][2];
        TexIndex[0][0] = 0.5f;     TexIndex[0][1] = 0.f;
        TexIndex[1][0] = 0.f;      TexIndex[1][1] = 1.f;
        TexIndex[2][0] = 1.f;      TexIndex[2][1] = 1.f;

        rcTexCoordPointer(2, RC_FLOAT, 0, TexIndex);

        rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
        rcEnable(RC_TEXTURE_2D);
        rcBindTexture(RC_TEXTURE_2D, g_textureName[0]);

        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        rcPushMatrix();
        rcTranslatef(0, 0, -5);
        rcRotatef(m_Angle, 0, 1, 0);
        rcDrawArrays(RC_TRIANGLES, 0, 3);
        rcPopMatrix();

        rcDisableClientState(RC_TEXTURE_COORD_ARRAY);
        rcBindMaterial(0);
        rcBindTexture(RC_TEXTURE_2D, 0);
        rcDisable(RC_TEXTURE_2D);

        rcDisableClientState(RC_VERTEX_ARRAY);
    }

    void StaticSceneDraw(void){
        rcStaticSceneBegin();
        RenderMirror();
        rcStaticSceneEnd();
    }

    void DynamicSceneDraw(void){
        RenderTriangle();
    }

protected:
    virtual BOOL OnIntialize(void){
        GenTexture();
        GenMaterial();

        rcSceneAllInit();

        {
            rcClearColor(0.2f, 0.2f, 0.2f, 1.0f);
            rcViewport(0, 0, Width(), Height());

            rcMatrixMode(RC_PROJECTION);
            rcLoadIdentity();

            rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 1.0f, 500.0f);
            rcuLookAt(-3, 0, 0, 0, 0, -8, 0, 1, 0);

            StaticSceneDraw();
```

```
        }
        return TRUE;
    }

    virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        m_Angle += 10.f;

        return TRUE;
    }
};

Tutorial      g_Tutorial;
```

삼각형이 움직이면서 사각형 거울에 반사되어 보인다.

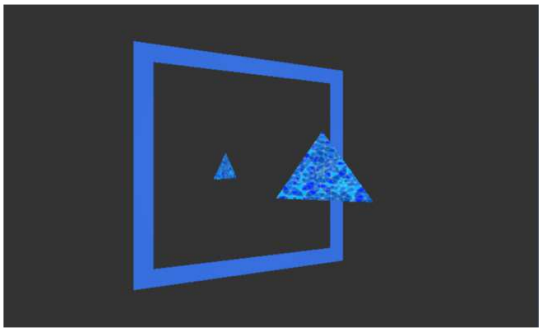

그림 29. 반사 효과를 적용한 결과

c. 굴절

삼각형이 있고, 그 앞에 있는 사각형에는 물성의 투과율과 굴절률을 설정한다. 삼각형이 투과 및 굴절된 결과를 얻을 수 있다.

```
#include "RCFramework.h"

// Programming Guide - Refraction

typedef struct RGBImageRec {
    int sizeX, sizeY;
    unsigned char *data;
} RGBImageRec;

typedef struct bmpBITMAPFILEHEADER{
    WORD    bfType;
    DWORD  bfSize;
    WORD    bfReserved1;
    WORD    bfReserved2;
    DWORD  bfOffBits;
} BITMAPFILEHEADER1;

typedef struct bmpBITMAPINFOHEADER{
    DWORD  biSize;
    DWORD  biWidth;
    DWORD  biHeight;
```

```
  WORD    biPlanes;
  WORD    biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  DWORD  biXPelsPerMeter;
  DWORD  biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPFILEHEADER2;

RGBImageRec *DIBImageLoad(char* path, int channel) {
  RGBImageRec* pImage=NULL;
  FILE *f=NULL;
  unsigned char *pBuf=NULL;
  int dataSize=0;
  int index=0;
  DWORD x=0;
  DWORD y=0;
  int bpp=0;

  if(channel !=3 && channel !=4) return pImage;

  f = fopen(path, "rb");
  if(f != NULL) {
        BITMAPFILEHEADER1 HD1;
        BITMAPFILEHEADER2 HD2;

        fseek(f, 0, SEEK_SET);
        fread(&HD1.bfType, sizeof(WORD), 1, f);
        fread(&HD1.bfSize, sizeof(WORD), 1, f);
        fseek(f, 10, SEEK_SET);
        fread(&HD1.bfOffBits, sizeof(int),1, f);

        fread(&HD2.biSize, sizeof(int), 1, f);
        fread(&HD2.biWidth, sizeof(int), 1, f);
        fread(&HD2.biHeight, sizeof(int), 1, f);
        fread(&HD2.biPlanes, sizeof(WORD), 1, f);
        fread(&HD2.biBitCount, sizeof(WORD), 1, f);
        fread(&HD2.biCompression, sizeof(int), 1, f);

        fseek(f,HD1.bfOffBits, SEEK_SET);

        bpp = HD2.biBitCount/8;
        if(bpp == 1 || bpp == channel)
        {
              pBuf = (unsigned char*) malloc(channel);

              pImage = (RGBImageRec*) malloc(sizeof(RGBImageRec));
              pImage->sizeX = HD2.biWidth;
              pImage->sizeY = HD2.biHeight;

              dataSize = HD2.biWidth*HD2.biHeight*channel;
              pImage->data = (unsigned char*) malloc(dataSize);

              for(y=0; y<HD2.biHeight; y++) {
                    for(x=0; x<HD2.biWidth; x++) {
                          fread(pBuf, bpp, 1, f);
                          if(bpp == 1) {
                                pBuf[1] = pBuf[2] = pBuf[0];
                                if(channel == 4)      pBuf[3] = 0;
                          }

                          index = (y*HD2.biWidth + x)*channel;
                          pImage->data[index]       = pBuf[2];
                          pImage->data[index + 1]  = pBuf[1];
                          pImage->data[index + 2]  = pBuf[0];
                          if(channel == 4)
                                pImage->data[index + 3] = pBuf[3];
                    }
              }

              if(pBuf) free(pBuf);
              pBuf = NULL;
        }

        fclose(f);
```

```
  }

  return pImage;
}

float g_refractionIndex1 = 1.0f;
float g_refractionIndex2 = 1.0f;

RGBImageRec    *g_texture;
RCuint         *g_textureName;

#define MATERIAL_TRIANGLE          0
#define MATERIAL_REFRACTION1       1
#define MATERIAL_REFRACTION2       2
unsigned int g_MaterialArray[3];

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Refraction", 800, 480){}
  virtual ~Tutorial(void){}

  void GenTexture(void){
        g_texture = (RGBImageRec *)DIBImageLoad("./scenedata/Guide_Refraction/texture0.bmp", 3);

        if(g_texture){
              g_textureName = new RCuint[1];

              rcGenTextures(1, g_textureName);
              rcBindTexture(RC_TEXTURE_2D, g_textureName[0]);
              rcTextureAlpha(128);
              rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                           g_texture[0].sizeX, g_texture[0].sizeY, 0,
                           RC_RGB, RC_UNSIGNED_BYTE,
                           g_texture[0].data);

              if(g_texture->data)   free(g_texture->data);
              free(g_texture);
        }
  }

  void GenMaterial(void){
        float transmittance[3] = {0.5, 0.5, 0.5};
        RCfloat color[3][4]={
              {0.8f, 0.3f, 0.4f, 0.0f},
              {0.2f, 0.4f, 0.8f, 0.0f},
              {0.4f, 0.8f, 0.2f, 0.0f},
        };

        rcGenMaterials(3, g_MaterialArray);

        rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK,
                     RC_AMBIENT_AND_DIFFUSE,
                     &color[MATERIAL_TRIANGLE][0]);

        rcBindMaterial(g_MaterialArray[MATERIAL_REFRACTION1]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK,
                     RC_AMBIENT_AND_DIFFUSE,
                     &color[MATERIAL_REFRACTION1][0]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_TRANSMITTANCE, transmittance);
        rcMaterialf(RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, g_refractionIndex1);

        rcBindMaterial(g_MaterialArray[MATERIAL_REFRACTION2]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK,
                     RC_AMBIENT_AND_DIFFUSE,
                     &color[MATERIAL_REFRACTION2][0]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_TRANSMITTANCE, transmittance);
        rcMaterialf(RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, g_refractionIndex2);
  }

  void RenderTriangle(void){
        RCfloat TriangleVertices[3][3];
```

```
	TriangleVertices[0][0] = 0.0f;
	TriangleVertices[0][1] = 0.707f;
	TriangleVertices[0][2] = 0.0f;

	TriangleVertices[1][0] = -1.0f;
	TriangleVertices[1][1] = -0.707f;
	TriangleVertices[1][2] = 0.0f;

	TriangleVertices[2][0] = 1.0f;
	TriangleVertices[2][1] = -0.707f;
	TriangleVertices[2][2] = 0.0f;

	rcEnableClientState(RC_VERTEX_ARRAY);

	rcVertexPointer(3, RC_FLOAT, 0, TriangleVertices);

	rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

	float TexIndex[3][2];
	TexIndex[0][0] = 0.5f;	TexIndex[0][1] = 0.f;
	TexIndex[1][0] = 0.f;	TexIndex[1][1] = 1.f;
	TexIndex[2][0] = 1.f;	TexIndex[2][1] = 1.f;
	rcTexCoordPointer(2, RC_FLOAT, 0, TexIndex);

	rcMatrixMode(RC_MODELVIEW);
	rcLoadIdentity();

	rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
	rcEnable(RC_TEXTURE_2D);
	rcBindTexture(RC_TEXTURE_2D, g_textureName[0]);

	rcPushMatrix();
	rcTranslatef(0, 0, -10);
	rcScalef(3, 3, 3);
	rcDrawArrays(RC_TRIANGLES, 0, 3);
	rcPopMatrix();

	rcDisableClientState(RC_TEXTURE_COORD_ARRAY);
	rcBindMaterial(0);
	rcBindTexture(RC_TEXTURE_2D, 0);
	rcDisable(RC_TEXTURE_2D);

	rcDisableClientState(RC_VERTEX_ARRAY);
}

void RenderRefraction(void){
	RCfloat QuadVertices[4][3];

	QuadVertices[0][0] = 2.0f;
	QuadVertices[0][1] = -2.0f;
	QuadVertices[0][2] = 0.0f;

	QuadVertices[1][0] = 2.0f;
	QuadVertices[1][1] = 2.0f;
	QuadVertices[1][2] = 0.0f;

	QuadVertices[2][0] = -2.0f;
	QuadVertices[2][1] = 2.0f;
	QuadVertices[2][2] = 0.0f;

	QuadVertices[3][0] = -2.0f;
	QuadVertices[3][1] = -2.0f;
	QuadVertices[3][2] = 0.0f;

	rcEnableClientState(RC_VERTEX_ARRAY);

	rcVertexPointer(3, RC_FLOAT, 0, QuadVertices);

	rcMatrixMode(RC_MODELVIEW);
	rcLoadIdentity();

	rcPushMatrix();
	rcTranslatef(0, 0, -5);

	rcBindMaterial(g_MaterialArray[MATERIAL_REFRACTION1]);
```

```
        rcMaterialf(RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, g_refractionIndex1);
        rcDrawArrays(RC_QUADS, 0, 4);

        rcPopMatrix();

        rcDisableClientState(RC_VERTEX_ARRAY);
    }

    void RefreshRefraction(){
        rcBindMaterial(g_MaterialArray[MATERIAL_REFRACTION1]);
        rcMaterialf(RC_FRONT_AND_BACK, RC_REFRACTION_INDEX, g_refractionIndex1);
    }

    void StaticSceneDraw(void){
        rcStaticSceneBegin();
        RenderTriangle();
        RenderRefraction();
        rcStaticSceneEnd();
    }

    void DynamicSceneDraw(void){
        RefreshRefraction();
    }

protected:
    virtual BOOL OnIntialize(void){
        GenTexture();
        GenMaterial();

        rcSceneAllInit();

        {
            rcClearColor(0.4f, 0.3f, 0.2f, 0.0f);
            rcViewport(0, 0, Width(), Height());

            rcMatrixMode(RC_PROJECTION);
            rcLoadIdentity();

            rcuPerspective(70.0f, (RCfloat)Width() / (RCfloat)Height(), 1.0f, 500.0f);
            rcuLookAt(-3, 0, 0, 0, 0, -8, 0, 1, 0);

            StaticSceneDraw();
        }
        return TRUE;
    }

    virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        DynamicSceneDraw();

        g_refractionIndex1 += 0.1f;
        g_refractionIndex2 -= 0.1f;

        return TRUE;
    }
};

Tutorial    g_Tutorial;
```

굴절률이 바뀌면서 투과 및 굴절되는 결과가 달라진다.

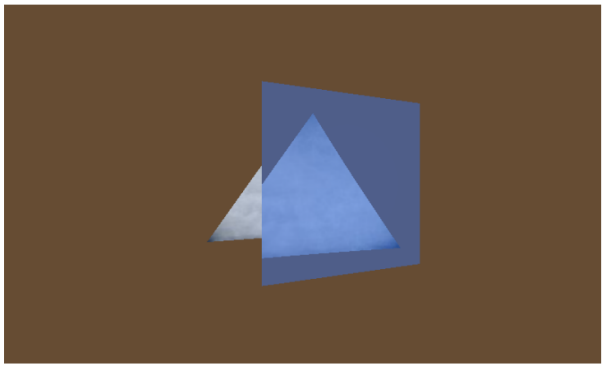

그림 30. 투과 및 굴절 효과를 적용한 결과

제 7장 객체 그리기

본 장에서는 물체를 랜더러로 보내어 영상으로 재현하는 방법에 대해서 설명한다. 물체는 특정 물성(Material)을 가진 3차원 공간상에 위치하는 가상의 삼각형들의 모임이라 할 수 있다. 이 집합은 특정 형태로 구성된다. 이 형태는 3차원 공간상에 존재하여 좌표의 변환이 이루어질 수 있다.

물체를 그리는 행위의 본질은 물성과 빛의 상관 관계를 통해서 최종색을 연산하는 것이다. 이런 연산 처리를 쉐이딩이라 한다. 쉐이딩을 수행하기 위한 알고리즘으로 실시간 처리 방식에서 가장 보편화된 기법이Phong 쉐이딩이다. Phong Illumination Model 이라고도 하는데, 빛의 입사각과 반사각, 물체의 주변광(ambient), 확산광(diffuse), 반사광(specular)으로 세분화된 속성을 통해서 연산하는 알고리즘이다. 여기에 다양한 빛의 효과를 보여주기 위해서 랜더링시에 물성을 따로 지정할 수 있다.

랜더링시에 실제 좌표 정보를 통해서 삼각형을 생성하고, 설정된 물성을 이용하여 물체를 생성하게 된다. 앞에서 언급한 바와 같이 특정 물체는 3차원 공간에서 표현할 수 있는 가장 작은 단위인 삼각형으로 구성할 수 있기 때문에, 삼각형을 기본 처리 단위, 즉 프리미티브(primitive)로 지정하여, 다수의 삼각형 집합으로 물체를 구성한다. 이 물체는 삼각형의 좌표들에 의해 3차원 공간상의 특정 모형으로 표현된다.

7.1 삼각형

3차원에서 삼각형은 매우 중요한 역할을 수행한다. 바로 물체를 표현하기 위한 가장 작은 단위로 사용되기 때문이다. 점, 선, 면은 3차원에서 물체를 표현하기 위해서 사용할 수 있는 대상이다. 점과 선은 3차원공간에서 표현하기에 어려움이 있다. 이론적으로 점과 점이 모여 선을 이루고, 선과 선이 모여 면을 이룬다. 점으로 표현된 선을 상상하면, 충분히 하나의 선으로 보여질 것이다. 하지만 시점이 선으로 가까이 다가가면 문제가

발생한다. 점과 점 사이에 공간이 발생하기 때문이다. 점을 충분히 사용하지 않을 경우 생기는 문제이다. 그렇다면 몇 개의 점을 사용해야 선을 표현하는데 충분한 것인지 아무도 알 수 없다. 이론적으로 점과 점 사이에는 무수히 많은 중간점들이 존재하기 때문이다. 이런 맹점을 해결하기 위해서 선분을 이용하여 선을 생성한다. 즉, 2개의 점 사이에 있는 중간점들은 매번 연산을 통해서 충분히 많은 점이 되도록 생성할 수 있다. 이런 방식을 그래픽 용어로 벡터 그래픽이라고 한다. 점으로 선을 표현하는 문제를 이렇게 해결하였으니 같은 방식으로 면도 표현할 수 있다.

모양이 각각인 면은 여러 개의 삼각형을 통해서 생성이 가능하다. 또한 평평한 면이 아닌 곡면인 경우, 예를 들어 공의 둥근 면도 작은 삼각형의 집합을 통해서 표현이 가능하다. 이를 응용하면 단순한 면뿐만 아니라 복잡한 물체도 표현이 가능하다. 따라서 물체 생성에 있어서 삼각형을 기본 단위로 한다.

a. 정점 목록(Vertex List)

삼각형을 구성하는 방식은 정점을 모아 목록화한 후 이를 하나의 집합으로 사용하는 것이다. 3개의 정점으로 하나의 삼각형을 구성한다. 따라서 삼각형을 생성하는데 2개의 방식이 존재한다. 첫번째 방식은 정점을 순차적으로 3개씩 저장하여 하나의 삼각형을 생성하는 방식이다. 두번째 방식은 정점을 저장하고, 삼각형이 될 정점의 색인값을 이용해서 삼각형을 생성하는 방식이다.

일반적으로 삼각형의 정점의 데이터는 적어도 3개의 좌표와 이에 해당하는 추가 정보를 가지고 있다. 법선 벡터 또는 텍스쳐 좌표가 그 예이다. 하지만 색인 데이터는 정수로 구성이 된다. 같은 위치에 있는 동일한 좌표가 중복 사용되어 삼각형이 만들어지는 경우 첫번째 방식에서는 데이터 크기가 상당히 커질 수 있다. 하지만 두번째 방식에서는 색인값이 중복될 뿐 실제 정점 데이터의 크기는 상대적으로 줄어들 수 있다. 이러한 방식은 데이터가 적어져서 메모리 사용량을 줄일 수 있다. 하지만, 추가 색인 정보를 따로 저장해야 하는 등의 부수적인 문제가 발생하게 된다. 이러한 이유로 물체의 저장 형태에 따라 최적화된 방식을 선택적으로 사용해야 한다.

b. 정점 데이터

정점 데이터는 다양한 구조로 저장이 가능하다. 필수적으로 사용되는 위치 좌표 (x, y, z)는 언제나 동일한 타입의 x, y, z 순서로 저장이 되어 있어야 한다. 정점의 경우 물성에 대한 정보를 함께 저장하여 사용할 수도 있다. 이런 경우는 데이터를 넘겨주는 함수를 통해서 명시해 주어야 한다.

void **rcVertexPointer** (RCint *size*, RCenum *type*, RCsizei *stride*, const RCvoid *pointer);

배열로 저장된 정점의 데이터를 입력하는 함수로 사용한다. 정점의 경우 입력되는 데이터에 대한 *size*와 *type*의 정보를 알려준다. 실제로 지원하는 정점 데이터의 좌표 (x, y, z) 개수 *size*는 3이다. 그 이외의 경우는 처리할 수 없다. *type*의 경우 RC_BYTE, RC_SHORT, RC_FLOAT, RC_FIXED를 지원한다. *stride*의 경우 정점 간의 간격을 바이크 크기로 명시한다. 정점의 좌표로만 구성될 경우 *stride*는 0을 지정하거나, 실제 크기를 지정하면 된다. 실제 데이터의 주소는 *pointer*에 입력한다.

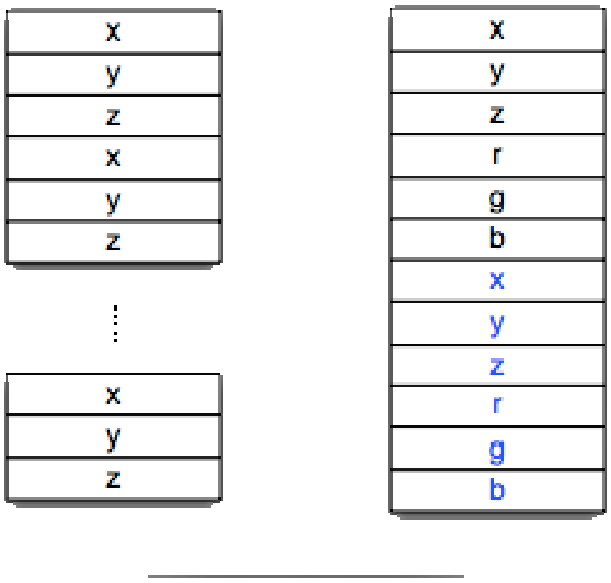

그림 31. 데이터 저장 방식

void **rcTexCoordPointer** (RCint *size*, RCenum *type*, RCsizei *stride*,
const RCvoid *pointer);

배열로 저장된 정점의 텍스쳐 좌표 데이터를 입력하는 함수로 사용한다. 정점의 경우 입력되는 데이터의 *size*와 *type*의 정보를 알려준다. 실제로 지원하는 텍스쳐 데이터의 좌표 (u, v) 개수 *size*는 2이다. 그 이외의 경우는 처리할 수 없다. *type*의 경우 RC_BYTE, RC_SHORT, RC_FLOAT, RC_FIXED를 지원한다. *stride*의 경우 텍스쳐 좌표 간의 간격을 바이트 크기로 명시한다. 각 정점의 테스쳐 좌표만 있을 경우 *stride*는 0을 지정하거나, 실제 크기를 지정하면 된다. 실제 데이터의 주소는 *pointer*에 입력한다.

텍스쳐의 경우 2차원 텍스쳐만을 지원하기 때문에 (u, v) 2개의 텍스쳐 좌표만을 사용한다.

void **rcNormalPointer** (RCenum *type*, RCsizei *stride*, const RCvoid *pointer);
배열로 저장된 정점의 법선 벡터 데이터를 입력하는 함수로 사용한다. 데이터의 *type*의 정보를 알려준다. *type*의 경우 RC_BYTE, RC_SHORT, RC_FLOAT, RC_FIXED를 지원한다. *stride*의 경우 법선 간의 간격을 바이트 크기로 명시한다. 각 정점의 법선 벡터만 있을 경우 *stride*는 0을 지정하거나, 실제 크기를 지정하면 된다. 실제 데이터의 주소는 *pointer*에 입력한다.

법선 벡터는 항상 (x, y, z) 3개의 좌표만을 사용하기 때문에 앞서의 두 함수와 달리 *size*를 따로 지정하지 않는다.

정점 배열을 구성할 때 위치와 물성에 대한 정보를 하나의 배열에 섞어서 사용하지 않도록 한다. 예를 들어 위치 정보와 물성을 섞어서 (XRYGZB)의 순서로 저장하여 사용할 수는 없다. 위치 정보는 순차적으로 저장되어 있어야 하고, 다른 정보도 순차적으로 저장해서 사용해야 한다.

각 요소들은 일반적으로 3개씩 구성된다. 위치에 대한 정보는 XYZ로 구성되며, 색은 RGB로 구성된다. 다만 텍스쳐 좌표에 대한 정보는 2개씩, 즉 UV로 구성된다. 이외의 경우는 현재로써는 지원하지 않는다.

7.2 삼각형 생성

정점 데이터는 정점 배열을 이용해서 입력한다. 이 데이터가 순서대로 삼각형을 구성하고 있으면, 그대로 사용할 수 있다. 그렇지 않으면, 이 정점의 색인 목록을 이용해서 삼각형을 구성할 수 있다. 색인을 사용할 경우 정점을 중복하여 저장할 필요가 없기 때문에 데이터 크기를 상당히 줄일 수 있다. 경우에 따라 이를 활용하면 매우 효과적이다. RayCore® API에서는 이 두가지 방법을 모두 지원한다.

a. 순차적인 정점 이용

정점 데이터가 반복적으로 사용되기 때문에, 메모리 사용에 있어서의 효율성은 감소하나 순차적인 삼각형의 처리가 가능하다.

void **rcDrawArray** (RCenum *mode*, RCint *first*, RCsizei *count*);

정점이 저장된 순서에 따라 삼각형을 생성한다. 삼각형의 생성 방식은 *mode*를 통해 지정된다. 지원하는 *mode* 설정은 RC_TRIANGLES, RC_QUADS, RC_TRIANGLE_FAN, RC_TRIANGLE_STRIP 이다. *rcVertexPointer*로 입력된 배열에서 처음 삼각형의 정점으로 사용할 색인을 *first*로 지정한다. *count*는 처리할 정점의 개수를 지정한다. 마지막 색인은 *first*+*count*-1가 된다.

b. 정점 색인 목록 이용

같은 정점 데이터를 공유하여 사용할 수 있기 때문에 메모리 사용량을 상당히 줄일 수 있다. 하지만, 정점을 순차적으로 사용하지 않기 때문에 대역폭이 늘어날 수 있다.

void **rcDrawElements** (RCenum *mode*, RCsizei *count,* RCenum *type*, *indices*);	RCvoid

정점이 저장된 순서에 따라 삼각형을 생성한다. *mode*를 통해 삼각형의 생성 방식이 지정된다. 지원하는 *mode* 설정은 RC_TRIANGLES, RC_QUADS, RC_TRIANGLE_FAN, RC_TRIANGLE_STRIP 이다. *rcVertexPointer*로 입력된 배열에 대한 색인 목록의 개수를 *count*에 지정한다. 목록 데이터의 자료형은 *type*에 지정한다. 목록 데이터의 주소를 *indices*를 이용해서 넘겨준다.

제 8장 랜더링의 시작

랜더링을 시작하는 지점을 명시한다. RayCore®는 후방향 랜더링의 레이트레이싱 방식으로 랜더링을 수행하기 때문이다. 랜더링할 객체를 모두 적재한 후 랜더링을 시작한다.

다음 함수를 통해서 랜더링의 시작을 명시한다.

void **rcFinish** (void);
랜더링의 시작을 알려주는 함수이다. 처리하고자 하는 데이터가 모두 적재되었음을 명시하고, 랜더링을 시작한다.

8.1 예제

예제에서 ***rcFinish***를 따로 사용하지 않는데, 부록 A에서 사용하는 ***RCFrameWork***의 내부에서 미리 지정해 두었기 때문에 따로 삽입할 필요가 없다.

제 9장 정적/동적 객체

RayCore® API에서는 영상에 반영할 객체를 메모리 상에 모두 적재한 후 랜더링을 수행하게 된다. 또한 각 데이터에 대한 가속 구조를 먼저 생성하여 사용한다. 이 과정은 반복되는 연산을 필요로 하게 되는데, 연산의 반복으로 인한 랜더링 속도의 지연을 발생시킨다. 이 문제를 해결하기 위해서 랜더링을 수행할 객체를 정적 객체와 동적 객체로 나누어 다루게 된다.

실시간 3차원 그래픽스의 대표적인 응용프로그램은 게임이다. 게임에는 배경이 되는 정적 객체가 있고, 배경 내에서 움직이는 동적 객체가 있다. 연극이나, 뮤지컬을 예로 들면 무대는 움직임이 거의 없는 정적 객체이고, 그 위에서 움직이는 배우들은 동적 객체라고 생각할 수 있다. 무대 장치는 막 사이에서만 변경이 될 뿐, 현재의 막이 지속되는 동안에는 바뀌지 않기 때문에 정적 객체라고 볼 수 있다. 하지만, 배우들은 무대에서 지속적으로 움직이며 연기를 수행하기 때문에 동적 객체라고 생각하면 된다.

래스터 방식에서는 이러한 구분이 없다. 왜냐하면 모든 프리미티브(primitive)를 순차적으로 처리하기 때문이다. 하지만, 레이트레이싱 방식에서는 프리미티브를 재활용하기 때문에 지속적인 프리미티브의 보관이 필요하다. 정적 객체의 경우 변경이 없다면, 메모리 상에 관련 정보를 그대로 유지하여 재활용할 수 있다. 이로 인해 가속 구조의 생성과 데이터의 적재에 필요한 비용을 상당히 줄일 수 있기 때문에 랜더링 속도를 향상시킬 수 있다.

따라서 객체를 정적/동적 객체로 구분하여 사용하면, 데이터 처리 속도를 높일 수 있다. 객체를 정적 객체로 구분하여 사용하는 방법을 본 장에서 설명한다.

9.1 객체 구분

3차원 좌표로 표현된 특정 단위의 물체를 객체라고 한다. 이 객체는 하나의 삼각형 프리미티브(primitive)로 구성되거나, 다수의 삼각형으로 구성된 캐릭터의 팔 혹은 다리 등으로 구성될 수도 있다. 아니면 캐릭터가 움직이는 공간 상의 맵이 될 수 있을 것이다. 이런 구분은 장면마다 달라질 수 있으며, 장면이 연결되는 지정된 시나리오에 따라 달라질 수도 있다.

래스터 방식은 모든 객체를 기본적으로 동적 객체로만 처리한다. RayCore® API에서는 그려야 할 데이터를 랜더러로 모두 보내고 난 후 영상을 생성한다. 여기에서의 구분은 동적 혹은 정적임을 기준으로 판단하지 않고 그려질 객체인지 아닌지를 판단하여 사용한다. 하지만, 레이트레이싱 랜더링에서는 그려할 지 아닌지를 쉽게 판단 할 수 없다. 왜냐하면 광선의 굴절 혹은 투과로 인하여 절두체 밖에 있는 객체들도 결과 영상에 영향을 주기 때문이다. 렌더러가 해당 객체가 움직이는지 고정되어 있는지를 알 수는 없다. 렌더러는 입력된 데이터를 기반으로 해당 장면의 영상을 생성할 뿐 장면이나 영상을 해석하거나 판단하지는 않는다. 렌더러에서는 가속 구조의 재생성 여부에 따라 동적/정적 객체의 구분이 이루어진다.

정적/동적 객체의 구분은 응용프로그램 내부에서 사용자가 별도로 관리해야 한다.

a. 동적 객체

동적 객체는 특정 장면 사이에서 움직임이 존재하는 객체를 의미한다. 어떤 장면에서는 움직이지 않고 있다가 특정 입력에 의해서 움직이게 되는 객체는 그 순간 동적 객체로 간주된다. 일반적으로 동적 객체는 움직이는 객체를 의미하지만, 실제 게임 응용프로그램을 개발하는 경우에는 움직일 가능성이 있는 것까지 포괄적으로 적용할 수 있다.

동적 객체로 처리한다는 것은 매번 가속 구조를 생성한다는 의미이다. 가속 구조 생성은 랜더링을 위한 전처리 과정으로 처리되지만, 실시간 랜더링 또는 인터렉티브 랜더링에서는 가속 구조 생성이 랜더링 처리 과정에 포함되는 것으로 보아야 한다.

정적 객체로 명확히 지시되지 않은 객체에 대해서는 동적 객체로 간주하여 처리한다. 즉, 입력되는 모든 데이터는 동적으로 움직일 가능성이 있다고 가정된다.

게임에서 사용하는 캐릭터가 대표적인 동적 객체이다. 캐릭터는 지속적으로 움직이면서 사용자와 상호 작용을 한다. 문이나 창문과 같은 객체는 거의 움직임이 없지만, 경우에 따라, 또는 상황에 따라 열리거나 닫히는 움직임을 보여 줄 수 있기 때문에 이는 동적 객체로 지정하여 사용할 수 있다. 물론 움직임이 필요 없는 캐릭터 혹은 문 등도 정적 객체로 지정하여 사용할 수 있다.

b. 정적 객체

정적 객체는 특정 장면 사이에서 변화가 없을 것으로 예상되거나, 변화가 없도록 모델링 또는 프로그램된 객체를 의미한다. 정적 객체로 지정되면 그 객체는 움직이지 않는 것으로 간주한다.

정적 객체로 지정되는 경우 해당 객체에 대한 가속 구조 생성 처리를 매 장면마다 재생성하지 않고, 한번 생성한 구조를 재활용하게 된다. 이는 가속 구조 연산을 줄이게 된다. 가속 구조 속에서 일부를 없애거나 첨가하게 되는 경우 가속 구조의 질이 나빠져 랜더링 속도가 매우 감소할 수 있다. 따라서 현재는 추가 삭제가 불가하다. 따라서, 가속 구조를 변경하기 위해서는 이를 재생성해야만 한다. 정적 객체의 재성성이 빈번한 경우 동적 객체로 사용하는 것과 차이가 거의 없게 된다. 결국 일정 시간동안 가속 구조 재생성이 필요없는 객체를 정적 객체로 지정하여 사용하는 것이 효율적이다. 예를 들어 게임에서 배경이나 지형 등은 일정 시간 동안 새로 업데이트를 수행하지 않아도 되기 때문에 정적 객체로 지정하여 사용할 수 있다.

c. 동적 속성의 정적 객체

정적 객체의 일부가 움직이는 경우에는 어떻게 될지 생각해 보자. 이런 경우 정적 객체들 중에서 움직이는 객체를 뺀 나머지 객체들로 가속 구조를 재생성해야 한다. 이와 동시에 움직이는 객체에 대해서 별도로 가속 구조를 생성해 주어야 한다. 이는 동적 객체로 사용하는 것과 차이가 없다.

정적 객체로 지정하였으나 동적 객체로 바뀌게 경우 이 객체가 정적 객체인지 동적 객체인지 불명확해진다. 움직임의 속성이 바뀌는 경우에는 이를 어떻게 지정하여 사용하는 것이 좋은지를 판단할 필요가 있다. 경우에 따라 처리할 데이터가 속도에 영향을 미칠 수도 있고 그렇지 않을 수도 있다. 이런 객체들은 별도의 관리가 필요하기 때문에 그 비용에 대해서도 고려할 필요가 있다.

두가지 속성을 모두 가질 수 있는 객체에 대해서는 RayCore® API 단계에서 자동으로 수행되지는 않는다. 상위 응용프로그램에서의 처리 방식에 따라 달라질 수 있기 때문에, 응용프로그램을 작성할 때 별도로 지정하여 그에 맞는 응용프로그램을 작성해야만 한다.

여기에서 기억할 중요한 것은 다음과 같다.

- 정적 객체는 한번 가속 구조를 생성하면 이를 초기화 하기 전까지는 다시 가속 구조를 생성하지 않는다.
- 정적 객체를 사용하지 않을 경우 객체 정보를 초기화하여 데이터를 사용하지 않도록 한다.
- 정적으로 지정하지 않은 객체는 모두 동적 객체로 취급한다.

동적 객체와 다르게 랜더링할 객체가 정적인 모델임을 지시해주는 방법에 대해서 알아보자.

9.2 정적 객체 프로그래밍

동적 객체에 대해서는 위에서 설명한 방식을 그대로 사용한다. 매 장면이 생성될 때마다 필요에 따라 동일한 작업을 수행하면 된다. 입력에 따라 객체가 바뀌는 경우 바뀐 위치로 좌표 이동을 수행하여 데이터를 적재한다.

a. 프로그래밍 모델

정적 객체를 지정하는 프로그래밍 모델은 간단한 begin-end 모델을 사용한다. 객체의 시작을 알리는 begin 함수를 사용하고, 끝을 알리는 end 함수를 사용한다.

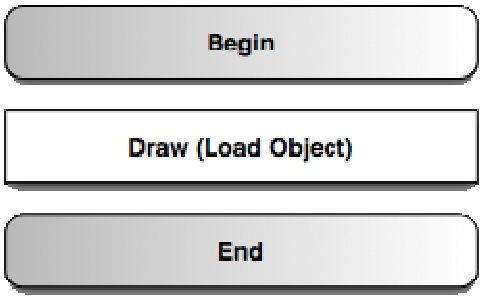

그림 32. 정적 객체에 대한 begin-end 모델

b. 함수 목록

정적 객체를 지정하기 위한 기본 적재 및 변환 함수와 물성 지정 함수는 동일하게 사용한다. 단지 정적 객체가 적재되기 시작하는 지점과 끝나는 지점에 대한 지정이 필요할 뿐이다. 따라서 RayCore® API에서는 이에 대한 함수를 지원한다. 정적 객체를 사용하지 않을 경우 이를 지정할 수 있는 함수가 지원되어 정적 객체에 대해서 사용 또는 삭제를 쉽게 할 수 있다.

정적 객체를 초기화 또는 삭제하는 함수는 다음과 같다.

void **rcSceneAllInit** (void);
정적 객체를 초기화할 때 사용한다. 정적 객체를 사용 중인 경우 이를 삭제하는 역할을 한다.

정적 객체의 적재 시작을 알리는 함수는 다음과 같다.

void **rcStaticSceneBegin** (void);

이 함수를 통해서 정적 객체 데이터가 적재되기 시작하는 순간을 지정한다. 이 함수를 호출하지 않고, 적재되는 모든 데이터는 동적 데이터로 간주하여 저장된다. 따라서 정적 객체를 사용하기 위해서는 필히 이 함수를 사용하여 객체를 지정해야 한다.

정적 객체의 적재가 완료됨을 지시하는 함수는 다음과 같다.

void **rcStaticSceneEnd** (void);
정적 객체가 모두 적재 되었음을 지정하는 함수이다. 이 함수를 사용하여 정적 객체의 가속 구조를 생성하는 시점을 지시한다.

9.3 주의점

정적 객체를 사용하는데 있어서 지켜야할 원칙은 다음과 같다.

- 정적 객체를 여러 개로 나누어 지정하지 않는다.
- 정적 객체 데이터를 메모리의 앞에 적재하여 고정시켜야 한다.

이런 원칙을 기준으로 프로그래밍할 때 주의해야 할 사항은 다음과 같다.

- 정적 객체는 하나의 장면에서 하나만을 사용해야 한다.
- 먼저 정적 객체를 생성한 후에 동적 객체를 생성한다.

9.4 예제 코드(Cornell Box*)

다음 예제는 확정 함수를 이용하여 정적 배경 객체를 설정하는 방법을 설명하는 것이다.

a. 초기화

기본 초기 파라미터는 다음과 같이 정의된다.

```
rcSceneAllInit();

rcClearColor(0.0f, 0.2f, 0.4f, 1.0f);

rcMatrixMode(RC_PROJECTION);
rcLoadIdentity();

rcFrustum(-1, 1, -0.6f, 0.6f, 1.7, 1000);
```

* Designed in Cornell university

```
rcuLookAt(27.8, 27.3, -80.0, 27.8, 27.3, 0, 0, 1, 0);

rcStaticSceneBegin();
rcEnableClientState(RC_VERTEX_ARRAY);

rcVertexPointer(3, RC_FLOAT, 0, Left_Wall);
rcGenMaterials(1, &materialID[0]);
rcBindMaterial(materialID[0]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE,&Red.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcVertexPointer(3, RC_FLOAT, 0, Right_Wall);
rcGenMaterials(1, &materialID[1]);
rcBindMaterial(materialID[1]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE,&Green.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcVertexPointer(3, RC_FLOAT, 0, Floor);
rcGenMaterials(1, &materialID[2]);
rcBindMaterial(materialID[2]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, &White.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcVertexPointer(3, RC_FLOAT, 0, Ceiling);
rcGenMaterials(1, &materialID[3]);
rcBindMaterial(materialID[3]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, &White.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcVertexPointer(3, RC_FLOAT, 0, Back_Wall);
rcGenMaterials(1, &materialID[4]);
rcBindMaterial(materialID[4]);
rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, &White.r);
rcDrawArrays(RC_TRIANGLES, 0, 6);

rcDisableClientState(RC_VERTEX_ARRAY);
rcStaticSceneEnd();
```

예제에는 배경이 되는 상자 내부에 2개의 상자가 있다. 위에서는 ***rcSceneAllInit()*** 함수를 이용하여 정적 객체를 초기화하고 기본 초기 파라미터를 설정하였다. 이후 ***rcStaticSceneBegin()***, ***rcStaticSceneEnd()*** 두 함수 사이에 배경이 되는 상자를 정적 객체로 설정하고 있다.

b. 장면(Scene) 랜더링

장면에 대한 랜더링은 광원에 대한 정의와 정점 데이터에 대한 설정, 두 부분으로 나뉜다.

4개의 광원은 천정에 위치한다. 각 광원은 켜거나 끌 수 있다. 조명이 비활성화되어 있는 경우, 앞서 설명한 대로 기본 광원이 자동적으로 시점 위치에 생성된다.

```
rcMatrixMode(RC_MODELVIEW);
rcLoadIdentity();

rcRotatef(-g_fSpinY, 1.0f, 0.0f, 0.0f );
rcRotatef(-g_fSpinZ, 0.0f, 1.0f, 0.0f );

if (Light){
      rcEnable(RC_LIGHTING);
      rcLightfv(RC_LIGHT0, RC_AMBIENT, Light_Ambient);
      rcLightfv(RC_LIGHT0, RC_DIFFUSE, Light_Diffuse);
      rcLightfv(RC_LIGHT0, RC_POSITION, Light_Position1);
```

```
        rcEnable(RC_LIGHT0);

        rcLightfv(RC_LIGHT1, RC_AMBIENT, Light_Ambient);
        rcLightfv(RC_LIGHT1, RC_DIFFUSE, Light_Diffuse);
        rcLightfv(RC_LIGHT1, RC_POSITION, Light_Position2);
        rcEnable(RC_LIGHT1);

        rcLightfv(RC_LIGHT2, RC_AMBIENT, Light_Ambient);
        rcLightfv(RC_LIGHT2, RC_DIFFUSE, Light_Diffuse);
        rcLightfv(RC_LIGHT2, RC_POSITION, Light_Position3);
        rcEnable(RC_LIGHT2);

        rcLightfv(RC_LIGHT3, RC_AMBIENT, Light_Ambient);
        rcLightfv(RC_LIGHT3, RC_DIFFUSE, Light_Diffuse);
        rcLightfv(RC_LIGHT3, RC_POSITION, Light_Position4);
        rcEnable(RC_LIGHT3);
    } else
        rcDisable(RC_LIGHTING);
```

c. 객체 그리기

2개의 상자가 있다. 하나는 움직이지 않지만, 다른 하나는 점핑한다. 이와 같은 동적 객체는 OpenGL 프로그래밍과 동일하게 랜더링 루프에서 처리한다.

```
rcEnableClientState(RC_VERTEX_ARRAY);

rcPushMatrix();
{
    if (dir)
        aniy += 0.1;
    else
        aniy -= 0.1;

    if (aniy < 0) {
        aniy = 0;
        dir = true;
    } else if (aniy > 10) {
        aniy = 10;
        dir = false;
    }
    rcTranslatef(0, aniy, 0);
}

rcVertexPointer(3, RC_FLOAT, 0, Short_Block);
rcBindMaterial(MaterialID_ShortBlock);
rcColor4f(White1.r, White1.g, White1.b, 0);
rcDrawArrays(RC_TRIANGLES, 0, 30);
rcPopMatrix();

rcVertexPointer(3, RC_FLOAT, 0, Tall_Block);
rcBindMaterial(MaterialID_TallBlock);
rcColor4f(White2.r, White2.g, White2.b, 0);
rcDrawArrays(RC_TRIANGLES, 0, 30);

rcDisableClientState(RC_VERTEX_ARRAY);
```

d. 정점 데이터

정점 테이터는 다음과 같다.

```
Pos Left_Wall[] = {
    {55.28, 0.0, 0.0}, {54.96,    0.0, 55.92}, {55.60, 54.88, 55.92},
    {55.28, 0.0, 0.0}, {55.60, 54.88, 55.92}, {55.60, 54.88,    0.0}
```

```
};
Pos Right_Wall[] = {
    {0.0, 54.88, 0.0}, {0.0, 54.88, 55.92}, {0.0, 0.0, 55.92},
    {0.0, 54.88, 0.0}, {0.0,     0.0, 55.92}, {0.0, 0.0,     0.0}
};
Pos Floor[] = {
    {55.28, 0.0, 0.0}, {0.0, 0.0,     0.0}, {    0.0, 0.0, 55.92},
    {55.28, 0.0, 0.0}, {0.0, 0.0, 55.92}, {54.96, 0.0, 55.92}
};
Pos Ceiling[] = {
    {55.60, 54.88, 0.0}, {55.60, 54.88, 55.92}, {0.0, 54.88, 55.92},
    {55.60, 54.88, 0.0}, {    0.0, 54.88, 55.92}, {0.0, 54.88,     0.0}
};
Pos Back_Wall[] = {
    {0.0, 54.88, 55.92}, {55.60, 54.88, 55.92}, {54.96, 0.0, 55.92},
    {0.0, 54.88, 55.92}, {54.96,     0.0, 55.92}, {    0.0, 0.0, 55.92}
};
Pos Short_Block[] = {
    {13.0, 16.5,    6.5}, {  8.2, 16.50, 22.5}, {24.0, 16.5, 27.2},
    {13.0, 16.5,    6.5}, {24.0, 16.50, 27.2}, {29.0, 16.5, 11.4},
    {29.0,    0.0, 11.4}, {29.0, 16.50, 11.4}, {24.0, 16.5, 27.2},
    {29.0,    0.0, 11.4}, {24.0, 16.50, 27.2}, {24.0,    0.0, 27.2},
    {13.0,    0.0,    6.5}, {13.0, 16.50,    6.5}, {29.0, 16.5, 11.4},
    {13.0,    0.0,    6.5}, {29.0, 16.50, 11.4}, {29.0,    0.0, 11.4},
    { 8.2,    0.0, 22.5}, {  8.2, 16.50, 22.5}, {13.0, 16.0,    6.5},
    { 8.2,    0.0, 22.5}, {13.0, 16.50,    6.5}, {13.0,    0.0,    6.5},
    {24.0,    0.0, 27.2}, {24.0, 16.50, 27.2}, {8.20, 16.5, 22.5},
    {24.0,    0.0, 27.2}, {  8.2, 16.50, 22.5}, {8.20,    0.0, 22.5}
};
Pos Tall_Block[] = {
    {42.3, 33.0, 24.7}, {26.50, 33.0, 29.6}, {31.40, 33.0, 45.6},
    {42.3, 33.0, 24.7}, {31.40, 33.0, 45.6}, {47.20, 33.0, 40.6},
    {42.3,    0.0, 24.7}, {42.30, 33.0, 24.7}, {47.20, 33.0, 40.6},
    {42.3,    0.0, 24.7}, {47.20, 33.0, 40.6}, {47.20,    0.0, 40.6},
    {47.2,    0.0, 40.6}, {47.20, 33.0, 40.6}, {31.40, 33.0, 45.6},
    {47.2,    0.0, 40.6}, {31.40, 33.0, 45.6}, {31.40,    0.0, 45.6},
    {31.4,    0.0, 45.6}, {31.40, 33.0, 45.6}, {26.50, 33.0, 29.6},
    {31.4,    0.0, 45.6}, {26.50, 33.0, 29.6}, {26.50,    0.0, 29.6},
    {26.5,    0.0, 29.6}, {26.50, 33.0, 29.6}, {42.30, 33.0, 24.7},
    {26.5,    0.0, 29.6}, {42.30, 33.0, 24.7}, {42.30,    0.0, 24.7}
};
```

e. 랜더링 결과

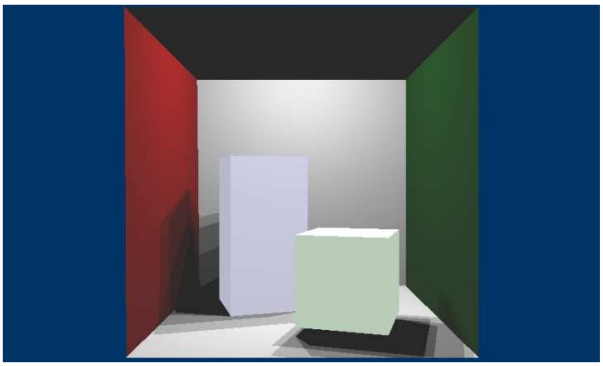

그림 33. 랜더링 결과

부록 A.

Framework

```
#ifndef __EGL_FRAMEWORK_H__
#define __EGL_FRAMEWORK_H__
#ifdef WIN32
  #include "platform/WindowsPlatform.h"
  #ifndef RC_FRAMEWORK
        #pragma comment(lib, "RCFramework.lib")
  #endif
#else
  #include "platform/X11Platform.h"
#endif

class RCFramework :
  protected WindowPlatform
{
protected:
  virtual ~RCFramework(void);

public:
  RCFramework(const char* sTitleName, int width, int height);

  BOOL Initialize(void);
  void Release(void);
  BOOL DrawScene(void);

  inline EGLDisplay CurrentDisplay(void)  {return m_eglDisplay;}
  inline EGLContext CurrentContext(void) {return m_eglContext;}
  inline EGLSurface CurrentSurface(void)  {return m_eglSurface;}
  inline int Width(void)          {return m_iWidth;}
  inline int Height(void)         {return m_iHeight;}

protected:
  virtual BOOL OnIntialize(void);
  virtual void OnRelease(void);
  virtual BOOL OnDraw(void);
  virtual BOOL OnPostDraw(void);

private:
  EGLDisplay        m_eglDisplay;
  EGLContext        m_eglContext;
  EGLSurface        m_eglSurface;
  int               m_iWidth, m_iHeight;
  const char*       m_sTitle;
};
```

```
#endif//__EGL_FRAMEWORK_H__
```

부록 B.
예제 프로그램

RayCore® 프로그래밍 모델은 전역 초기화와 데이터 전송의 두 단계로 구성되어 있다. 전역 초기화 단계에서는 절두체, 시야 및 랜더링에 사용되는 텍스쳐 데이터에 대한 전역값을 설정하며, 두번째 단계는 기하 변환을 갖는 데이터 열을 전송한다. 최종 이미지는 이러한 정보를 바탕으로 레이트레이싱 알고리즘을 구현한RayCore® 하드웨어에 의해 생성된다. 두번째 단계는 모든 장면에서 반복된다. 이러한 과정으로 장면과 장면 사이에서 제어 이벤트를 처리한다.

1.1 Cube 객체

텍스쳐를 가진 큐브에 대한 프로그래밍 예제이다. 위에서 설명한대로 두 단계로 프로그래밍한다.

a. 장면(Scene) 초기화

초기 설정 단계에서 몇 가지 설정은 RayCore® API 함수로 이루어진다.

배경색은 (0, 0, 0, 1)의 검은색으로 설정되어 있다. 만약 객체에 광선이 교차되지 않으면, 프레임 버퍼에 배경색이 저장된다.

RayCore®에서는 깊이 테스트는 사용하지 않는다. 깊이 테스트가 활성화되어 있어도 아무런 영향을 주지 않는다.

투영 행렬을 설정한다. 이는 절두체 정보에 의해 설정된다. 이는 OpenGL과 동일하다.

OpenGL에서 시점은 항상 원점 좌표로 설정되어 있다. 그러나, 레이트레이싱은 임의의 좌표를 사용할 수 있다. ***rcuLookAt*** 함수를 사용하여 시점을 설정한다. 투영 행렬에 영향을 주지는 않는다.

```
rcClearColor(0.0f, 0.0f, 0.0f, 1.0f);

rcMatrixMode(RC_PROJECTION);
rcLoadIdentity();
rcFrustumf(-0.9622504483f, 0.9622504483f, -0.577350269f, 0.577350269f, 1, 100);
rcuLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);
```

b. 텍스쳐 적재

장면에서 다중 텍스쳐를 사용할 수 있다. 이들은 랜더링 전에 로딩된다. 우선 텍스쳐 객체는 ***rcGenTextures***로 생성된다. 각 텍스쳐 ID는 배열에 저장된다. 텍스쳐 로딩 함수를 호출하기 전에, 버퍼 ID로 현재 텍스쳐 ID를 설정한다. 현재 ID에 의해 텍스쳐가 n번째 텍스쳐 객체에 로딩된다. 모든 텍스쳐 이미지는 현재 ID 설정 및 데이터 로드에 의해 로딩된다. 이 예제에서는 큐브의 모든 면에 대해 6개의 텍스쳐를 사용한다.

텍스쳐는 ***rcTexImage***에 의해 밉맵(mipmap)으로 사용된다.

텍스쳐는 단지 한번만 로딩한다. 왜냐하면, 매번 텍스쳐를 로딩하는 것이 부담되기 때문이다.

```
rcGenTextures(6, TextureArray);

rcBindTexture(RC_TEXTURE_2D, TextureArray[Blender]);
rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                    pBitmap[Blender]->sizeX, pBitmap[Blender]->sizeY,
                    0, RC_RGB, RC_UNSIGNED_BYTE,
                    pBitmap[Blender]->data);
```

c. 모델

모델은 두 가지 방법으로 표현된다. 하나는 인덱스 모드이고, 다른 하나는 순차적인 삼각형 모드이다. 인덱스 모드는 인덱스와 정점 좌표에 대한 2개의 배열을 설정한다. 텍스쳐 매핑을 사용하는 경우, 텍스쳐 좌표 배열이 추가된다. 또한, 정점 배열에 대한 법선 벡터를 사용할 수 있다.

정점 위치는 배열로 저장된다. 이를 연속적으로 사용하지는 않는다. 이 정점에 대한 인덱스 번호를 갖는다.

```
Pos g_cubePos_indexed[] = {
      {-1.0f, -1.0f, 1.0f},      // 0
      {1.0f, -1.0f, 1.0f},       // 1
      {1.0f, 1.0f, 1.0f},        // 2
      {-1.0f, 1.0f, 1.0f},       // 3
```

```
        {-1.0f,-1.0f, -1.0f},      // 4
        {-1.0f, 1.0f, -1.0f},      // 5
        {1.0f, 1.0f, -1.0f},       // 6
        {1.0f,-1.0f, -1.0f},       // 7
};
```

텍스쳐 좌표도 정점과 동일하다. 그러나, 2D 이미지에 대한 것이므로, 2차원을 갖는다. 1번 정점은 1번 텍스쳐 좌표를 사용한다.

```
TexC g_cubeTexcoord_indexed[] = {
        {0.0f, 0.0f},        // 0
        {1.0f, 0.0f},        // 1
        {1.0f, 1.0f},        // 2
        {0.0f, 1.0f},        // 3
        {0.0f, 0.0f},        // 4
        {0.0f, 1.0f},        // 5
        {1.0f, 1.0f},        // 6
        {1.0f, 0.0f},        // 7
};
```

다음은 인덱스 배열을 보여준다. 첫번째 4개의 인덱스가 하나의 사각형을 만든다. 다음 4개의 인덱스는 그 다음 사각형을 만든다.

```
RCubyte g_cubeIndices[] =
{
        0, 1, 2, 3,     // Quad 0
        4, 5, 6, 7,     // Quad 1
        5, 3, 2, 6,     // Quad 2
        4, 7, 1, 0,     // Quad 3
        7, 6, 2, 1,     // Quad 4
        4, 0, 3, 5      // Quad 5
};
```

연속 데이터 모드는 인덱스없이 저장된다. 이는 같은 정점이 반복적으로 저장된다. 그러나, 순차적으로 사각형을 형상한다. 다음의 코드에서 이를 보여준다.

```
Pos g_cubePos[] =
{
        // Quad 0
        {-1.0f, -1.0f, 1.0f},      // 0 (unique)
        {1.0f, -1.0f, 1.0f},       // 1 (unique)
        {1.0f, 1.0f, 1.0f},        // 2 (unique)
        {-1.0f, 1.0f, 1.0f},       // 3 (unique)
        // Quad 1
        {-1.0f, -1.0f, -1.0f},     // 4 (unique)
        {-1.0f, 1.0f, -1.0f},      // 5 (unique)
        {1.0f, 1.0f, -1.0f},       // 6 (unique)
        {1.0f, -1.0f, -1.0f},      // 7 (unique)
        // Quad 2
        {-1.0f, 1.0f, -1.0f},      // 5 (start repeating here)
        {-1.0f, 1.0f, 1.0f},       // 3 (repeat of vertex 3)
        {1.0f, 1.0f, 1.0f},        // 2 (repeat of vertex 2... etc.)
        {1.0f, 1.0f, -1.0f},       // 6
        // Quad 3
        {-1.0f, -1.0f, -1.0f},     // 4
        {1.0f, -1.0f, -1.0f},      // 7
        {1.0f, -1.0f, 1.0f},       // 1
        {-1.0f, -1.0f, 1.0f},      // 0
        // Quad 4
        {1.0f, -1.0f, -1.0f},      // 7
        {1.0f, 1.0f, -1.0f},       // 6
```

```
        {1.0f, 1.0f, 1.0f},       // 2
        {1.0f, -1.0f, 1.0f},      // 1
        // Quad 5
        {-1.0f, -1.0f, -1.0f},    // 4
        {-1.0f, -1.0f, 1.0f},     // 0
        {-1.0f, 1.0f, 1.0f},      // 3
        {-1.0f, 1.0f,-1.0f}       // 5
};
```

d. 장면(Scene) 랜더링

최종 이미지는 랜더링 장면 처리에 의해 생성된다. 이는 프로그램이 종료될 때까지 반복된다. 3가지를 설정한다. 광원 설정, 기하 변환 정보, 동적 객체 데이터이다.

광원이 설정되지 않아도, RayCore® API는 시점의 위치에 하나의 빛을 설정한다. 왜냐하면 최종색은 광원없이는 보여지지 않기 때문이다. 이 예제에서는 광원을 설정하지 않는다.

기하 변환을 설정한다. 이는 OpenGL 방법과 같다. 모델뷰 행렬은 ***rcTranslatef***, ***rcRotatef***, ***rcScalef*** 등의 변환 함수에 의해 생성된다. 이 모델뷰 행렬은 각 정점 좌표에 대해서 계산된다. 정점 배열과 텍스쳐 좌표 배열이 사용되는데, ***rcEnableClientState***함수로 활성화된다.

```
rcMatrixMode( RC_MODELVIEW );
rcLoadIdentity();
rcTranslatef(0.0f, 0.0f, -6.0f);
rcRotatef(-g_fSpinY, 1.0f, 0.0f, 0.0f);
rcRotatef(-g_fSpinZ, 0.0f, 1.0f, 0.0f);
rcEnableClientState(RC_VERTEX_ARRAY);
rcEnableClientState(RC_TEXTURE_COORD_ARRAY);
```

rcVertexPointer는 정점 배열 주소를 연결한다. RC_TEXTURE_2D 파라미터에 의해 할성화된 택스쳐 맵핑을 물성에 연결한다. 여러 파라미터가 속성에 사용되도록 전달된다.

```
rcVertexPointer(3, RC_FLOAT, 0, g_cubePos_indexed);
rcTexCoordPointer(2, RC_FLOAT, 0, g_cubeTexcoord_indexed);

rcBindMaterial(MaterialIDs[Blender]);
rcEnable(RC_TEXTURE_2D);
rcBindTexture(RC_TEXTURE_2D, TextureArray[Blender]);
rcDrawElements(RC_TRIANGLE_FAN, 4, RC_UNSIGNED_BYTE, &g_cubeIndices[0] );

rcBindMaterial(MaterialIDs[Coffeematic]);
rcEnable(RC_TEXTURE_2D);
rcBindTexture(RC_TEXTURE_2D, TextureArray[Coffeematic]);
rcDrawElements(RC_TRIANGLE_FAN, 4, RC_UNSIGNED_BYTE, &g_cubeIndices[4] );
rcVertexPointer(3, RC_FLOAT, 0, g_cubePos);
rcTexCoordPointer(2, RC_FLOAT, 0, g_cubeTexs);

rcBindMaterial(MaterialIDs[Blender]);
rcEnable(RC_TEXTURE_2D);
rcBindTexture(RC_TEXTURE_2D, TextureArray[Blender]);
rcDrawArrays(RC_TRIANGLE_FAN, 0, 4);

rcBindMaterial(MaterialIDs[Coffeematic]);
rcEnable(RC_TEXTURE_2D);
rcBindTexture(RC_TEXTURE_2D, TextureArray[Coffeematic]);
rcDrawArrays(RC_TRIANGLE_FAN, 4, 4);
```

e. 랜더링 결과

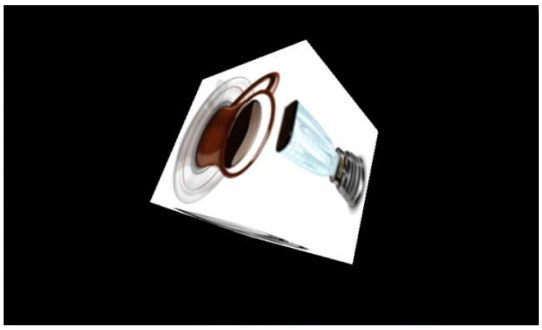

그림 34. 랜더링 결과

부록 C.
Earth 예제

광원을 이동하는 예제이다.

```
#include "RCFramework.h"

// Programming Guide - Earth Example

#include "earth.h"

typedef struct RGBImageRec {
  int sizeX, sizeY;
  unsigned char *data;
} RGBImageRec;

typedef struct bmpBITMAPFILEHEADER{
  WORD    bfType;
  DWORD  bfSize;
  WORD    bfReserved1;
  WORD    bfReserved2;
  DWORD  bfOffBits;
} BITMAPFILEHEADER1;

typedef struct bmpBITMAPINFOHEADER{
  DWORD  biSize;
  DWORD  biWidth;
  DWORD  biHeight;
  WORD    biPlanes;
  WORD    biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  DWORD  biXPelsPerMeter;
  DWORD  biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPFILEHEADER2;

RGBImageRec *DIBImageLoad(char* path, int channel) {
  RGBImageRec* pImage=NULL;
  FILE *f=NULL;
  unsigned char *pBuf=NULL;
  int dataSize=0;
  int index=0;
```

```
  DWORD x=0;
  DWORD y=0;
  int bpp=0;

  if(channel !=3 && channel !=4) return pImage;

  f = fopen(path, "rb");
  if(f != NULL) {
        BITMAPFILEHEADER1 HD1;
        BITMAPFILEHEADER2 HD2;

        fseek(f, 0, SEEK_SET);
        fread(&HD1.bfType, sizeof(WORD), 1, f);
        fread(&HD1.bfSize, sizeof(WORD), 1, f);
        fseek(f, 10, SEEK_SET);
        fread(&HD1.bfOffBits, sizeof(int),1, f);

        fread(&HD2.biSize, sizeof(int), 1, f);
        fread(&HD2.biWidth, sizeof(int), 1, f);
        fread(&HD2.biHeight, sizeof(int), 1, f);
        fread(&HD2.biPlanes, sizeof(WORD), 1, f);
        fread(&HD2.biBitCount, sizeof(WORD), 1, f);
        fread(&HD2.biCompression, sizeof(int), 1, f);

        fseek(f,HD1.bfOffBits,SEEK_SET);

        bpp = HD2.biBitCount/8;
        if(bpp == 1 || bpp == channel)
        {
              pBuf = (unsigned char*) malloc(channel);

              pImage = (RGBImageRec*) malloc(sizeof(RGBImageRec));
              pImage->sizeX = HD2.biWidth;
              pImage->sizeY = HD2.biHeight;

              dataSize = HD2.biWidth*HD2.biHeight*channel;
              pImage->data = (unsigned char*) malloc(dataSize);

              for(y=0; y<HD2.biHeight; y++) {
                    for(x=0; x<HD2.biWidth; x++) {
                          fread(pBuf, bpp, 1, f);
                          if(bpp == 1) {
                                pBuf[1] = pBuf[2] = pBuf[0];
                                if(channel == 4)      pBuf[3] = 0;
                          }

                          index = (y*HD2.biWidth + x)*channel;
                          pImage->data[index]       = pBuf[2];
                          pImage->data[index + 1]  = pBuf[1];
                          pImage->data[index + 2]  = pBuf[0];
                          if(channel == 4)
                                pImage->data[index + 3] = pBuf[3];
                    }
              }

              if(pBuf) free(pBuf);
              pBuf = NULL;
        }

        fclose(f);
  }

  return pImage;
}

#define Galaxy            0
#define Earth             1

unsigned int TextureArray[2];
unsigned int MaterialIDs[2];

float Light_Position[] = {-17.802584, 0.353599, 8.770458, 1};
float Light_Ambient[] = {0, 0, 0};
float Light_Diffuse[] = {1, 1, 1};
float Light_Specular[] = {1, 1, 1};
```

```
float g_turn = 35;

class Tutorial : public RCFramework
{
public:
  Tutorial(void) : RCFramework("Programming Guide - Earth Example", 800, 480){}
  virtual ~Tutorial(void){}

  void Load(RGBImageRec **pBitmap){

        pBitmap[0] = (RGBImageRec *)DIBImageLoad("./scenedata/Guide_Earth/Galaxy.bmp", 3);
        pBitmap[1] = (RGBImageRec *)DIBImageLoad("./scenedata/Guide_Earth/Earth.bmp", 3);
  }

  void CreateTexture(void){
        RGBImageRec *pBitmap[2];
        int i=0;

        for(i=0; i<2; i++) {
              pBitmap[i] = NULL;
        }
        Load(pBitmap);

        rcGenTextures(2, TextureArray);

        rcBindTexture(RC_TEXTURE_2D, TextureArray[Galaxy]);
        rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                        pBitmap[Galaxy]->sizeX, pBitmap[Galaxy]->sizeY,
                        0, RC_RGB, RC_UNSIGNED_BYTE,
                        pBitmap[Galaxy]->data);

        rcBindTexture(RC_TEXTURE_2D, TextureArray[Earth]);
        rcTexImage2D(RC_TEXTURE_2D, 0, RC_RGB,
                        pBitmap[Earth]->sizeX, pBitmap[Earth]->sizeY,
                        0, RC_RGB, RC_UNSIGNED_BYTE,
                        pBitmap[Earth]->data);

        for(i=0; i<2; i++) {
              if(pBitmap[i]) {
                    if(pBitmap[i]->data) free(pBitmap[i]->data);
                    free(pBitmap[i]);
              }
        }
  }

  void SetMaterial(void){
        float AmbientDiffuse[2][3] = {
              {0.588235, 0.588235, 0.588235},
              {0.529412, 0.529412, 0.529412},
        };

        float Specular[2][3] = {
              {0, 0, 0},
              {1, 0, 0},
        };

        float Shininess[2] = {
              2.000000,
              3.031433,
        };

        rcEnable(RC_TEXTURE_2D);

        rcGenMaterials(1, &MaterialIDs[Galaxy]);
        rcBindMaterial(MaterialIDs[Galaxy]);
        rcBindTexture(RC_TEXTURE_2D, TextureArray[Galaxy]);
        rcMaterialfv(RC_FRONT_AND_BACK,
                        RC_AMBIENT_AND_DIFFUSE,
                        AmbientDiffuse[Galaxy]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, Specular[Galaxy]);
        rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, Shininess[Galaxy]);

        rcGenMaterials(1, &MaterialIDs[Earth]);
        rcBindMaterial(MaterialIDs[Earth]);
        rcBindTexture(RC_TEXTURE_2D, TextureArray[Earth]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT_AND_DIFFUSE, AmbientDiffuse[Earth]);
```

```
        rcMaterialfv(RC_FRONT_AND_BACK, RC_SPECULAR, Specular[Earth]);
        rcMaterialf(RC_FRONT_AND_BACK, RC_SHININESS, Shininess[Earth]);
    }

    void StaticSceneDraw(void){
        RCuint materialID;

            float galaxyV[12] = {
                -1, -1,   0,
                 1, -1,   0,
                 1,  1,   0,
                -1,  1,   0
            };

            float galaxyT[8] = {
                0, 0,
                1, 0,
                1, 1,
                0, 1
            };

    rcBindMaterial(MaterialIDs[Galaxy]);
    rcEnable(RC_TEXTURE_2D);
    rcBindTexture(RC_TEXTURE_2D, TextureArray[Galaxy]);

    rcStaticSceneBegin();

    rcEnableClientState(RC_VERTEX_ARRAY);
    rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

    rcVertexPointer(3, RC_FLOAT, 0, galaxyV);
    rcTexCoordPointer(2, RC_FLOAT, 0, galaxyT);

    rcMatrixMode(RC_MODELVIEW);
    rcLoadIdentity();
    rcTranslatef(0, 0, -18.0);
    rcScalef(8, 4.8, 1);

    rcEnable(RC_USE_TEXTURE_ONLY);
    rcDrawArrays(RC_QUADS, 0, 4);
    rcDisable(RC_USE_TEXTURE_ONLY);

    rcDisableClientState(RC_VERTEX_ARRAY);
    rcDisableClientState(RC_TEXTURE_COORD_ARRAY);

    rcStaticSceneEnd();
    }

protected:
    virtual BOOL OnIntialize(void){

rcDepthBounce(14);

rcSceneAllInit();

        {
            rcClearColor(0.1f, 0.1f, 0.1f, 1.0f);
            rcViewport(0, 0, Width(), Height());

            rcMatrixMode(RC_PROJECTION);
            rcLoadIdentity();

            rcuPerspective(30, (float)Width() /(float)Height(), 10, 10000 );

            rcuLookAt(0, 0, 0, 0, 0, -1, 0, 1, 0);

            CreateTexture();
            SetMaterial();

            {
                rcEnable(RC_LIGHTING);
                rcEnable(RC_LIGHT0);
                rcLightfv(RC_LIGHT0, RC_POSITION, Light_Position);
                rcLightfv(RC_LIGHT0, RC_AMBIENT, Light_Ambient);
                rcLightfv(RC_LIGHT0, RC_DIFFUSE, Light_Diffuse);
                rcLightfv(RC_LIGHT0, RC_SPECULAR, Light_Specular);
```

```
            }

            StaticSceneDraw();
        }

        return TRUE;
    }

    virtual BOOL OnDraw(void){
        static int i = 0;
        i++;  // 3 times loop with each color.
        if(IsTestMode() && i>3) return FALSE;

        {
            rcClear(RC_COLOR_BUFFER_BIT | RC_DEPTH_BUFFER_BIT);

            rcMatrixMode(RC_MODELVIEW);
            rcLoadIdentity();

            rcPushMatrix();
            rcRotatef(g_turn, 0, 1, 0);
            rcLightfv(RC_LIGHT0, RC_POSITION, Light_Position);
            rcPopMatrix();

            rcTranslatef(0, 0, -7.0);
            rcRotatef(35, 0, 1, 0);
            rcRotatef(15, 1, 0, 0);
            rcRotatef(-90, 0, 0, 1);

            rcEnableClientState(RC_VERTEX_ARRAY);
            rcEnableClientState(RC_NORMAL_ARRAY);
            rcEnableClientState(RC_TEXTURE_COORD_ARRAY);

            rcVertexPointer(3, RC_FLOAT, sizeof(struct Pos), g_EarthVertices);
            rcNormalPointer(RC_FLOAT, 0, g_EarthNormals);
            rcTexCoordPointer(2, RC_FLOAT, 0, g_EarthTexCoords);

            rcBindMaterial(MaterialIDs[Earth]);
            rcDrawArrays(RC_TRIANGLES, 0, sizeof(g_EarthVertices)/sizeof(Pos));

            rcDisableClientState(RC_VERTEX_ARRAY);
            rcDisableClientState(RC_NORMAL_ARRAY);
             rcDisableClientState(RC_TEXTURE_COORD_ARRAY);

            g_turn += 5;

            if (g_turn == 360)
                g_turn = 0;
        }

        return TRUE;
    }
};

Tutorial    g_Tutorial;
```

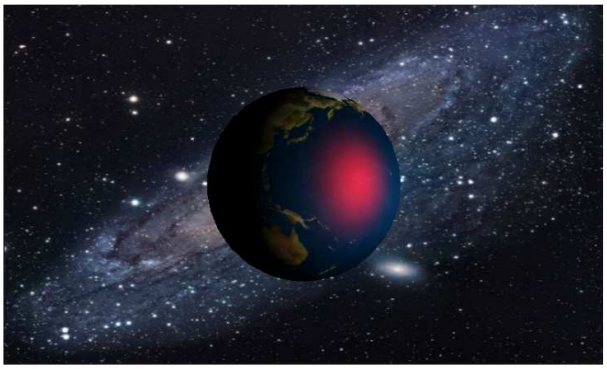

그림 35. 랜더링 결과

제 10장 부록 D.
프로그램 개발 환경 설정

1.1 리눅스 개발환경 설정

a. 라이브러리 경로 설정

ArriaV의 경우 '~/Demos/app/lib',
StratixV의 경우 '~/Demos/app/lib' 디렉토리에는 RayCore® 랜더링에 필요한 라이브러리들이 위치하고 있다. 이 라이브러리들을 심볼릭 링크 또는 /etc 폴더에서 설정(configuration)을 통해 경로를 지정한 후 컴파일 하여야 한다.

- Makefile 작성 시 제공한 SDK 가 설치된 폴더에서 라이브러리가 있는 폴더를
  (예)
  32bit linux: '~/Demos/app/lib/linux_x86',
  64bit linux: '~/Demos/app/lib/linux_x68') library path 로 설정한다.

  (예) LIBDIR := $(LIBDIR) -lstdc++ -lpthread -lm -lX11 –lRayCoreAPI -lRCDDK –lRCHAL –lRCDDIArriaV

- 제공된 makefile 로 컴파일을 하면 '~/Demos/app/lib' 디렉토리에 정적 라이브러리'RCFramework.a'가 생성된다.
- Application 컴파일 시 반드시 필요한 라이브러리들은 다음과 같다.

[리눅스에 포함된 라이브러리]
stdc++ , pthread , m, X11 (Xlibrary는 libgtk2.0-dev를 설치하도록 권장한다.)

[RayCore®에 포함된 동적 및 정적 라이브러리]

libRayCoreAPI.so, libRCDDK.so, libRCHAL.so, libRCDDIArriaV.so, RCFramework.a

b. 헤더파일 경로 설정

Makefile 작성 시, include path를 포함시키려면 아래와 같이 설정하도록 한다.

```
INC   := \
        -I$(예제파일 소스가 들어 있는 경로) \
        -I../RCFramework/src      \
        -I../../../include \
        -I../../../include/khronos    \
  -I../../../include/siliconarts
```

c. Makefile작성 예

```
## Copyright (c) | 2010 ~ 2013 Siliconarts, Inc. All Rights Reserved
## tutorials
## Date :

.SUFFIXES : .cc .o

CXX                   := $(CROSS)g++

INC       := \
        -I../RCFramework/src      \
        -I../../../include \
        -I../../../include/khronos    \
        -I../../../include/siliconarts

ifeq ($(LBITS),64)
  OS_ARCH      := linux_x64
else
  OS_ARCH      := linux_x86
endif

LIBS   := -L../../../lib/$(OS_ARCH)
LIBS       := $(LIBS) -lstdc++ -lpthread -lm -lX11 -lRayCoreAPI -lRCDDK -lRCHAL -lRCDDIArriaV
CXXFLAGS       = $(INC)
SRC_PATH              := $(PWD)

OBJS       := \
  $(SRC_PATH)/main.o

SRCS                  := \
  $(SRC_PATH)/main.cpp

TARGET       = rc_simpletriangle

all : $(TARGET)

$(TARGET) : $(OBJS)
  $(CXX) -o $@ ../../../lib/RCFramework.a $(OBJS) $(LIBS)

dep :
  gccmakedep $(INC) $(SRCS)

clean :
  rm -rf $(OBJS) $(TARGET) core

new :
```

```
$(MAKE) clean
$(MAKE)
```

1.2 EGL Native Window 설정

RayCore® 개발 키트는 리눅스를 지원하고 있기 때문에, EGL도 각 플랫폼에 따른 의존성을 가지게 된다.

a. 리눅스

리눅스에서 EGL은 X11/Xlib.h, X11/Xutil.h, X11/Xatom.h 헤더 파일을 사용한다. 리눅스의 X라이브러리의 윈도우 핸들 불러와서 창을 생성한다.

그림 36. 리눅스에서 응용프로그램 창 생성

1.3 EGL 설정

EGL은 리눅스와 같은 OS와 RayCore® API사이에서 이 둘 간의 인터페이스 기능을 지원한다. EGL은 하드웨어 디스플레이 정보를 받아 Surface를 생성하여 Context에 랜더링된 화면을 출력하는 역할을 한다. RayCore® 개발 키트를 사용하여 응용프로그램을 개발하는 경우 정적 라이브러리인 'RCFramework.a' 에서 EGL 설정을 해 주므로, 헤더파일인 'RCFramework.h'을 반드시 포함시켜서 하며, 프로그램 시작 부분에서 ***rcSceneAllInit*** 함수를 호출하여 EGL을 생성 및 초기화를 하여야 한다.

a. EGL 초기화

EGL에 대한 속성 설정(attribute conifg)는 반드시 기본 RGB는 8, 8, 8로 하며, DEPTH는 16으로 하고, EGL_RENDERABLE_TYPE은 EGL_OPEN_EG_BIT로 설정해야만 한다. EGL에 대한 속성 컨텍스트(attribute context)는 EGL_CONTEXT_CLIENT_VERSION을 반드시 1로 두어야 한다.

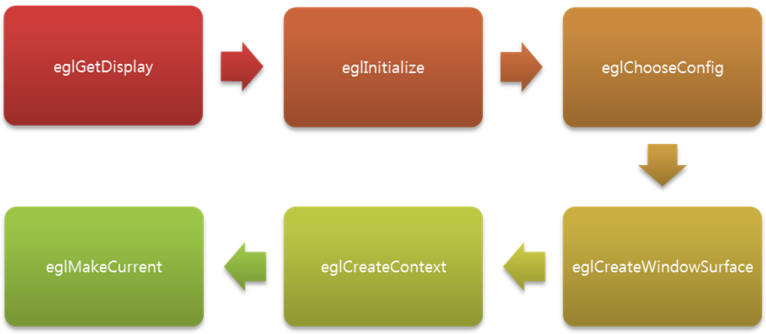

그림 37. EGL 초기화 순서

```
EGLint      attribConfig[] = {
        EGL_RED_SIZE,          8,
        EGL_GREEN_SIZE,        8,
        EGL_BLUE_SIZE,         8,
        EGL_DEPTH_SIZE,        16,
        EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
        EGL_NONE
};

EGLint      attribContext[] = {
        EGL_CONTEXT_CLIENT_VERSION, 1,
        EGL_NONE
};
```

Application 시작 시, Full Screen 모드로 실행하려면, Egl Config의 EGL_SURFACE_TYPE에서 EGL_PIXMAP_BIT를 설정하여 준다. 모든 응용 프로그램의 실행 후, 화면에서 'Alt+Enter'를 눌러 Full Screen 모드를 Toggle할 수 있다. (이 기능은 MS윈도우에서만 쓸 수 있다.)

```
EGLint      attribConfig[] = {
        EGL_RED_SIZE,          8,
        EGL_GREEN_SIZE,        8,
        EGL_BLUE_SIZE,         8,
        EGL_DEPTH_SIZE,        16,
        EGL_SURFACE_TYPE, EGL_PIXMAP_BIT,
        EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
        EGL_NONE
};
```

b. EGL 그리기

일반적으로 ***rcFinish***를 호출한 뒤 ***eglSwapBuffer***를 호출한다. ***rcFinish***가 호출되지 않은 경우 ***eglSwapBuffer***에서 내부적으로 이를 호출하기 때문에 ***eglSwapBuffer***만 호출하여도 된다.

```
rcFinish();
eglSwapBuffers(eglDisplay, eglSurface);
```

c. EGL 종료

eglMakeCurrent(eglDisplay,EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT) 를 호출하여 EGL을 종료해 준다.

1.4 RCFramework 클래스를 이용한 프로그래밍

a. 일반 설정

'RCFramework.h'를 반드시 포함시키도록 한다. RCFrameWork static library는 RayCore® API에 맞는 EGL 선언이 되어 있으므로 이 라이브러리를 반드시 이용해야 한다. (config 설정, display, surface, context value 생성 및 구현부 실행)

RayCore® API는 상태머신으로 사용자가 아래와 같은 API를 사용하여 상태를 설정해 주어야 랜더링된 화면이 출력된다.

b. 초기화

OnInitialize(void) 함수는 장면(scene)이 초기화 되었는지 체크하여 true 또는 false를 반환하는 가상함수이다. 이 안에서 RayCore® API의 초기화 설정을 하도록 한다.

```
rcSceneAllInit(); //Initialize both static and dynamic scene data

//Set the RayCore    API state variables
rcClearColor(0.0f, 0.0f, 0.0f, 1.0f);      //Set the clear color
rcViewport(0, 0, Width(), Height());    //Set the viewport by window size
rcMatrixMode(RC_PROJECTION);    //Set the projection mode
rcLoadIdentity();
rcuPerspective(60.0f, (RCfloat)Width() / (RCfloat)Height(), 0.1f, 100.0f);
rcuLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);

StaticSceneDraw();                          //Draw the static scene
```

만약 물성과 텍스처를 생성 및 사용하였다면 연결된 부분을 해제하여 주어야 한다.

```
rcBindMaterial(0);
rcBindTexture(RC_TEXTURE_2D, 0);
rcDisable(RC_TEXTURE_2D);
```

c. 장면(Scene) 데이터 그리기

정적 장면을 그리기 전과 후에는 반드시 ***rcStaticSceneBegin()***과 ***rcStaticSceneEnd()***를 설정해 주어야 한다. 정적 장면은 ***OnInitialize(void)***에서 초기에 한번만 설정해 주면, 화면 갱신시 화면에 항상 도시된다.

정적 장면으로 설정되지 않는 모든 데이터는 동적 장면으로 그려진다. ***OnDraw(void)***에서 화면 갱신 전 항상 다시 설정해 주어야 화면에 도시된다.

장면(scene)을 랜더링하기 위한 간단한 코딩 순서는 다음과 같다.

- 매트릭스 모드 설정

```
rcMatrixMode(RC_MODELVIEW);
rcLoadIdentity();
```

- 물성 연결 및 설정

```
rcBindMaterial(1);
rcMaterialfv(RC_FRONT_AND_BACK, C_AMBIENT_AND_DIFFUSE, color);
```

- 정점 그리기

```
rcEnableClientState(RC_VERTEX_ARRAY);
rcVertexPointer(3, RC_FLOAT, 0, vertices);
rcDrawArrays(RC_TRIANGLES, 0, 3);
rcDisableClientState(RC_VERTEX_ARRAY);
```

1.5 일반적인 프로그래밍 소스 코드

```
//*****************************//
//Example : Simple Reflection          //
//*****************************//
EGLDisplay m_eglDisplay;
EGLContext m_eglContext;
EGLSurface m_eglSurface;
int             m_iWidth = 800;
int             m_iHeight = 480;

#define MATERIAL_FRAME          0
#define MATERIAL_MIRROR         1
#define MATERIAL_TRIANGLE       2
unsigned int g_MaterialArray[3];

RCfloat       m_Angle=0.0f;

int main(int argc, char ** argv) {

      //*****************************//
      //Initialize EGL
      //*****************************//
      {
            EGLint       attribConfig[] = {
                  EGL_RED_SIZE,        8,
                  EGL_GREEN_SIZE,    8,
                  EGL_BLUE_SIZE,      8,
                  EGL_DEPTH_SIZE,    16,
                  EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
                  EGL_NONE
            };
            EGLint       attribContext[] = {
                  EGL_CONTEXT_CLIENT_VERSION, 1,
                  EGL_NONE
            };
            EGLConfig eglConfig;
            EGLint nConfigs = 0;

            m_eglDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf("There is an error while the function, eglGetDisplay.\n");
                  return 0;
            }
            eglInitialize(m_eglDisplay, NULL, NULL);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf("There is an error while the function, eglInitialize.\n");
                  return 0;
            }
            eglChooseConfig(m_eglDisplay, attribConfig, &eglConfig, 1, &nConfigs);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf("There is an error while the function, eglChooseConfig.\n");
                  return 0;
            }
            if(nConfigs == 0)
            {
                  printf("There is no config selected.\n");
                  return 0;
            }

            //HWND HWnd;
            //m_eglSurface = eglCreateWindowSurface(m_eglDisplay, eglConfig,
(EGLNativeWindowType)HWnd, 0);
            m_eglSurface = eglCreateWindowSurface(m_eglDisplay, eglConfig, NULL, 0);
            if(eglGetError() != EGL_SUCCESS)
            {
                  printf(   "There is an error while the function, eglCreateWindowSurface.\n");
```

```
            return 0;
        }
        m_eglContext = eglCreateContext(m_eglDisplay, eglConfig, EGL_NO_CONTEXT, attribContext);
        if(eglGetError() != EGL_SUCCESS)
        {
            printf("There is an error while the function, eglCreateContext.\n");
            return 0;
        }
        if(eglMakeCurrent(m_eglDisplay, m_eglSurface, m_eglSurface, m_eglContext) != EGL_TRUE)
        {
            printf("There is an error while the function, eglMakeCurrent.\n");
            return 0;
        }
}
//*******************************//


//*******************************//
//Set the camera
//*******************************//
{
   //Set the clear color
        rcClearColor(0.73f, 0.2f, 0.23f, 1.0f);

        //Set the viewport by window size
        rcViewport(0, 0, m_iWidth, m_iHeight);

        rcMatrixMode(RC_PROJECTION);
        rcLoadIdentity();

        rcuPerspective(60.0f, (RCfloat)m_iWidth / (RCfloat)m_iHeight, 1.0f, 500.0f);
        rcuLookAt(-3, 0, 0, 0, 0, -8, 0, 1, 0);
}
//*******************************//


//*******************************//
//Set the light
//*******************************//
{
        RCfloat ambient[] = {0.2f, 0.2f, 0.2f, 1.0f};
        RCfloat diffuse[] = {0.7f, 0.7f, 0.7f, 1.0f};
        RCfloat specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
        RCfloat pos[] = {0.0f, -0.5f, 1.0f, 1.0f};

        rcLightfv(RC_LIGHT0, RC_POSITION, pos);
        rcLightfv(RC_LIGHT0, RC_AMBIENT, ambient);
        rcLightfv(RC_LIGHT0, RC_DIFFUSE, diffuse);
        rcLightfv(RC_LIGHT0, RC_SPECULAR, specular);
}
//*******************************//


//*******************************//
//Set the material
//*******************************//
{
        RCfloat Reflectance[] = {1.0, 1.0, 1.0};
        RCfloat NotReflectance[] = {0, 0, 0};
        RCfloat color[3][4]={
                {0.2f, 0.4f, 0.8f, 0.0f},
                {0.0f, 0.8f, 0.4f, 0.0f},
                {0.0f, 1.0f, 0.0f, 0.0f},
        };

        rcGenMaterials(3, g_MaterialArray);

        rcBindMaterial(g_MaterialArray[MATERIAL_FRAME]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK,
                        RC_AMBIENT_AND_DIFFUSE,
                        &color[MATERIAL_FRAME][0]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, NotReflectance);

        rcBindMaterial(g_MaterialArray[MATERIAL_MIRROR]);
        rcDisable(RC_TEXTURE_2D);
```

```
        rcMaterialfv(RC_FRONT_AND_BACK,
                        RC_AMBIENT_AND_DIFFUSE,
                        &color[MATERIAL_MIRROR][0]);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, Reflectance);

        rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);
        rcDisable(RC_TEXTURE_2D);
        rcMaterialfv(RC_FRONT_AND_BACK,
                        RC_AMBIENT_AND_DIFFUSE,
                        &color[MATERIAL_TRIANGLE][0]);

        RCfloat ambient[] = {0.8,0.8,0.8,0};
        RCfloat diffuse[] = {0.2,0.2,0,0};
        rcMaterialfv(RC_FRONT_AND_BACK, RC_AMBIENT, ambient);
        rcMaterialfv(RC_FRONT_AND_BACK, RC_DIFFUSE, diffuse);

        rcMaterialfv(RC_FRONT_AND_BACK, RC_REFLECTION, NotReflectance);
}
//*****************************//


//*****************************//
//Initialize all scene
//*****************************//
//Initialize both static and dynamic scene data
rcSceneAllInit();
//*****************************//


//*****************************//
//Draw the static scene
//*****************************//
rcStaticSceneBegin();
{
        RCfloat QuadVertices[4][3];

        //Rectangle
        QuadVertices[0][0] = 2.0f;   QuadVertices[0][1] = -2.0f;     QuadVertices[0][2] = 0.0f;
        QuadVertices[1][0] = 2.0f;   QuadVertices[1][1] = 2.0f;      QuadVertices[1][2] = 0.0f;
        QuadVertices[2][0] = -2.0f;  QuadVertices[2][1] = 2.0f;      QuadVertices[2][2] = 0.0f;
        QuadVertices[3][0] = -2.0f;  QuadVertices[3][1] = -2.0f;     QuadVertices[3][2] = 0.0f;

        rcDisable(RC_TEXTURE_2D);

        rcEnableClientState(RC_VERTEX_ARRAY);

        //Set the modelview matrix mode
        rcMatrixMode(RC_MODELVIEW);
        rcLoadIdentity();

        //Set the vertex data
        rcVertexPointer(3, RC_FLOAT, 0, QuadVertices);


        //Bind the material of a mirror's frame
        rcBindMaterial(g_MaterialArray[MATERIAL_FRAME]);

        //Rendering the mirror's frame
        rcPushMatrix();
        rcTranslatef(0, 0, -10);
        rcScalef(2, 2, 2);
        rcDrawArrays(RC_QUADS,0,4);
        rcPopMatrix();


        //Bind the material of a mirror
        rcBindMaterial(g_MaterialArray[MATERIAL_MIRROR]);

        //Rendering the mirror
        rcPushMatrix();
        rcTranslatef(0, 0, -9.999f);
        rcScalef(1.7f, 1.7f, 1.7f);
        rcDrawArrays(RC_QUADS, 0, 4);
        rcPopMatrix();

        rcDisableClientState(RC_VERTEX_ARRAY);
```

```
    }
    rcStaticSceneEnd();
    //*****************************//


    //*****************************//
    //Draw the dynamic scene
    //*****************************//
    {
        RCfloat TriangleVertices[3][3];

        //Triangle
        TriangleVertices[0][0] = 0.0f;     TriangleVertices[0][1] = 0.707f;     TriangleVertices[0][2] =
0.0f;
        TriangleVertices[1][0] = -1.0f;    TriangleVertices[1][1] = -0.707f;    TriangleVertices[1][2] =
0.0f;
        TriangleVertices[2][0] = 1.0f;     TriangleVertices[2][1] = -0.707f;    TriangleVertices[2][2] =
0.0f;

        while(1)
        {
            rcEnableClientState(RC_VERTEX_ARRAY);

            //Set the modelview matrix mode
            rcMatrixMode(RC_MODELVIEW);
            rcLoadIdentity();

            //Set the vertex data
            rcVertexPointer(3, RC_FLOAT, 0, TriangleVertices);

            //Bind the material of a triangle
            rcBindMaterial(g_MaterialArray[MATERIAL_TRIANGLE]);

            //Rendering the triangle
            rcPushMatrix();
            rcTranslatef(0, 0, -5);
            rcRotatef(m_Angle, 0, 1, 0);
            rcDrawArrays(RC_TRIANGLES, 0, 3);
            rcPopMatrix();

            rcDisableClientState(RC_VERTEX_ARRAY);

            //Display all scene on the screen
            if(rcGetError() == RC_NO_ERROR)
            {
                rcFinish();
                eglSwapBuffers(m_eglDisplay, m_eglSurface);
            }else{
                printf("There was an error while drawing scene.\n");
                break;
            }

            //Change the rotation angle of a triangle
            m_Angle += 10.f;
        }
    }
    //*****************************//


    //*****************************//
    //Release EGL
    //*****************************//
    if(m_eglDisplay) eglMakeCurrent(m_eglDisplay,
                                    EGL_NO_SURFACE,
                                    EGL_NO_SURFACE,
                                    EGL_NO_CONTEXT);
    if(m_eglSurface != EGL_NO_SURFACE)  eglDestroySurface(m_eglDisplay, m_eglSurface);
    if(m_eglContext != EGL_NO_CONTEXT)  eglDestroyContext(m_eglDisplay, m_eglContext);
    if(m_eglDisplay != EGL_NO_DISPLAY)  eglTerminate(m_eglDisplay);
    m_eglDisplay    = EGL_NO_DISPLAY;
    m_eglContext    = EGL_NO_CONTEXT;
    m_eglSurface    = EGL_NO_SURFACE;
    //*****************************//

}
```