# Introduction

 Lab Project 3.2 에서 우리는 terrain 데모를 위한 rendering 코드를 살펴 보았다. 그러나 응용프로그램은 terrain 주변을 돌아 보기 위한 세 가지의 카메라 모드를 제공한다:first person, third person, spacecraft. 이것은 pitch, roll, yaw 를 할 수 있게 한다. 우리는 제한된 중력 시스템을 포함하여 카메라가 땅으로 떨어지지만 땅 밑으로는 가지 않도록 하였다. 그리고 간단한 마찰 모델을 적용하여 terrain 위에서 부드러운 방향이동을 가능하게 하였다. 이번 chapter에서는 카메라 시스템에 대해서 논의해 본다. 그리고 자신만의 고유한 카메라 시스템 관리를 어떻게 하는 지도 알아본다. 이 시간이 끝나면 low level 에서의 view matrix 의 작동법을 이해하게 되며, 게임에 필요한 대부분의 카메라 시스템을 생성할 수 있을 것이다.

# 4.1. The View Matrix

 Chapter 1 에서 카메라를 world 좌표계와 카메라의 local 방향과 위치에 geometry 와의 관계를 반영하도록 scene 의 geometry 를 재배치시키는 transform 의 inverse 로 생각하도록 배웠다. 이러한 방법으로 geometry 를 재배치 시키는 것은 scene 을 rendering 할 때, 만약 가상 카메라의 렌즈를 통해 그것들을 본다면 원래 world 의 원점으로부터 재배치된 상태로 rendering 되는 것이다. 이것을 가능하게 하려면, world 에 있는 모든 vertex 에 대해 카메라의 rotation 과 translation 의 반대로 적용할 필요가 있다.

 이것은 무엇이 나타나는지 생각할 때의 통찰이다. 우리는 model space 로부터 버텍스 **P** 를 취하여 행렬의 곱을 이용하여 transformation 시리즈(SRT)를 적용하여 world space 의 새로운 버텍스 **P'** 로 처리할 수 있다는 것을 알고 있다. 두 버텍스들의 관계는 다음과 같다:

이러한 대수적 관계는 역의 관계를 취할 수 있다. 이것을 P 에 대해 풀어보면:

**Mworld-1** 는 **Mworld** 의 역행렬이다. 이 행렬을 이용하여 world space 의 점 **P'** 를 곱하면 원래의 local space 의 점 **P** 를 얻을 수 있다. 그래서 우리는 **Mworld-1** 를, **Mworld**  를 적용한 P 에 대한 *undo* 효과가 나온다고 말할 수 있다. 다시 말하면 위의 다른 방정식에서 **P** 는 여전히 방정식의 한쪽에 남기 때문에 상쇄된다고 볼 수 있다.(Again this makes sense since we used one to cancel out the other in the equation above so that **P** was left alone on one side of the equation.)

 일반적으로 만약 matrix M 이 space A 좌표에서 space B 좌표로 transform 하는 것을 나타낸다면, 그것의 역행렬인 M-1 은 그 방정식의 역의 관계에 대한 transform 을 나타낸다고 한다. -- space B 의 점을 취해 space A 로 보낸다. 만약 space A 가 entity X 의 local 좌표계라면 space B 에 존재하는 어떠한 points 는 X 의 역행렬의 곱에 의해 간단하게 space A 로 다시 보낼 수 있는 것이다.

 이것은 어떠한 camera system 이라도 갖고 있는 공통적 기반이다. 우리가 scene 을 rendering 할 때, 장면이 camera 를 통해 적절히 보여지기를 바란다. 그것을 하기 위해서는 world 에 있는 모든 vertex 들을 camera 의 local space 로 transform 시켜야 한다. 만약 우리가 user 입력에 기초한 camera 를 위한 world matrix 를 만든다면, 그 matrix 는 world 에 있는 camera 가 어디에 있는지 말해주며 world 의 축에 대한 방향이 어떻게 되었는지 알 수 있다. rendering 목적으로 camera 의 local space 를 기준으로 world 상에 있는 다른 몇몇의 객체를 얻으려면 모든 world space 상의 버텍스들은 camera 의 world matrix(보통 view matrix 라고 부른다) 의 역행렬을 사용하여 곱해주면 얻을 수 있다.

 사실상 camera 의 이동 효과를 상쇄할 수 있는 다른 대안은 +Z(주어진 left-handed 좌표계에서 local space???) 보는 방향으로 world 의 origin 으로 하는 것이다. 이것은 어떠한 matrix 의 역행렬과의 곱은 identity matrix 가 되는 것을 이용하는 것이다.

identiry matrix 의 row 와 column 은 완벽하게 주요한 3D 좌표계를 설명한다. 그러므로 우리는 camera 를 이동하거나 회전하더라도 상관없다.

 virtual camera 를 생성하는 것은 일반적으로 Camera::MoveForward 와 Camera::PitchUp, Camera::Strafe 등과 같은 메서드를 만듬으로서 완성된다. camera 클래스는 view matrix 를 유지하는 역할을 갖는다.(camera 의 local space matrix) 그리고 그러한 메서드를 호출에 응하여 view matrix 를 재구성하는 역할을 갖는다. 이 클래스는 오직 view matrix 에 대한 관리자가 될 필요는 없다. 그것은 종종 projection matrix 를 관리하는 역할을 한다. 이러한 방법으로 우리는 field of view(FOV) 와 near/far clip plane 을 변경하는 메서드를 만들어 볼 수도 있다.

 코드를 작성하기 전에, Chapter 1 에서 소개된 view matrix 의 속성들에 대해 자세히 알아봐야 한다. 우리는 그러한 방법을 수행할때 역행렬이 왜 사용되는지 완벽히 이해해볼 것이다. 특히, world space 에서 view space 로 vertex 들을 transform 하기 위해서 virtual camera 에서 view matrix 의 column 에 right, up, look vector 들을 저장하는지 알아볼 것이다.-- world matrix 에서는 row 에 그것들을 저장한다.

 Chapter 1 에서 표준 world matrix 는 object 의 local 좌표계의 방향과 world 좌표계의 원점으로 부터의 위치를 담고 있다.:

**World Matrix**

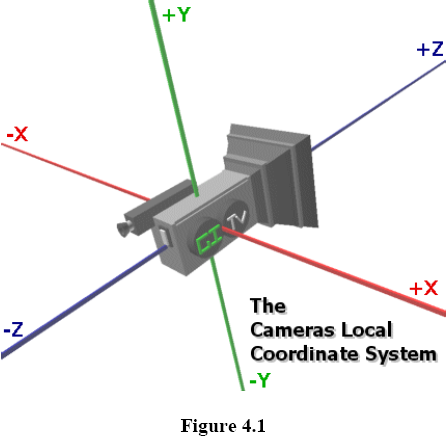
|  |  |  |  |
| --- | --- | --- | --- |
| Right Vector.x | Right Vector.y | Right Vector.z | 0 |
| Up Vector.x | Up Vector.z | Up Vector.z | 0 |
| Look Vector.x | Look Vector.y | Look Vector.z | 0 |
| Position.x | Position.y | Position.z | 1 |

다음 표에 보여지는 것은 view matrix 이다. 그것은 camera 의 local 좌표계를 invert 한 vector들로 기술되고 4번째 row 에 있는 vector는 camera 위치 base로 한 inverse translation 으로 되어있다. 이 translation은 world 의 원점을 공유하며 camera 의 위치를 반영하는 위치로 vertex들을 이동시킬 것이다.

**View Matrix**

|  |  |  |  |
| --- | --- | --- | --- |
| Right Vector.x | Up Vector.x | Look Vector.x | 0 |
| Right Vector.y | Up Vector.y | Look Vector.y | 0 |
| Right Vector.z | Up Vector.z | Look Vector.z | 0 |
| -(Position**·**Right Vector) | -(Position**·**Up Vector) | -(Position**·**Look Vector) | 1 |

Matrix 의 첫 번째 column 에 저장되어 있는 Right Vector 는 camera local 좌표계 상의 X 축 방향을 기술한다. 두 번째 column 에 저장된 Up Vector 는 Y 축, 세 번째 column 의 Look Vector 는 Z 축 방향을 각각 기술한다.(Fig 4.1)



View matrix 상의 vector들은 우리가 잠깐 논의했던 camera 의 local 좌표계의 축을 따르는 상대 위치적 정보를 기술한다. 만약 view matrix 를 invert 했다면 world 에 있는 camera 의 위치와 방향이 기술된 camera 의 world matrix 를 얻을 것이다. (Fig 4.2)

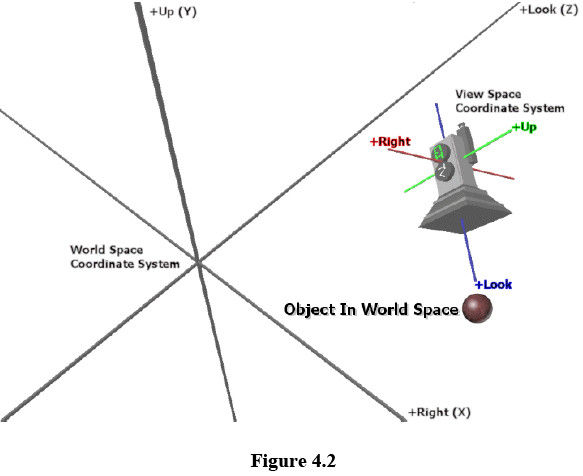
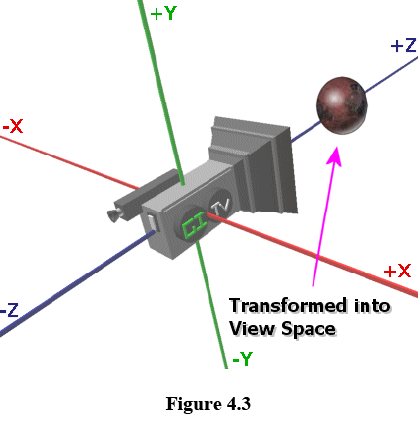


Fig 4.2 는 view matrix 의 inverse 를 사용하여 어떻게 mesh 가 그려지는지 보여준다. View matrix 가 이미 invert 되어있기 때문에 다시 invert 하면 표준 world matrix 가 얻어진다. 만약 mesh object 와 같이 camera 를 그리고 싶다면 그 matrix 를 사용하면 된다.(view matrix 의 inverse)

우리는 sphere를 그리기 위해서 camera 의 local 좌표계를 기준으로 맞춰야 되는 것을 알고 있다. 논의 했던 것과 같이 object A 를 다른 object B 의 local 좌표로 옮기려면 모든 A의 vertex들을 B의 world matrix의 inverse matrix를 곱하면 된다. 이번 case 에서 B는 camera 이기 때문에, 우리는 camera 의 world matrix 의 inverse matrix가 될 것이다. 이 inversion이 처리되면 camera 의 world matrix 는 곧 view matrix 가 된다. Fig 4.3 은 sphere 가 view space 로 transform 되었을 때를 보여준다. Camera 가 system 의 원점이고 sphere 는 world space 에 있던 것처럼 여전히 camera의 바로 앞에 있다는 것을 주목하라. Camera 의 local 좌표계로 옮겨지기 전의 상태와 완벽하게 동일한 관계를 유지하고 있다.



# 4.1.1 Vectors, Matrices, and Planes Revisited

왜 vector 와 matrix 를 곱하여 virtual space 로부터 vector를 transform 하는지에 대해 이해하기 위해서 vector 와 matrix, 그것들에 대해 생각해 볼 수 있는 다른 방법(이 시점에서 할 수 없는)에 대해 논의 해봐야 한다. 이러한 목적을 위해 우리는 일반적으로 적용할 수 있는 position vector를 언급할 것이다.

우리는 model space, world space, view space 등과 같은 여러 space들의 논의를 통해 어느 정도까지는 단일 수학적 space에 대한 처리를 하였다. 이러한 space 에서 좌표계(여기서는 left-handed)를 사용하여 위치를 지정할 수 있다.

# 4.3.5 Third Person Cameras

Third Person Camera System은 Tomb Raider™, Splinter Cell™, Mario64™ 등 많은 게임에 사용되었다. Third Person Camera 는