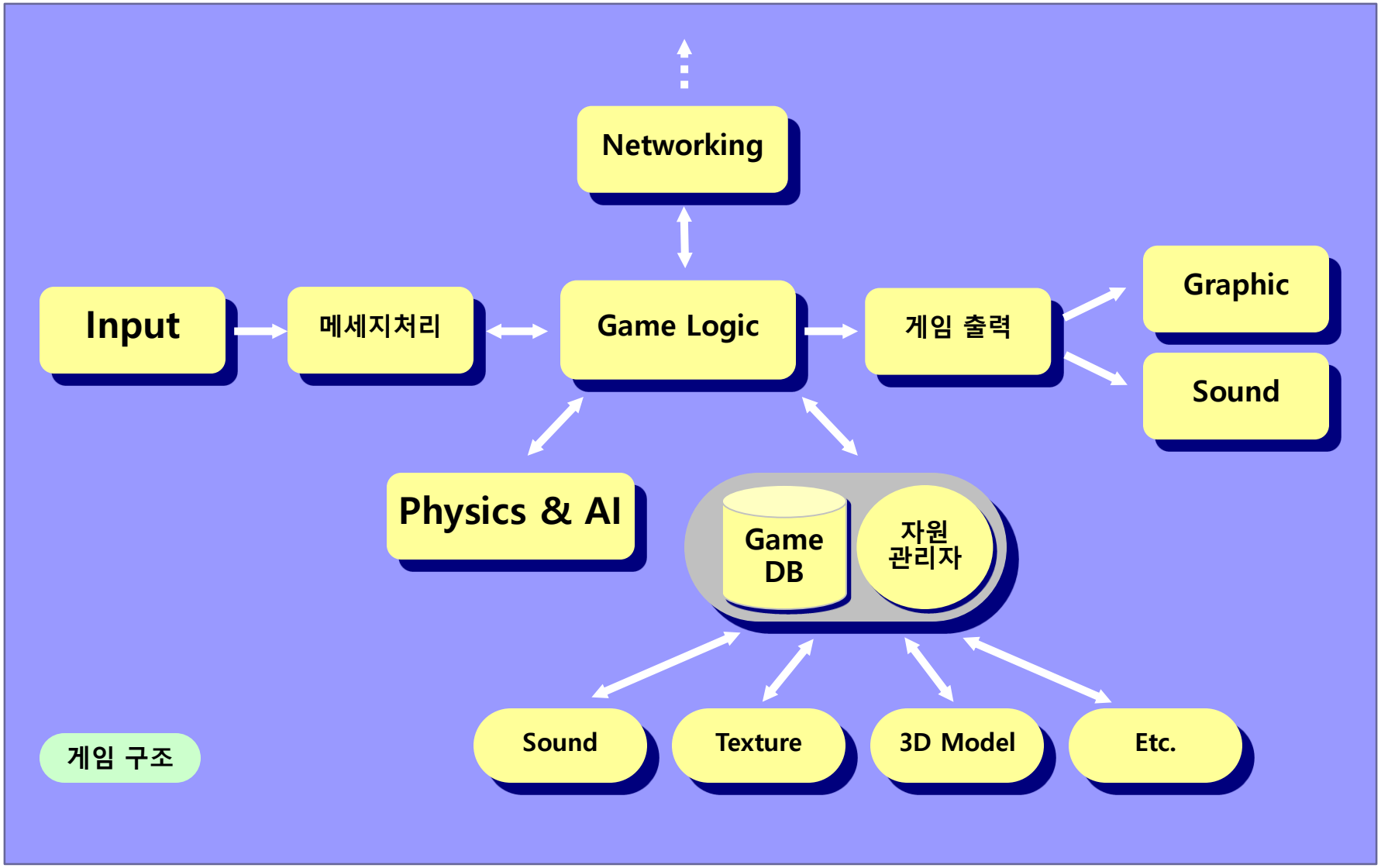
게임엔진

강의 정리

전형적인 게임 구조



게임 엔진이란?

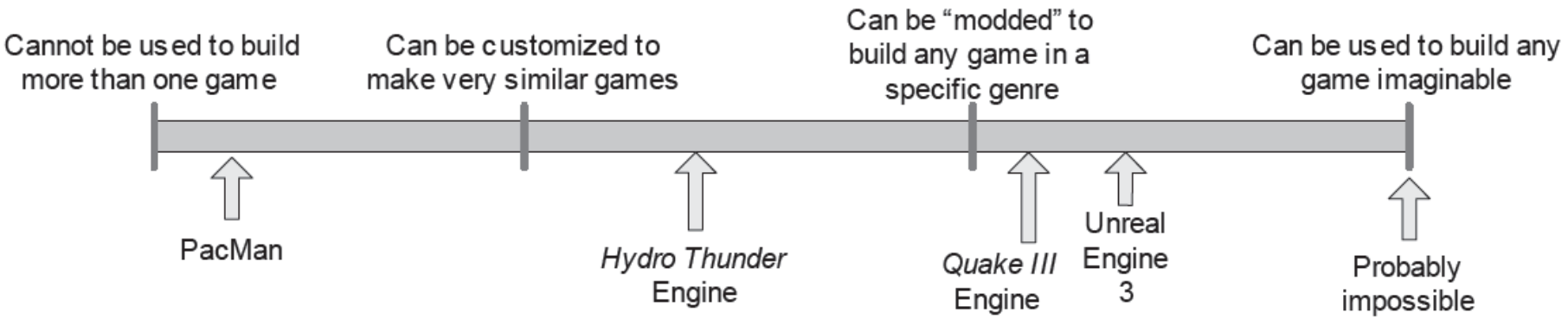
■ 게임 실행에 필요한 기능들을 효과적으로 개발할 수 있도록 설계된 소프트웨어 라이브러리 및 툴들의 집합

- □ 미들웨어 라이브러리: 렌더링, 물리, AI, 사운드, 및 네트워크 처리
- □ 툴: 맵 에디터, 씬 그래프 에디터 등등.

게임 엔진의 재사용성

- 게임과 엔진의 경계를 명확히 나누는 것은 쉽지 않음.

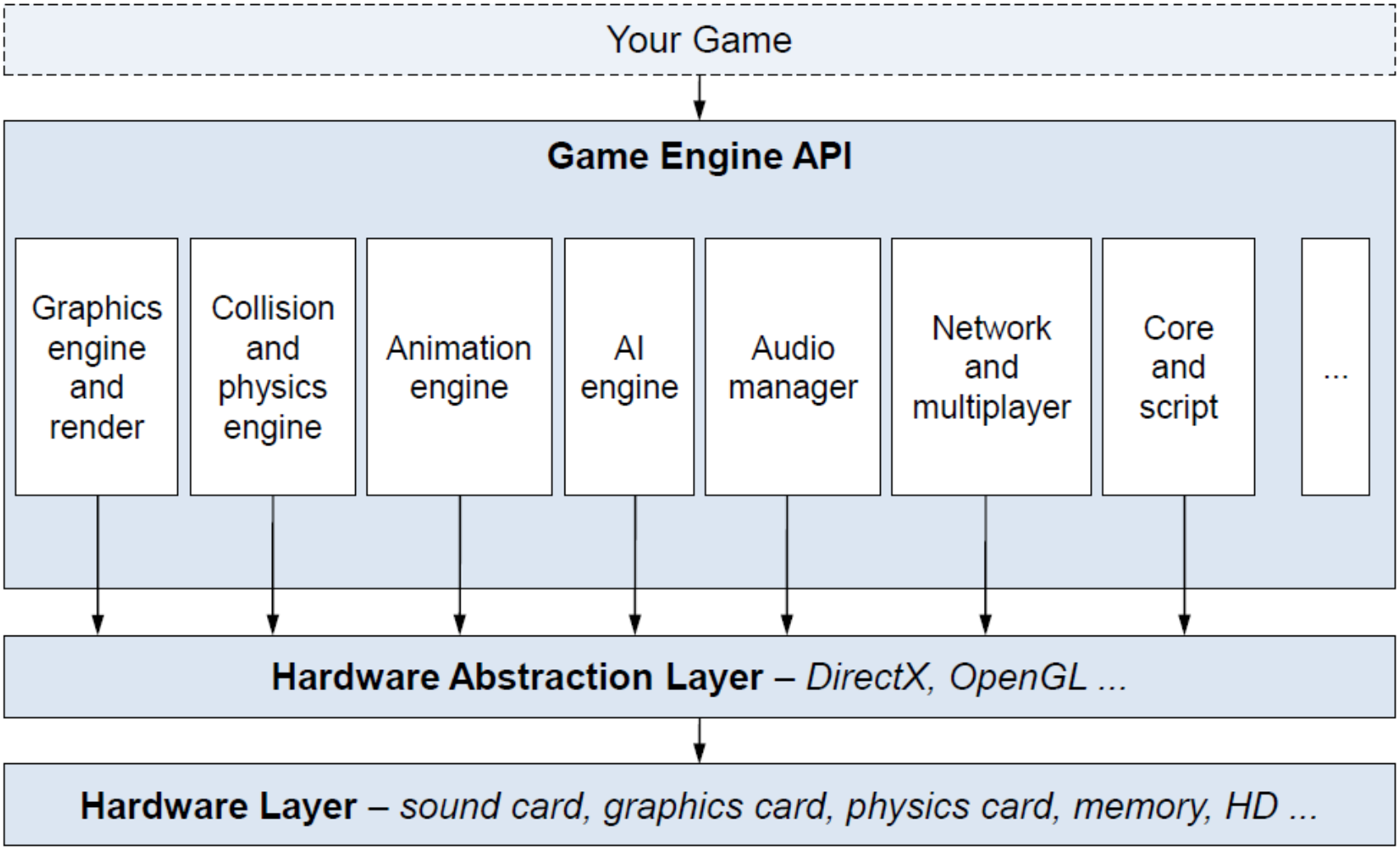
- 모든 장르에 다 적용되는 엔진을 개발하는 것은 사실상 불가능한 일…



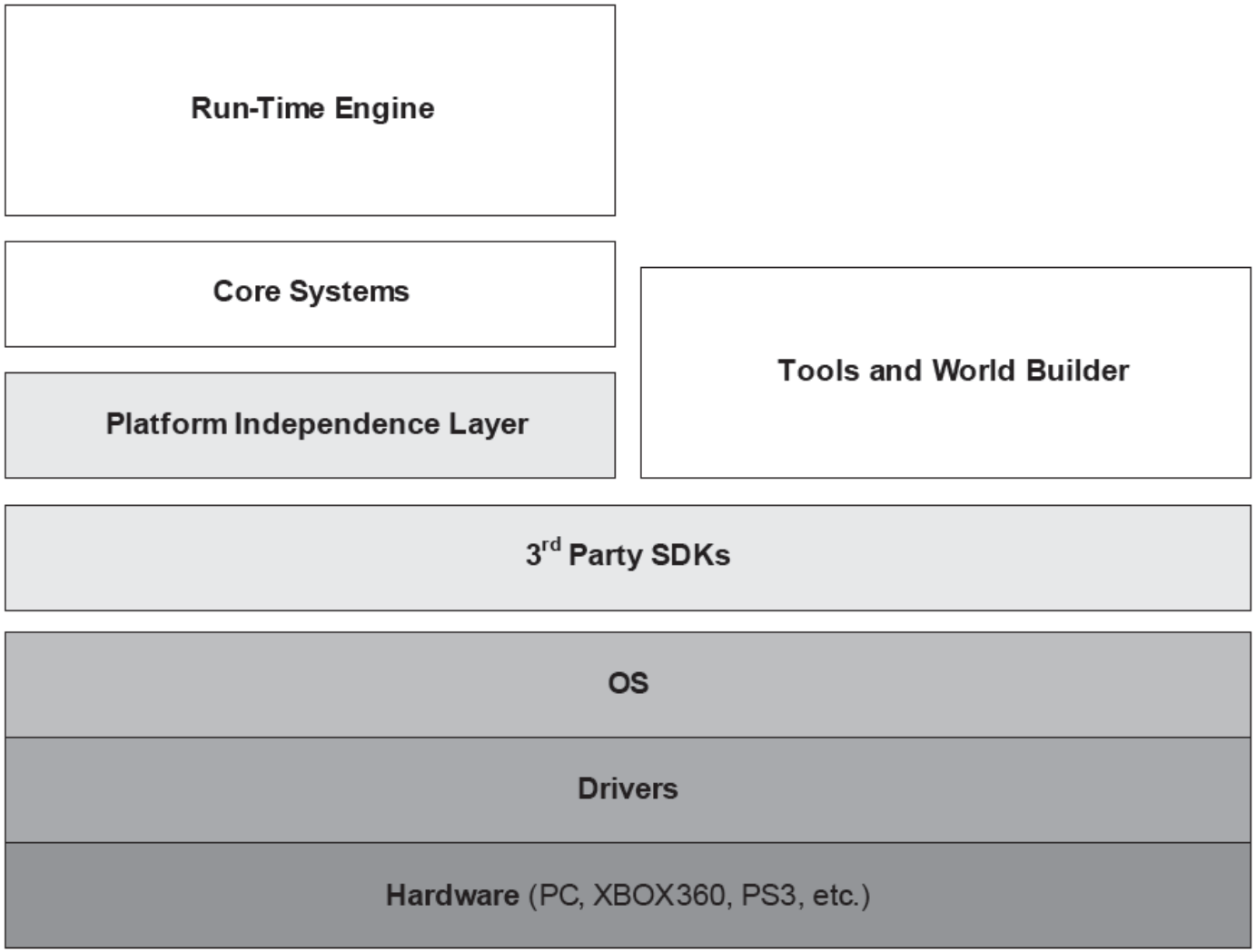
게임 장르별로 요구되는 기술들은?

장르	주요 기술
FPS	실내외 배경의 효과적인 렌더링 NPC의 매우 사실적인 애니메이션과 인공지능 빠른 응답속도(온라인의 경우)
플랫포머 & TPS	게임 사물(밧줄, 사다리, 발판 등) 들의 사실적인 움직임 메인 캐릭터 전신의 사실적 애니메이션 시야 보장을 위한 카메라 충돌 시스템
격투게임	다양하고 사실적인 격투 애니메이션 DB 정확한 사용자 입력 처리
레이싱게임	트랙 구성을 위한 효과적인 자료 구조 정교한 물리 시뮬레이션 및 포스피드백(Force feedback) 물체 변형
실시간전략 시뮬레이션	대규모 객체 처리 동적 환경 구축 정교한 전략 전술 인공 지능
MMOG	대규모 접속 네트워크 분산 처리 실시간 네트워크 지연 최적화 대규모 게임 월드 분할 처리

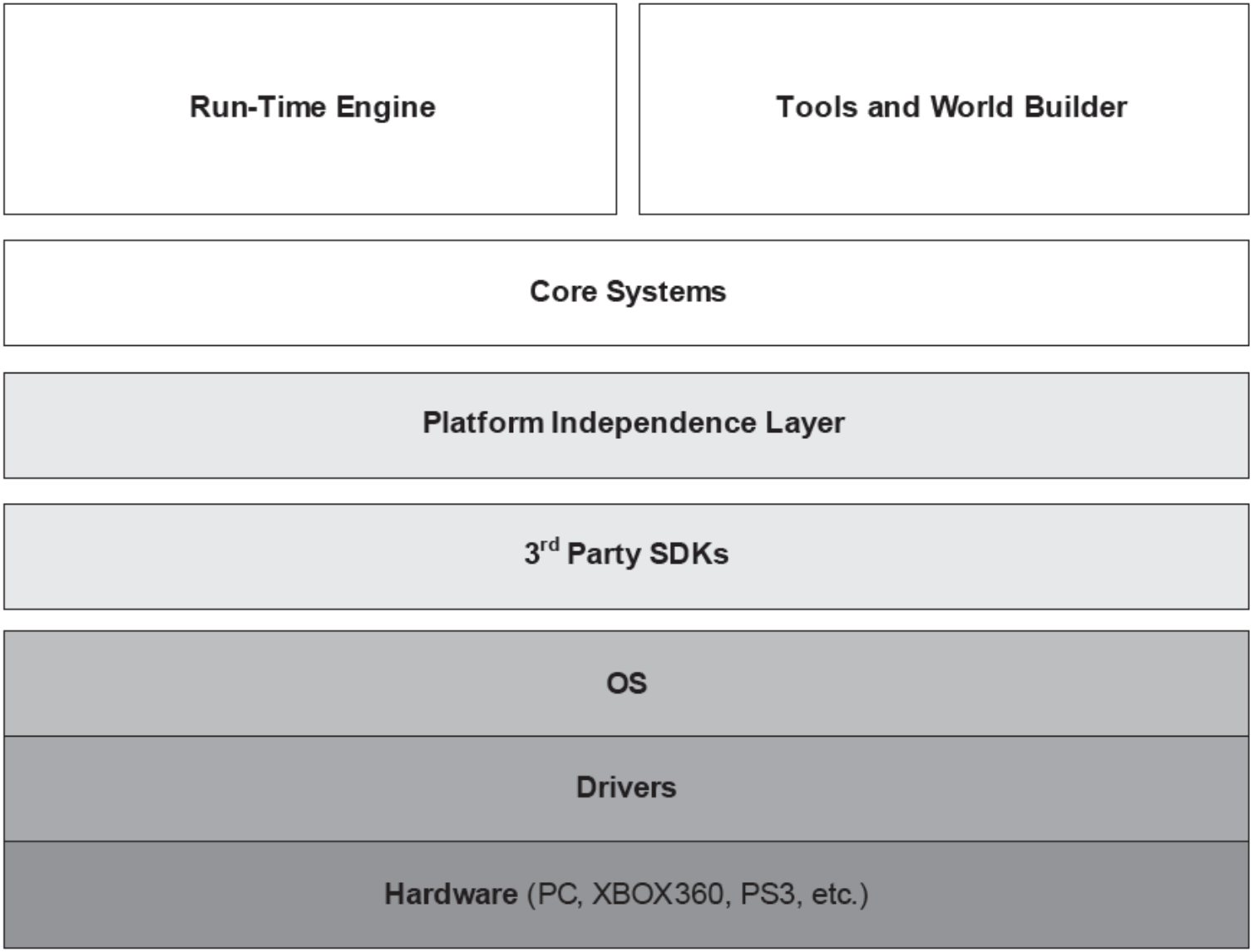
런타임 게임 엔진 구조



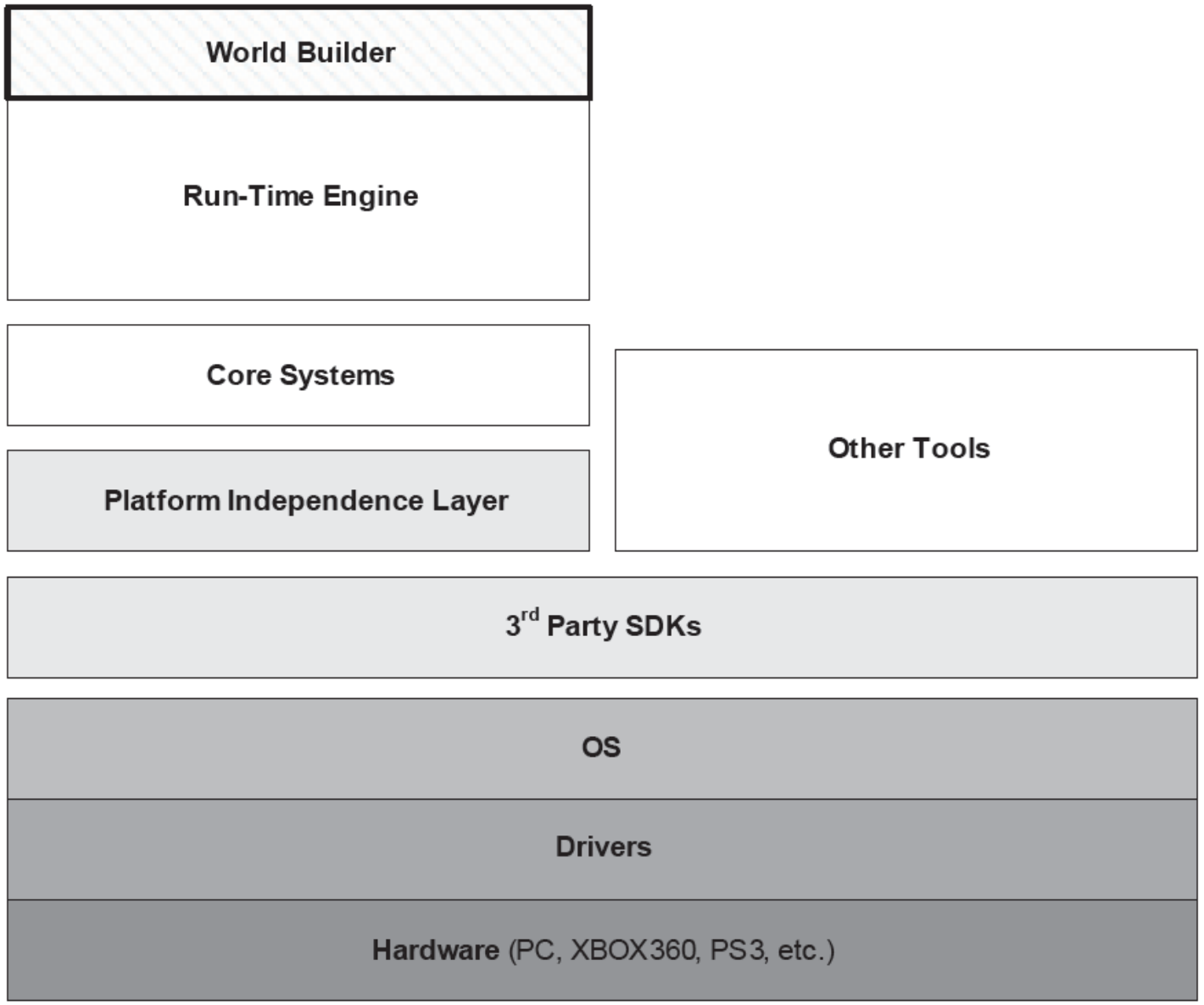
툴 독립 구조



도구 역시 프레임워크를 공유하는 구조

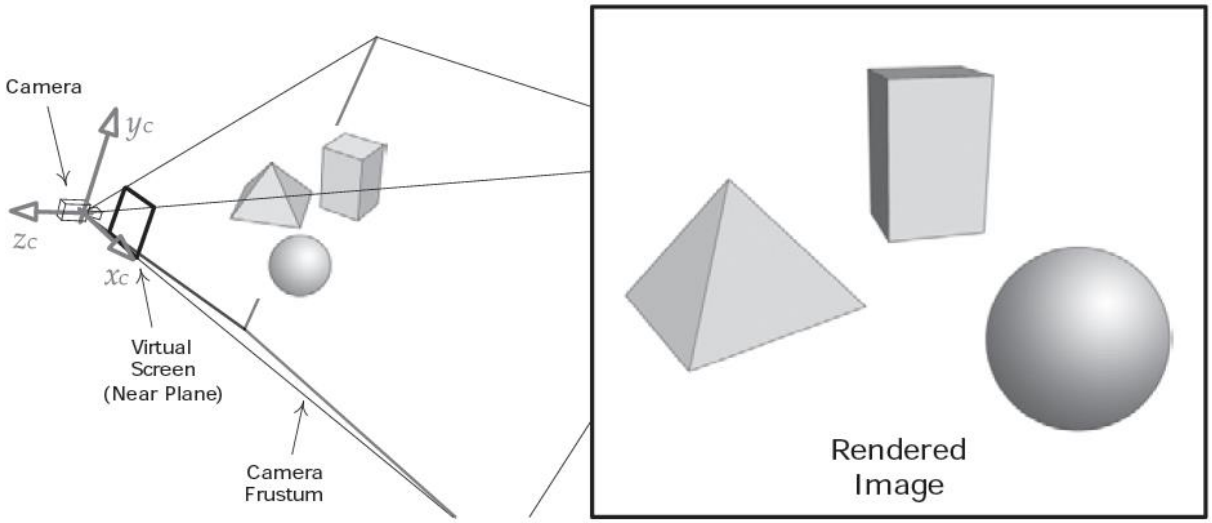


런타임 엔진과 에디터의 통합 구조 - UnreadEd, Unity3D



3D 씬(Scene) 렌더링 프로세스 개관

- 3차원 표면으로 표시된 가상의 씬 구성
- 가상의 카메라로 씬을 바라보게 만듬
- 다양한 광원들이 정의됨
- 씬 내에 존재하는 표면들의 시각적인 속성이 계산/결정



씬의 구성

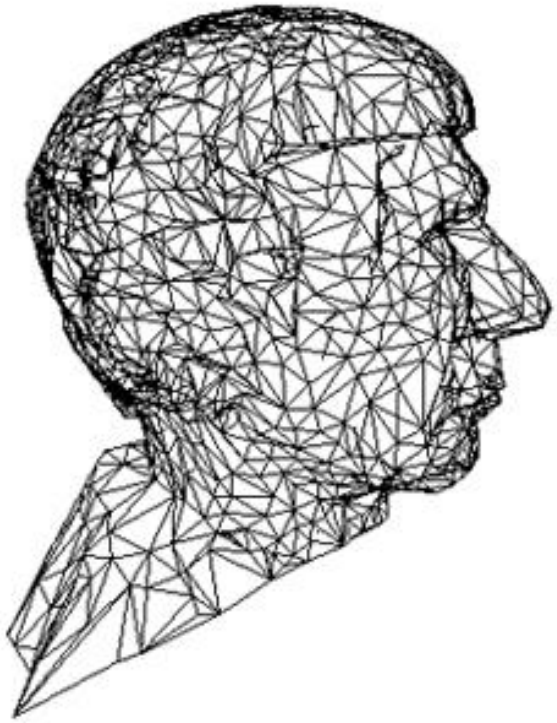
- **실제 세계의 씬은 다양한 물체들로 구성**
  - 물체는 3D 공간에서 부피를 차지
  - 단단한 물체(벽돌) vs. 형상이없는물체(물, 연기)
  - 불투명, 투명, 반투명

- **컴퓨터그래픽에서 대부분의 물체는 “단단한 물체”로 가정.**

3D 모델의 표현: 폴리곤과 메쉬

- **폴리곤(Polygon)**
  - 3개의 점이 모이면 하나의 면(face)을 만들 수 있다. 이렇게 3개의 점으로 만들어진 삼각형을 폴리곤이라 한다.

- **메쉬(Mesh)**
  - 폴리곤들이 모여서 하나의 3차원 물체를 만들게 되는데 이것을 메쉬라고 부른다. 다시 말해 메쉬는 폴리곤이 모여서 만들어진 3차원 공간의 객체(object)다.



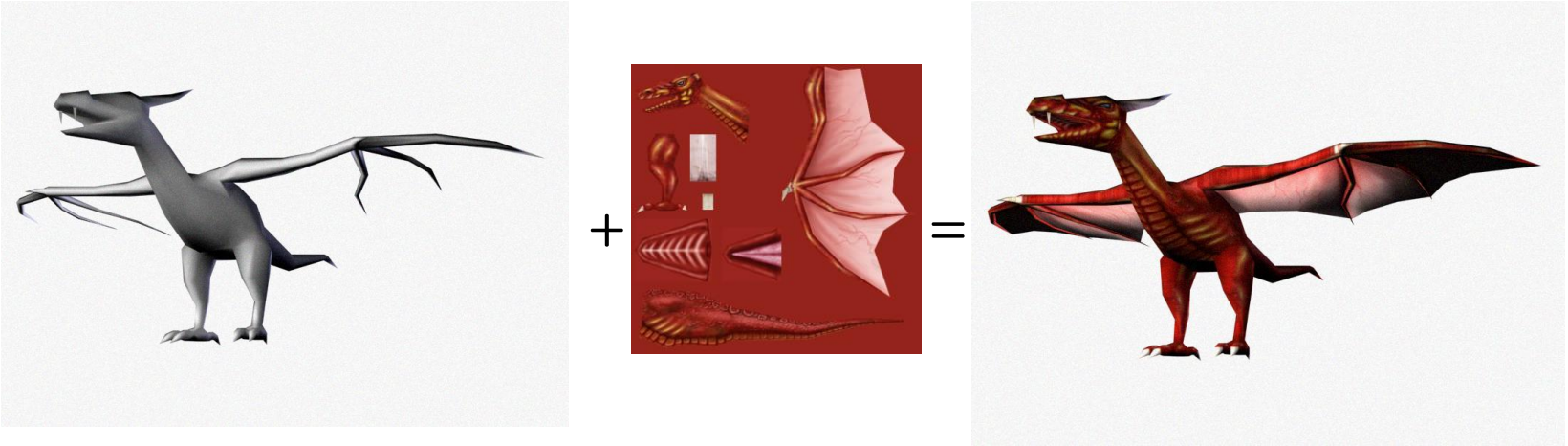
3D 모델의 표현: 텍스처 매핑

텍스처

- 폴리곤만으로 화면에 물체를 나타내는 것은 제약을 많이 갖게 된다. 3차원 물체에 2차원의 이미지(jpg, bmp, gif 등)를 입혀서 다양한 표현을 가능하게 해준다.
- 텍스처는 표면위의 점의 색상 계산을 오프라인으로 미리 계산해놓은 것이다. → 일일이 계산하는 것보다 훨씬 속도가 빠르다(근사 최적화).

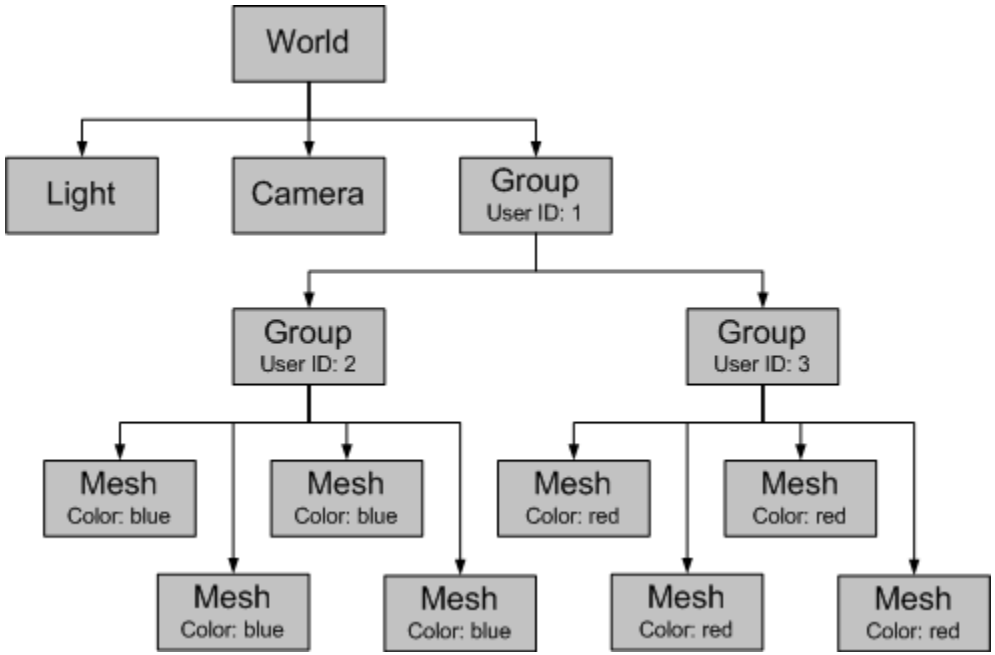
텍스처 매핑

- 3차원 물체에 텍스처를 입히는 것을 텍스처 매핑이라 하고, 매핑되는 2차원 이미지를 텍스처, 혹은 텍스처 맵이라고 부른다.



씬 그래프(Scene Graph)

- **오우거 엔진의 씬 구성은 씬 그래프의 구조를 만드는 과정임.**
- **씬 그래프**
  - 그래픽 씬을 나타내기 위한 표현 방식으로써, 일반적으로 공간적 정보를 담아 논리적으로 표현할 수 있는 구조를 지님.
  - 씬 노드(Scene Node)들의 트리(Tree)
  - 씬 노드는 엔터티(메쉬모델)를 담고 있으며, 수학적으로는 Local Axis의 역할을 하는 변환 행렬임
    .



Sample World Scene Graph
(Source: http://www.ibm.com/developerworks/java/library/wi-mobile2/index.html)

시간 측정

- **게임은 시간에 따른 가상 세계의 시뮬레이션**

- **따라서, 정확한 시간의 측정이 매우 중요함**

- **함수 time()**
  - 표준 C 라이브러리 함수
  - 1970년 1월 1일을 기준으로 현재까지 경과 시간을 “초” 단위로 알려줌
  - 1초는 컴퓨터 게임에서 너무나 긴 시간

- **모든 CPU에 포함된 정밀타이머(high-resolution timer)**
  - CPU 가 켜진 시점으로부터 경과한 클럭틱(tick)의 개수를 저장
  - 3GHz → 초당 3억번 클럭틱이 진행
  - 타이머의 정밀도: 1/30 억 = 0.333ns

- **정밀타이머 이용 윈도우 API 함수**
  - QueryPerformanceCounter() - 경과된 클럭 수
  - QueryPerformanceFrequency() - 현재 CPU의 성능 주파수(ex. 3GHz)

프레임레이트(Frame Rate)와 시간 델타(Time Delta)

프레임

- 특정 순간에 화면에 그려지는 하나의 그림

프레임 레이트(Frame Rate)

- 3D 화면을 연속적으로 얼마나 빨리 보여주는가?
- 단위: Hz, FPS(Frame Per Sec) - 초당 몇 개의 프레임을 보여주는가?
- 영상물: 24 FPS
- 북미, 일본의 게임물: 30FPS, 60FPS
- 유럽: 50FPS

시간 델타(Time Delta)

- 두 프레임 사이에 경과한 시간
- 한장의 프레임을 그리는 데 걸리는 시간
- 프레임 시간(Frame Time)
- 씬의 구성요소 밀도에 따라 시간이 달라지는 것이 문제임.

$$\Delta t = \frac{1}{FPS}$$

time delta가 중요한 이유

- **객체 위치 계산의 핵심 요소**
  - x : 객체의 위치
  - v : 객체의 속도(등속 운동 가정)

$$x_{\text{다음프레임}} = x_{\text{현재프레임}} + v\Delta t$$

초창기의 CPU 종속적 게임

- delta time 개념이 없음.
- 그냥 물체의 움직임을 pixel 값의 변화로 표시
- 문제점은?
  - CPU 성능에 따라, 물체의 움직이는 속도가 달라짐.
  - single player 게임에서는 문제가 아닐수도…

delta time의 측정

- **기본적인 측정 방법**
  - 프레임 그리기 전, 후의 CPU 정밀 타이머의 값의 차이를 측정하면 됨.

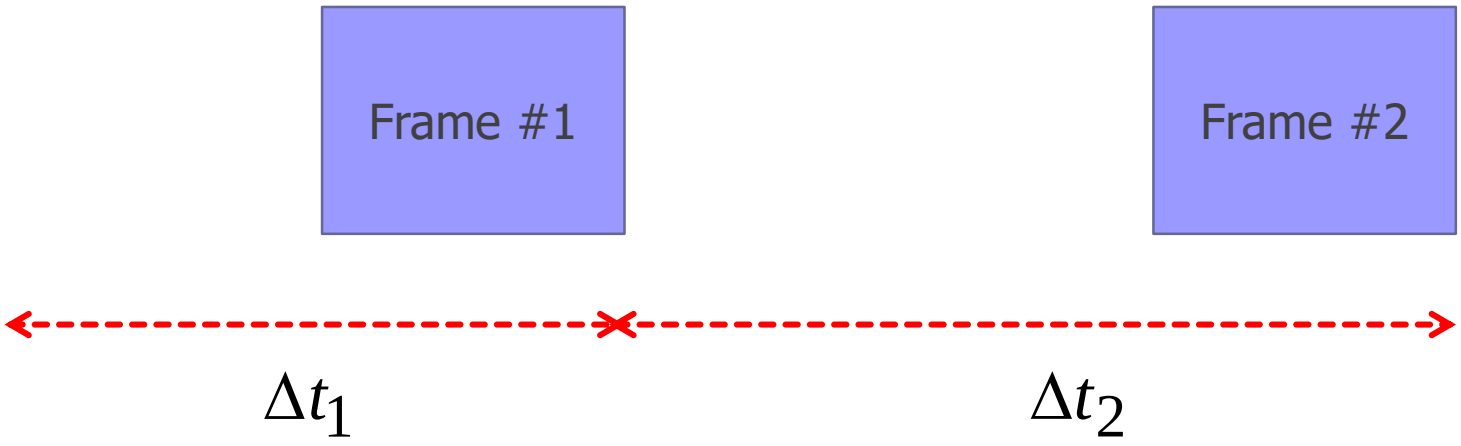
- **문제점은?**
  - 앞선 프레임의 delta time을 다음 프레임에서 객체의 위치 계산에 사용함. 미래의 delta time을 예측해서 사용하는 것임.
  - 따라서, 두개의 delta time의 차이가 난다면, 오차가 발생할 수 밖에 없음.
  - 이동평균을 이용하여 계산하는 방법도 괜찮음.

$x_1 = x_0 + v\Delta t_0$

$x_1 = x_0 + v\Delta t_1$

$x_2 = x_1 + v\Delta t_1$

$x_2 = x_1 + v\Delta t_2$



프레임 레이트의 조절

- **delta time을 계산, 예측하지 말고, 아예 고정!!(예를 들어, 30FPS=33.3333ms)**
  - 측정된 시간이 목표시간 보다 짧으면?
    - 쉰다..
  - 측정된 시간이 목표시간 보다 길면?
    - 프레임을 포기한다. 프레임을 그리지 않는다. Frame Skip→뚝뚝 끊겨보이게 됨.
  - 평균적인 frame rate가 목표 frame rate가 비슷할 경우에 제대로 작동함.

- **프레임 레이트를 고정했을 경우의 장점**
  - 물리 엔진은 일정 간격으로 업데이트를 해야 안정성있는 최적의 성능을 발휘.
  - 화면 티어링(tearing)의 방지
  - 녹화 및 재생 기능의 안정성 - 디버깅의 주요 도구
    - 게임 플레이 도중 발생하는 모든 이벤트의 시각을 기록하게 됨.
    - 프레임 레이트가 일정하지 않으면 정확한 순서대로 진행되지 않을 수 있음.

가상 타임 라인(Abstract Timeline)

- **시스템에서 사용되는 여러 개의 시간축**

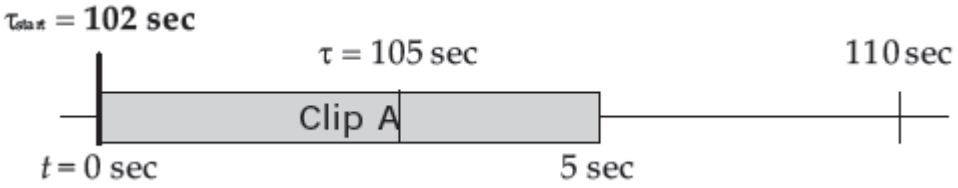
- **로컬 타임 라인**
  - 특정 시스템, 또는 객체들이 갖는 타임라인
  - 예)애니메이션 클립 타임라인, 오디오 클립 타임라인

- **실제 시간 타임 라인**
  - 기준 타임 라인

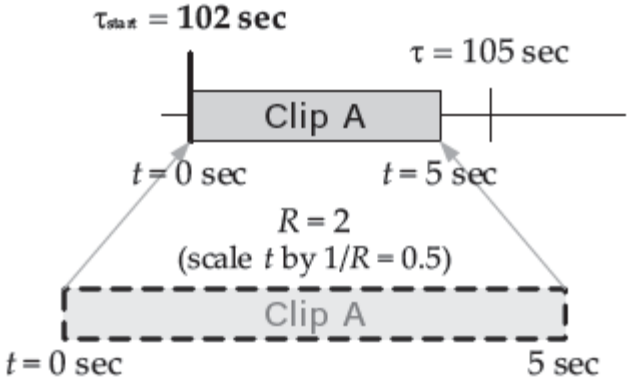
- **게임 시간 타임 라인**
  - 실제의 시간(실시간 타임라인)은 계속 진행되지만, 게임 시간(게임 타임 라인)은 중간에 멈출 수도 있고, 느리게 움직일 수도 있음.
  - 게임 시간을 멈추든지 또는 느리게 진행시키면서, 렌더링 엔진과 카메라는 다른 타임라인을 이용해 정상적으로 움직이면 효과적인 디버깅이 가능해짐.

타임라인 매핑

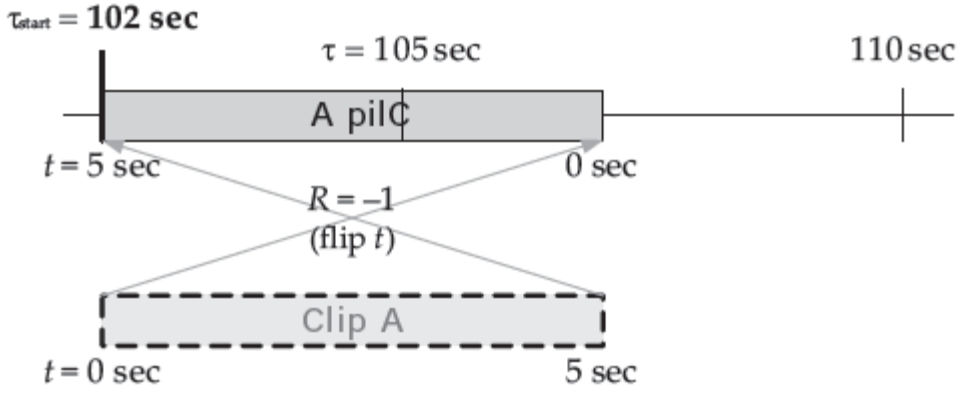
- 애니메이션을 다양한 방법으로 수행 가능



일반적인 애니메이션



2배 빠른 애니메이션



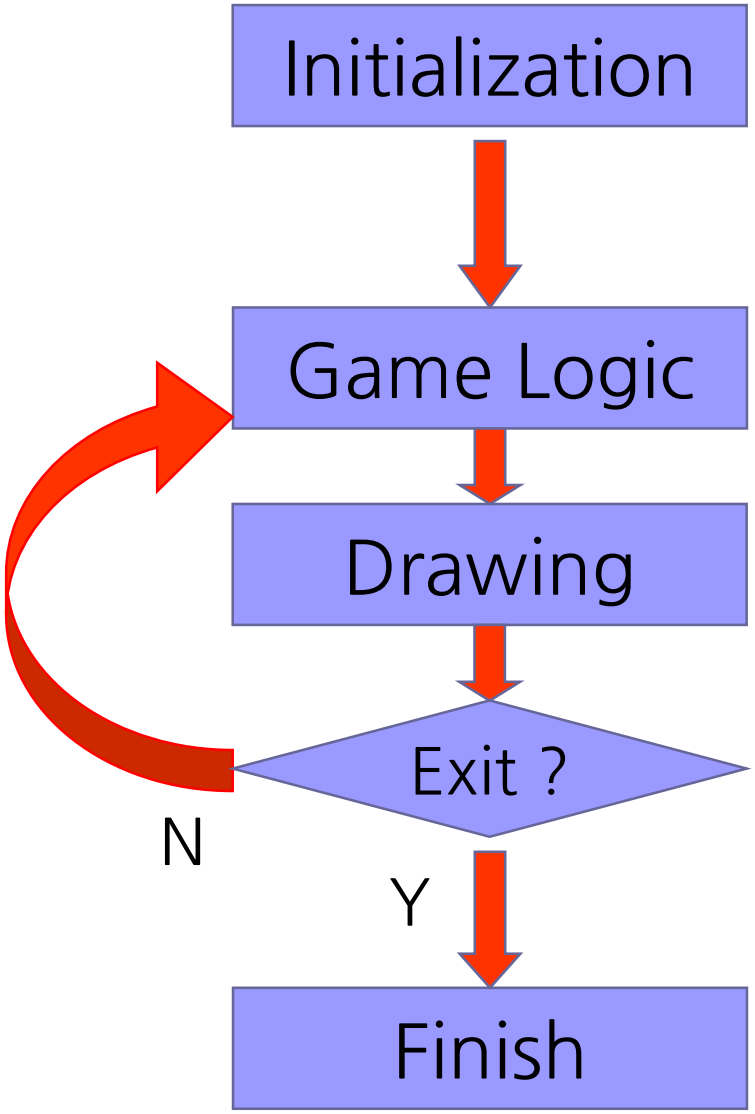
거꾸로 재생 애니메이션

게임 루프

- **게임의 하부 시스템**
  - 장치 I/O, 렌더링, 애니메이션, 충돌 감지 및 처리, 강체 물리 시뮬레이션, 멀티플레이어 네트워크

- **하부 시스템들은 주기적인 갱신(update)이 필요**
  - 시간의 진행에 따른 하부 시스템들의 변화를 반영해야 함.
  - 시스템 별로 갱신 주기가 다름.
  - 애니메이션: 30,60Hz
  - 물리시뮬레이션: 120Hz
  - AI : 초당 한두번 정도?

일반적인 기본 게임 루프



게임 루프 구조

■ 이벤트 기반 업데이트

□ 이벤트 - 게임의 상태에 변화가 생기는 것(ex. 조이패드 버튼 누르기, 폭발, 적 캐릭터가 플레이어 캐릭터를 발견한 순간)

□ 이벤트 핸들링
- 이벤트가 발생하면, 연관있는 객체들에게 알려줌(Nonification)
- 객체들은 이벤트에 대한 처리(handling)를 하게 됨.

□ 게임 객체들의 상태 갱신에 주로 사용됨.

□ 객체다마 갱신 주기가 다를 경우, 이벤트 기반 업데이트 방식이 효과적임.

□ 렌더링을 이벤트 핸들링으로 처리하려면?
- 주기적으로 (예를 들어, 1/60 초마다) 타이머 이벤트를 발생하여, 렌더링 시스템에 전달함.

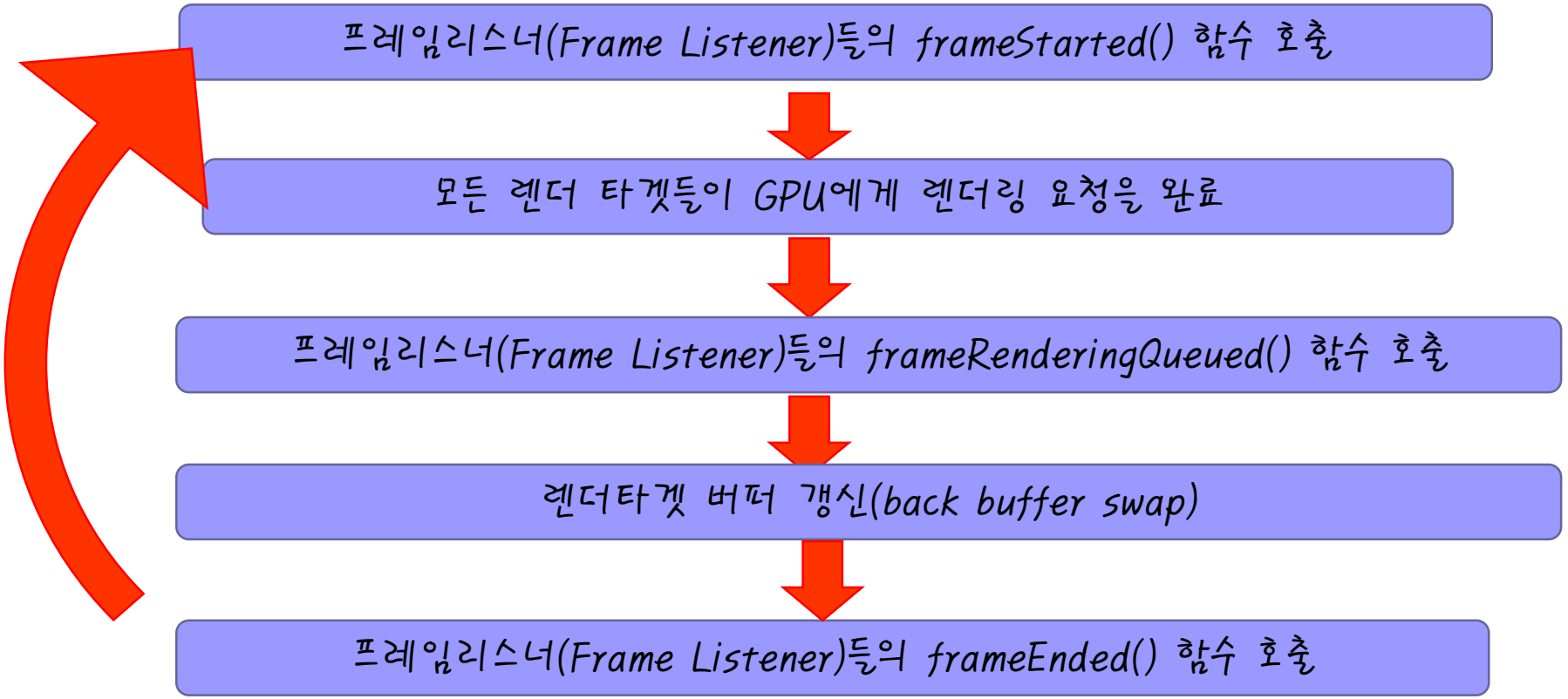
콜백 주도 프레임워크

- **프레임워크**
  - 부분적으로 구성된 “실행가능한” 애플리케이션
  - 프레임워크의 기본기능을 교체하거나, 새로운 기능을 추가할 수 있는 구조
  - 콜백 함수
    - 프레임워크가 호출하는 함수
    - 개발자가 원하는 기능을 함수로 구현해서 덧붙이는 구조.
    - 예) Ogre3D 엔진의 FrameListener 구조

오우거 엔진의 메인 렌더링 루프

- Root::startRendering() 함수에서 이루어짐.

- 메인 루프 수행 내용



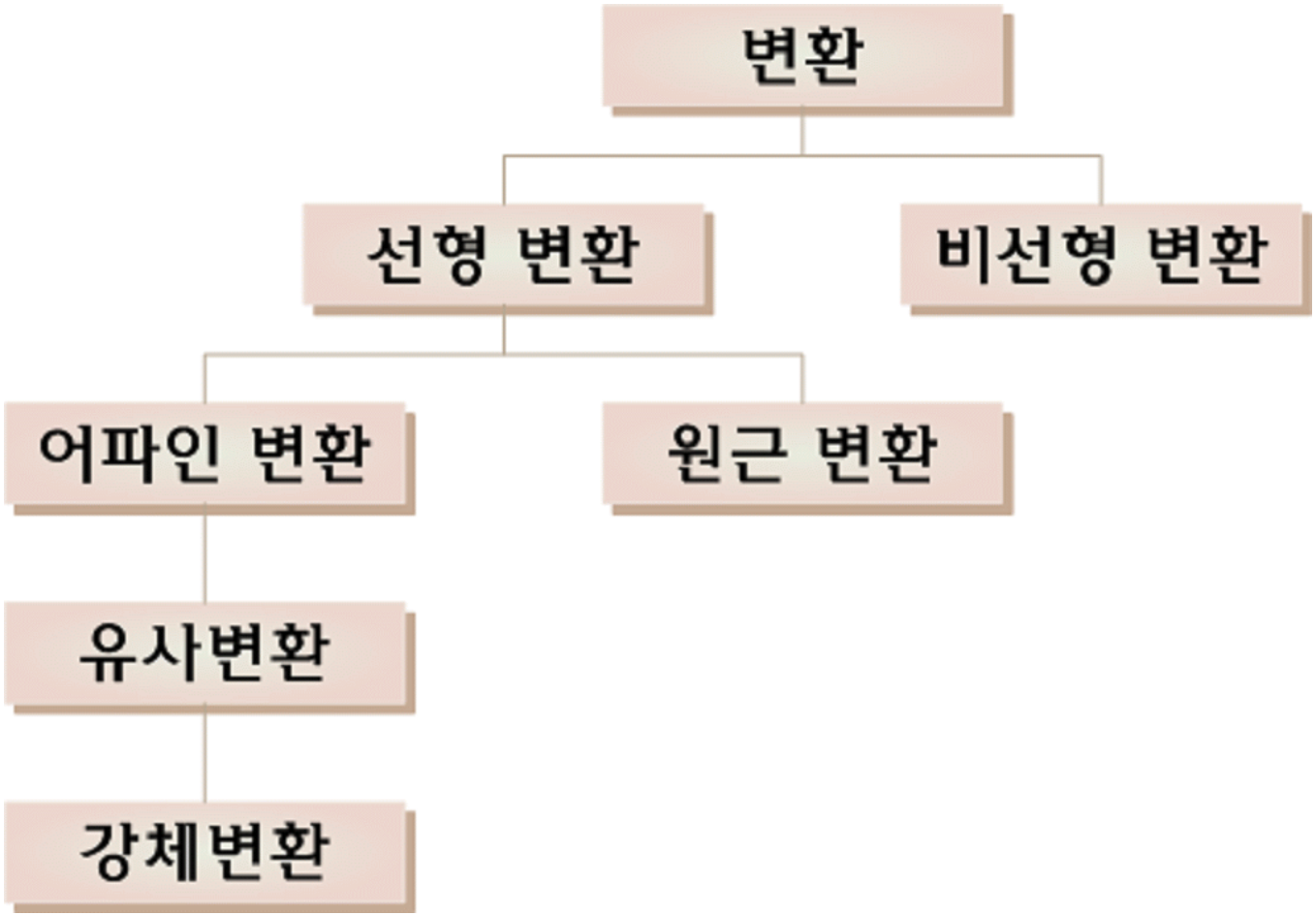
- 루프의 중단
  - frameStarted(), frameEnded(), frameRenderingQueued() 에서 하나라도 false 리턴.

기하 변환(Geometry Transformation)

- 물체 변환 또는 좌표계 변환의 기본
- 이동, 회전, 크기조절 등
- 행렬을 이용한 계산이 주로 활용(선형 변환)

$$(x, y, z) \rightarrow (x', y', z')$$

변환의 분류



■ 강체변환(Rigid Body Transformation)

- □ 이동변환, 회전변환
- □ 물체 자체의 모습은 불변

■ 유사변환(Similarity Transformation)

- □ 강체변환 + 균등 크기조절 변환, 반사변환
- □ 물체면 사이의 각이 유지됨.
- □ 물체내부 정점간의 거리가 일정한 비율로 유지됨

■ 어파인변환(Affine Transformation)

- □ 유사변환 + 차등 크기조절 변환, 전단변환
- □ 물체의 타입이 유지
  - 직선은 직선으로, 다각형은 다각형으로, 곡면은 곡면으로
  - 평행선이 보존
  - 변환행렬의 마지막 행이 항상 (0, 0, 0, 1)

■ 원근변환(Perspective Transformation)

- □ 평행선이 만남.
- □ 직선이 직선으로 유지
- □ 변환행렬의 마지막 행이 (0, 0, 0, 1) 아님.

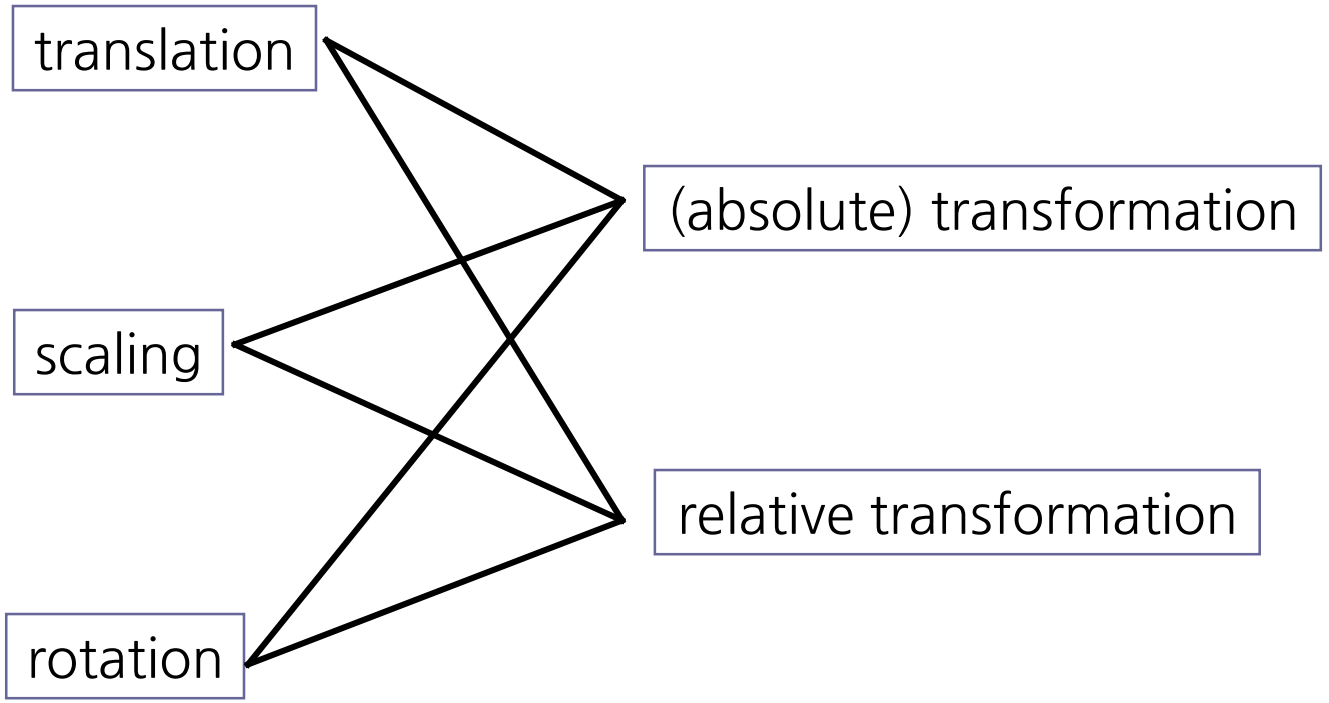
■ 선형변환(Linear Transformation)

- □ 어파인 변환 + 원근 변환
- □ 선형 조합(Linear Combination)으로 표시되는 변환
- □ x' = ax + by + cz에서 x'는 x, y, z 라는 변수를 각각 상수 배 한 것을 더한 것이다.

Absolute VS. Relative Transformation

- **절대 변환**
  - 최종값을 지정
  - 원점을 기준으로 하면, 원점과의 차이값

- **상대변환**
  - 차이값을 지정

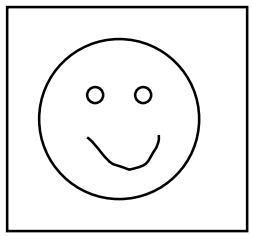


애니메이션의 방법

- **키프레임 애니메이션(Key-frame animation)**
  - 수동으로 애니메이션을 구성
- **모션 캡쳐(Motion capture)**
  - 애니메이션을 측정하여 기록함.
- **절차적 애니메이션(Procedural Animation)**
  - 자동으로 애니메이션을 생성.
  - 일종의 시뮬레이션

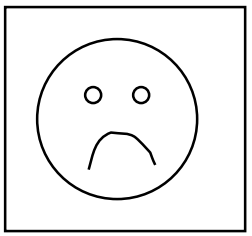
키프레임(Key Frame)과 보간(Interpolation)

- 움직임의 특징이 되는 키 프레임을 지정(사용자가 수동으로)
- 보간 알고리즘을 이용하여 키 프레임 사이의 중간 프레임들을 생성해냄.



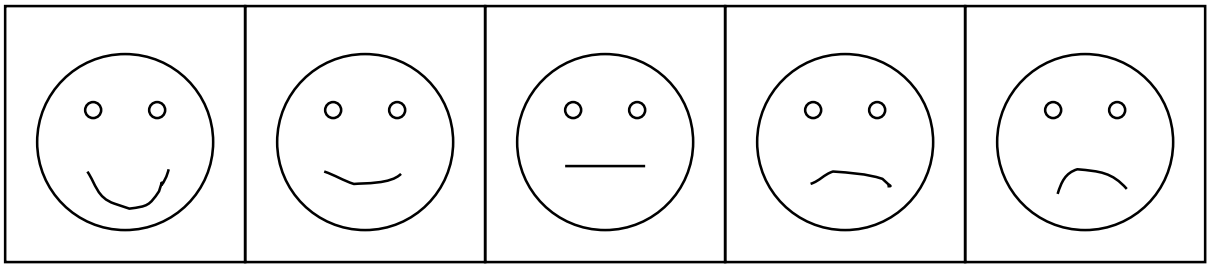
Key Frame #1

0.5 초



Key Frame #2

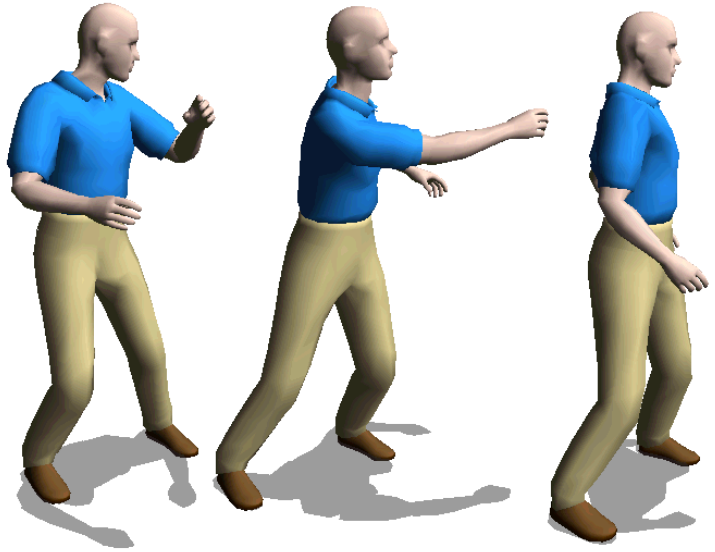
1.5초

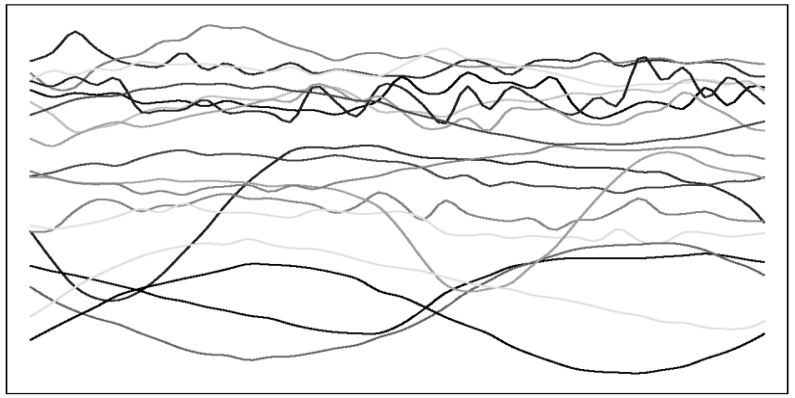


Interpolation

키프레임 애니메이션의 특징

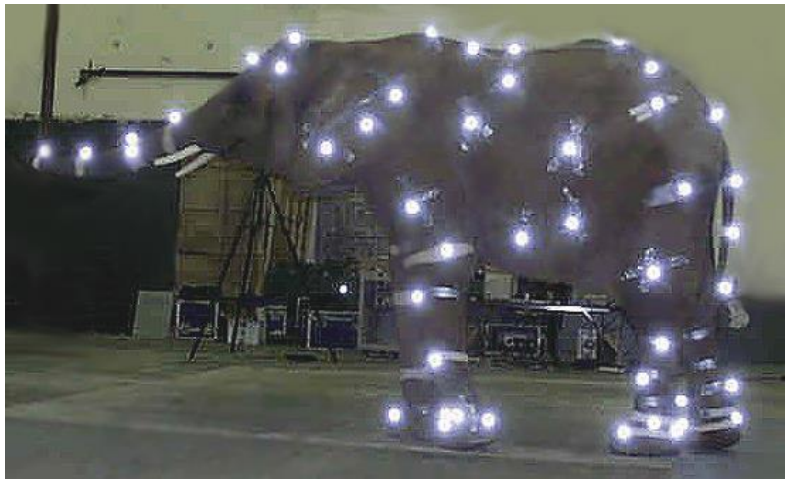
- 숙련된 애니메이터가 필요함.
- 수작업으로 키프레임을 만들어야 하기 때문에, 시간과 비용이 많이 필요.
- 고품질 / 고비용





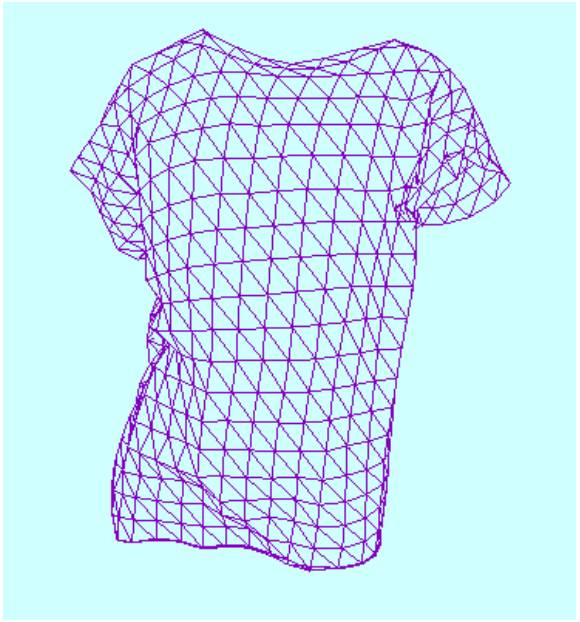
모션캡쳐

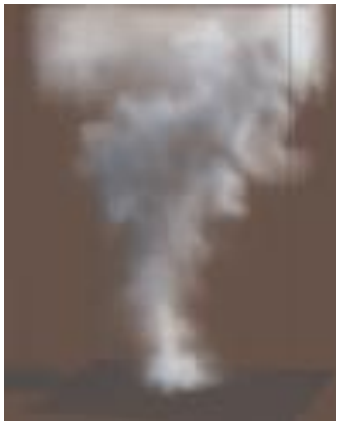
- 사람의 몸에 센서 또는 마커를 부착하여, 오브젝트의 움직임을 동작 데이터로써 그대로 측정하고 기록하는 방식.
  - 광학식 및 기계식
- 기록된 데이터를 가공하는데 긴 시간이 걸림.
- 적절한 품질 / 적절한 비용.



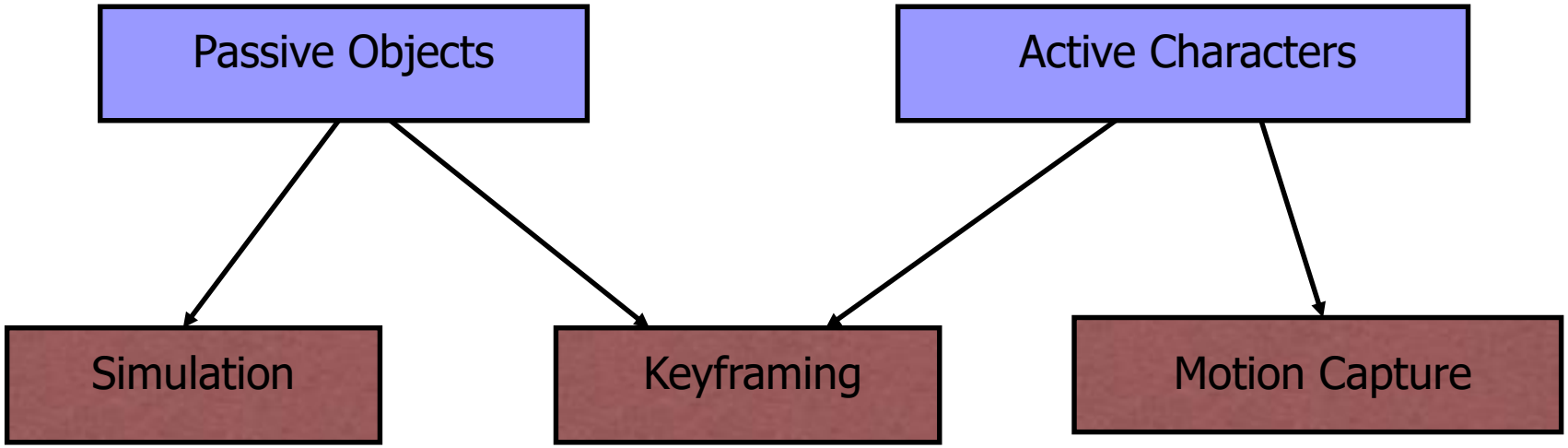
절차적 애니메이션(Procedural Animation)

- 수학적인 시뮬레이션을 이용하여 물체의 움직임을 자동으로 생성하는 방식.
- 입자 시스템(Particle system)이나 유동 표면(Flexible surface)과 같은 자연 현상이나 사람이 직접 제어하기에는 복잡한 시스템의 애니메이션 구현에 유리함.
  - 유체, 안개, 연기, 구름, 옷 표면
- 실시간 구현을 위해서는 고성능의 하드웨어 필요.



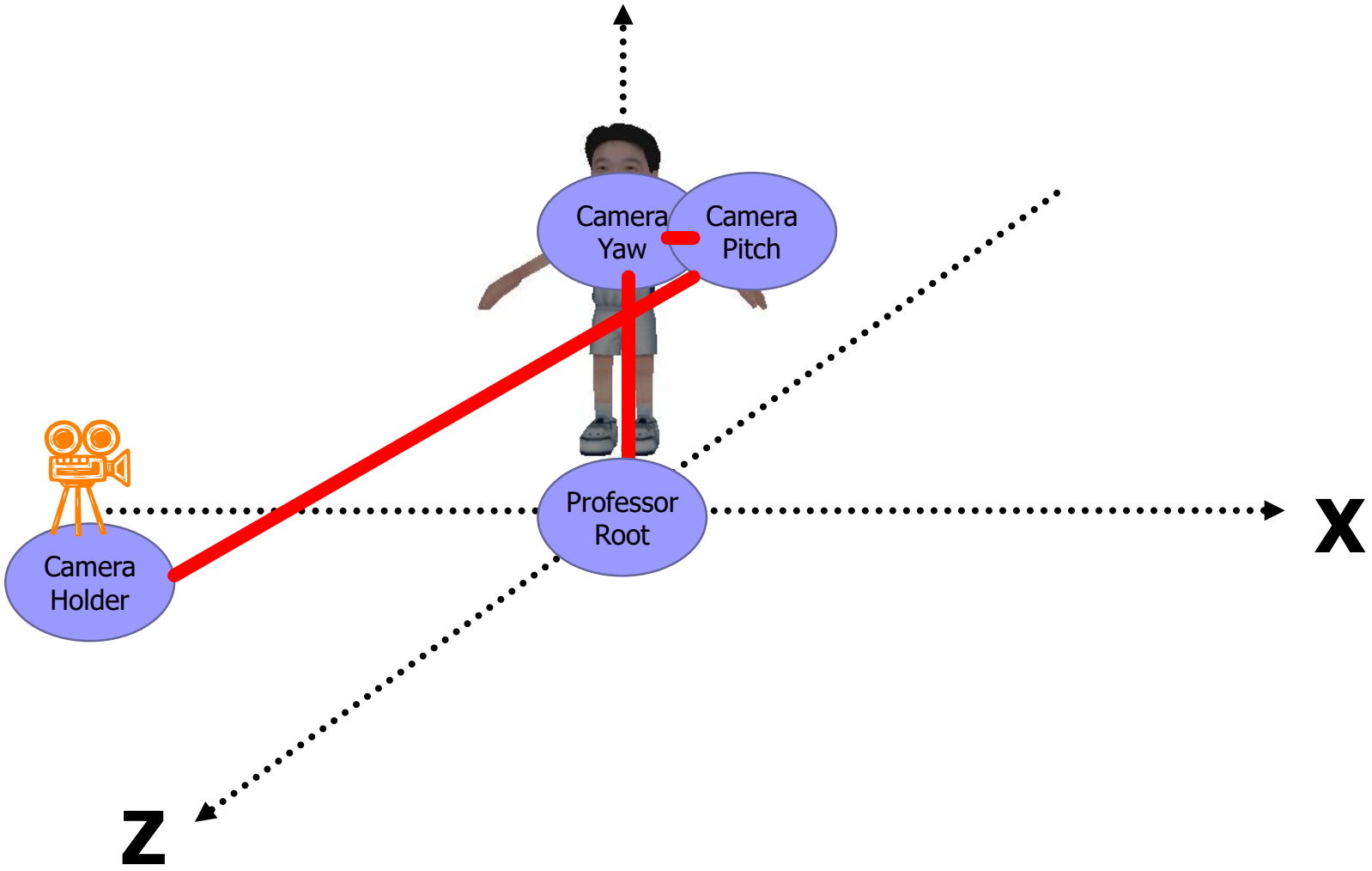


애니메이션 방법의 선택

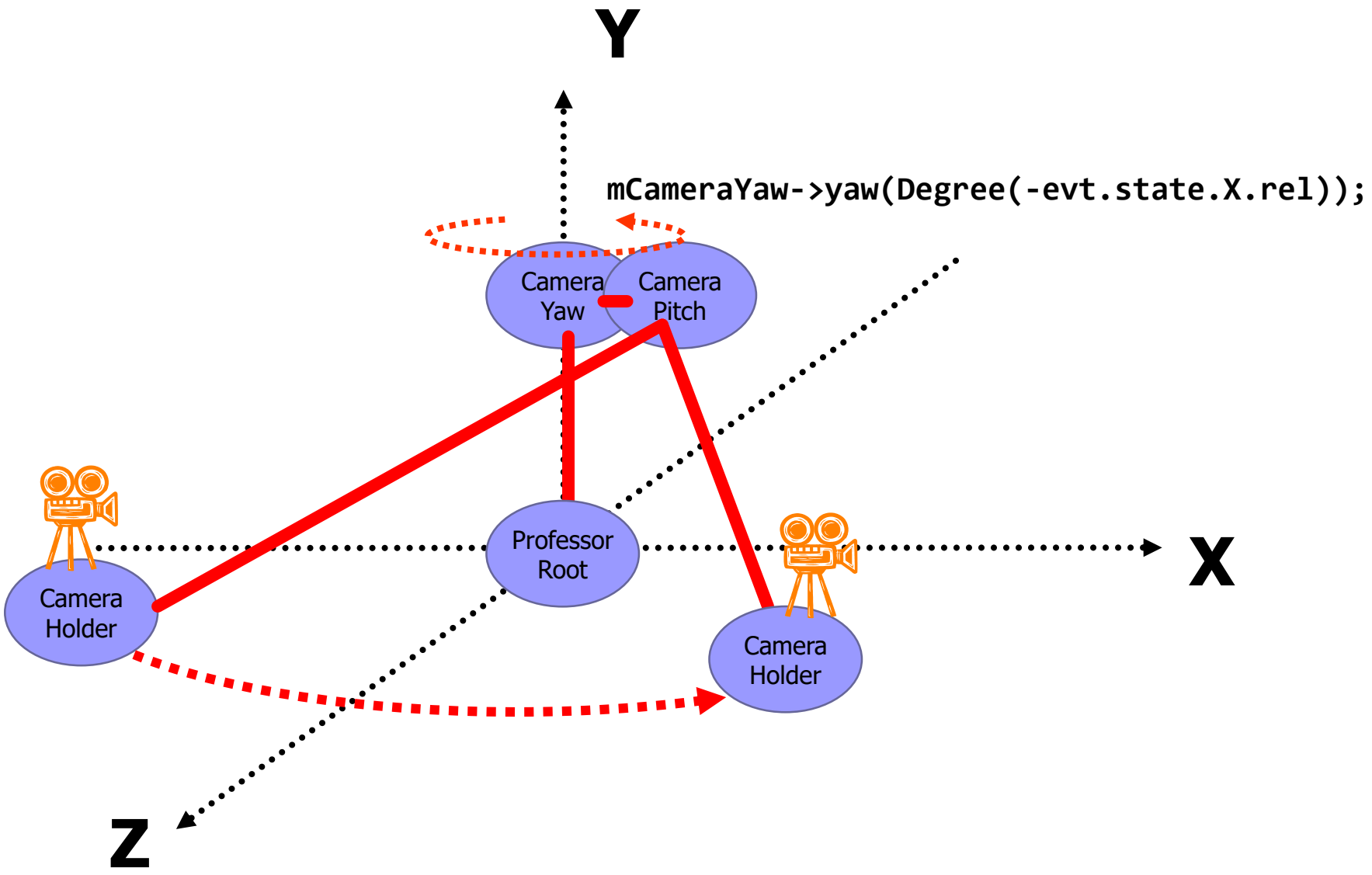


카메라 컨트롤 및 홀더 노드 설정

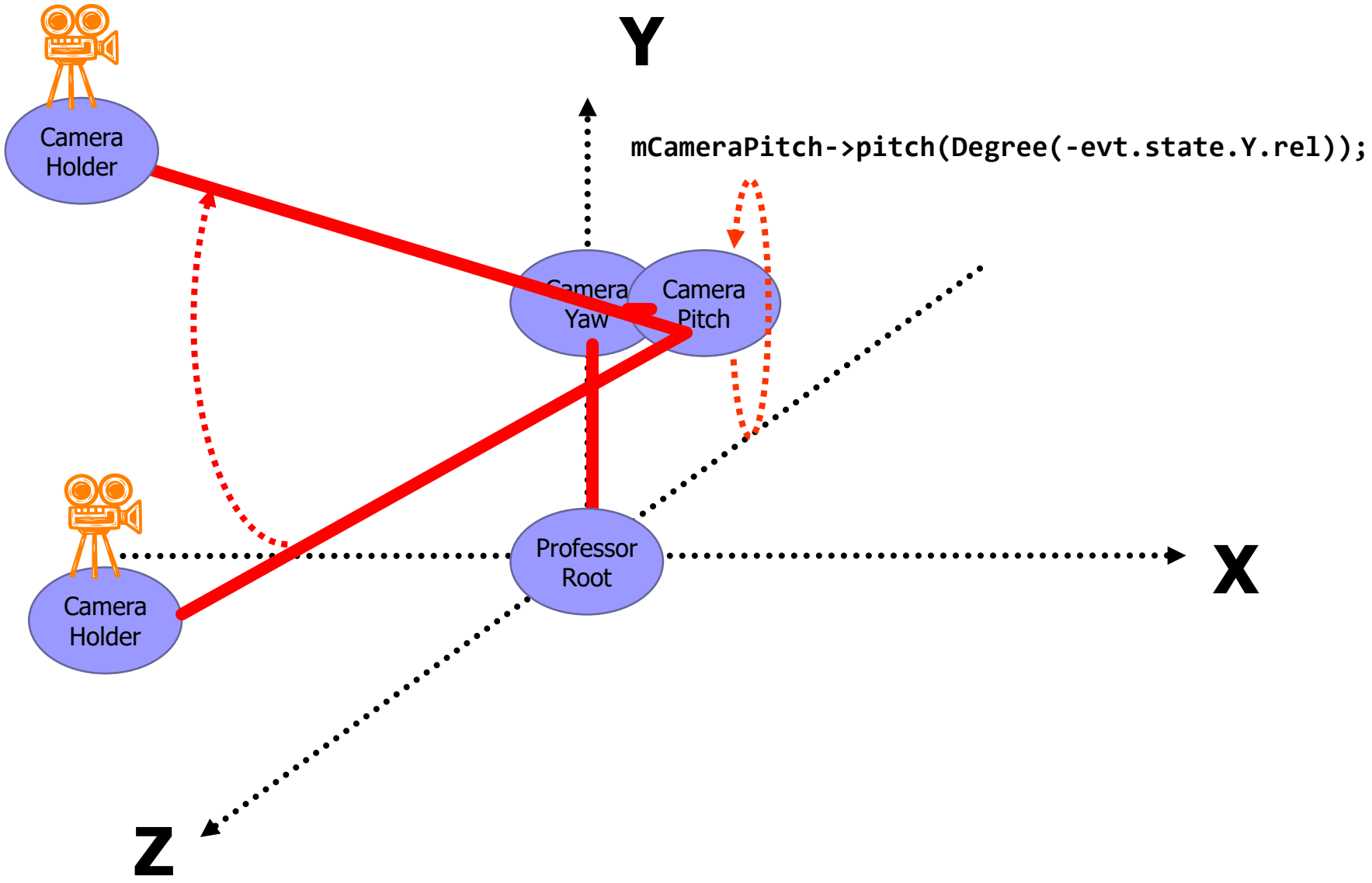
```
SceneNode* cameraYaw = professorRoot->createChildSceneNode("CameraYaw",Vector3(0.0f,120.0f,0.0f));
SceneNode* cameraPitch = cameraYaw->createChildSceneNode("CameraPitch");
SceneNode* cameraHolder = cameraPitch->createChildSceneNode("CameraHolder",Vector3(0.0f,80.0f,500.0f));
```



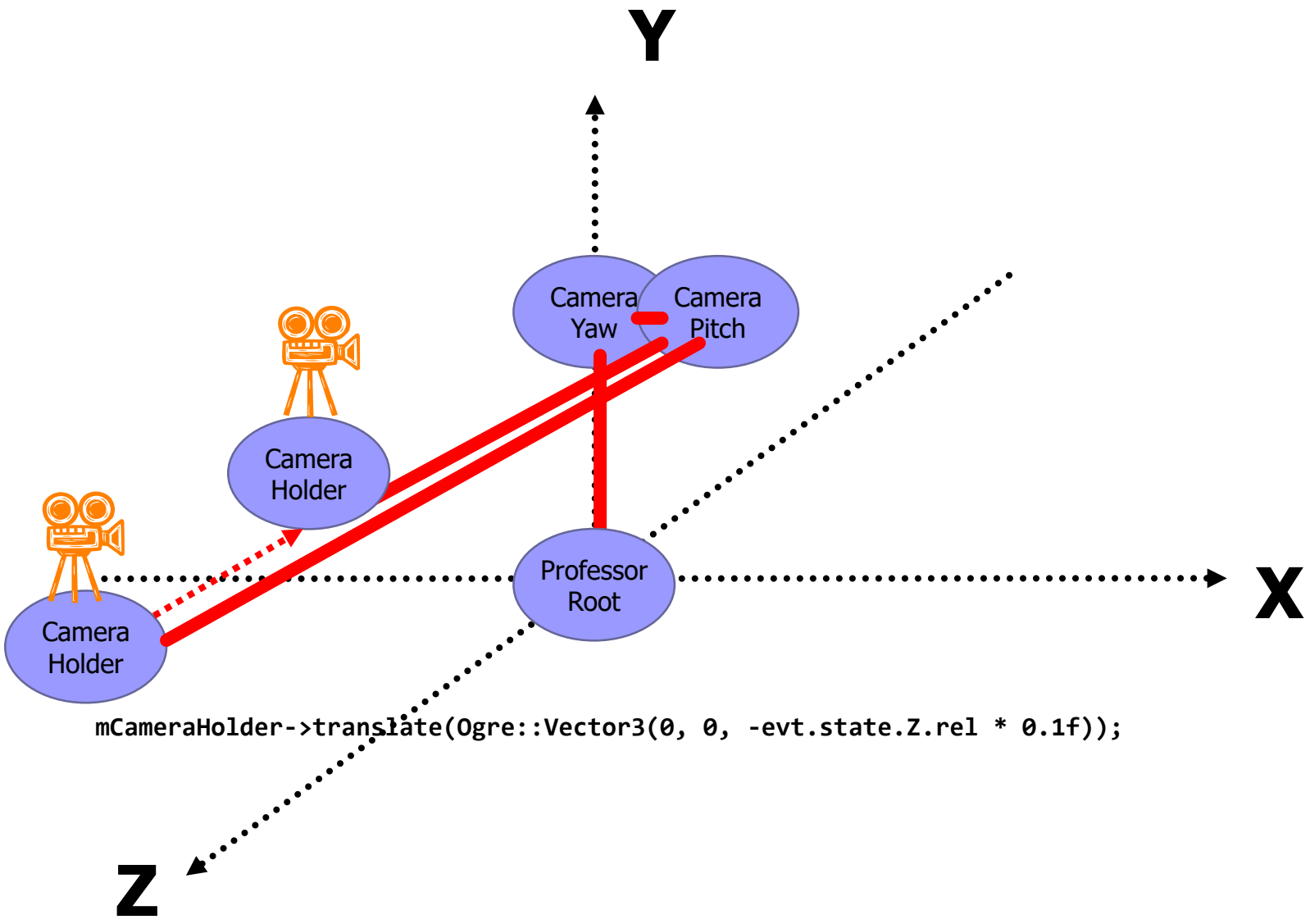
Yaw Control



Pitch Control



Zoom Control




```
mCameraHolder->translate(Ogre::Vector3(0, 0, -evt.state.Z.rel * 0.1f));
```

월드 구성 요소

■ 게임 월드

□ 게임의 배경이 되는 2D 또는 3D 의 가상 게임 세계

□ **“정적 구성 요소”**

- 움직이지 않거나, 게임 플레이에 적극적으로 관여하지 않는 요소들
- 지형, 빌딩, 길, 다리, 등등

□ **“동적 구성 요소”**

- 움직이이면서 게임의 진행에 직접적으로 개입하는 요소들
- 캐릭터, 탈것, 무기, 아이템, 수집할 수 있는 물건
- 파티클 이미터(Particle Emitter)
- 동적 광원(Dynamic Light)
- 영역(Region) - 중요한 게임이벤트를 검출하는, 눈에 보이지 않는 공간
- 스플라인(Spline) - 물체의 경로를 정의

■ 동적 요소과 정적 요소의 구별은 “최적화” 관점에서 보는 것이 타당.

□ 어떤 물체의 상태가 변하지 않는다면?

- 미리 계산함으로써 런타임 시간 계산을 없앨 수 있음.
- 정적 배경 물체들 - 월드 공간으로 미리 배치 가능
- 라이트맵(Light Map)
- 그림자맵(Shadow Map)
- 정적 환경 차폐 정보(Static Ambient Occlusion Information)
- PRT(Precomputed Radiance Transfer) 구면조화 함수 계수(Spherical Harmonics coefficient)

□ 파괴가능한 배경의 구현?

- 세가지 다른 버전(멀쩡한 버전, 손상 입은 버전, 완전히 파괴된 버전)을 미리 준비
- 게임 진행에 따라 교체

동적 요소 구현: 게임 객체

- **게임 객체(Game Object)**
  - 게임 월드의 거의 모든 동적 구소 요소를 지칭
  - Entity, Actor, Agent
  - 게임 객체의 본질: 속성과 행동의 모음
    - 속성 - 게임 객체의 현재 상태
    - 행동 - 시간에 따라, 혹은 이벤트에 반응해서 상태가 변하는 방식
  - 타입(Type)
    - 게임 객체는 타입에 의해 구분
    - 타입이 다른 객체는 다른 속성 스키마와 다른 행동을 가짐
    - 인스턴스 - 한 타입의 인스턴스는 속성 스키마와 행동은 같지만 속성들의 같이 달라짐.
  - 팩맨의 객체 타입
    - 유령, 알약, 파워 쿠키, 팩맨

게임 객체 모델

- **가상의 월드에 존재하는 동적인 존재들을 모델링하고 시뮬레이션할 수 있게 게임 엔진이 지원하는 기능들**

- **어떤 게임을 구성하는 구체적인 존재들을 시뮬레이션할 때 발생하는 문제들을 해결하는 데 쓰이는 특정한 객체지향 프로그래밍 인터페이스**
  - Ogre3d 엔진의 Entity 클래스와 API

- **게임 엔진을 만든 프로그래밍 언어를 확장하는 개념으로도 볼 수 있음.**
  - 파이썬, 루아 등의 스크립트 언어로 제어할 수 있는 게임 객체 인터페이스

런타임 객체 모델

동적으로 게임 객체를 생성하고 파괴하기

- 게임 플레이 중 생성되고 소멸되는 동적 요소의 구현
  - 체력 아이템, 폭발, 새로운 적의 출현

로우레벨 엔진 시스템과 연동

- 게임 객체의 렌더링, 애니메이션, 물리시뮬레이션을 위해 하위 엔진 시스템과 연계가 필요

객체 행동 실시간 시뮬레이션

- 모든 게임 객체들의 상태를 동적으로 업데이트해야 함.
- 경우에 따라 정해진 순서에 따라 업데이트 필요
  - 객체간의 의존성, 객체의 하위 엔진 시스템에 대한 의존성, 하위 시스템 간 상호 의존성

새로운 객체 타입을 정의할 수 있는 기능

- 월드 에디터에서 새로운 객체 타입을 쉽게 정의하고 추가하는 기능
- 대부분의 엔진은 프로그래머의 손을 거쳐야 함.

- **고유 객체 식별자 ID**
  - 숫자, 또는 문자

- **게임 객체 질의**
  - 게임 월드 내의 객체들을 찾을 수 있는 방법(식별자들 통해)
  - 특정 조건을 만족하는 객체의 탐색(ex. 플레이어 캐릭터 주위 20미터 안의 모든 적을 탐색)

- **게임 객체 참조(Reference)**
  - 인스턴스의 포인터, 핸들, 스마트 포인터 등

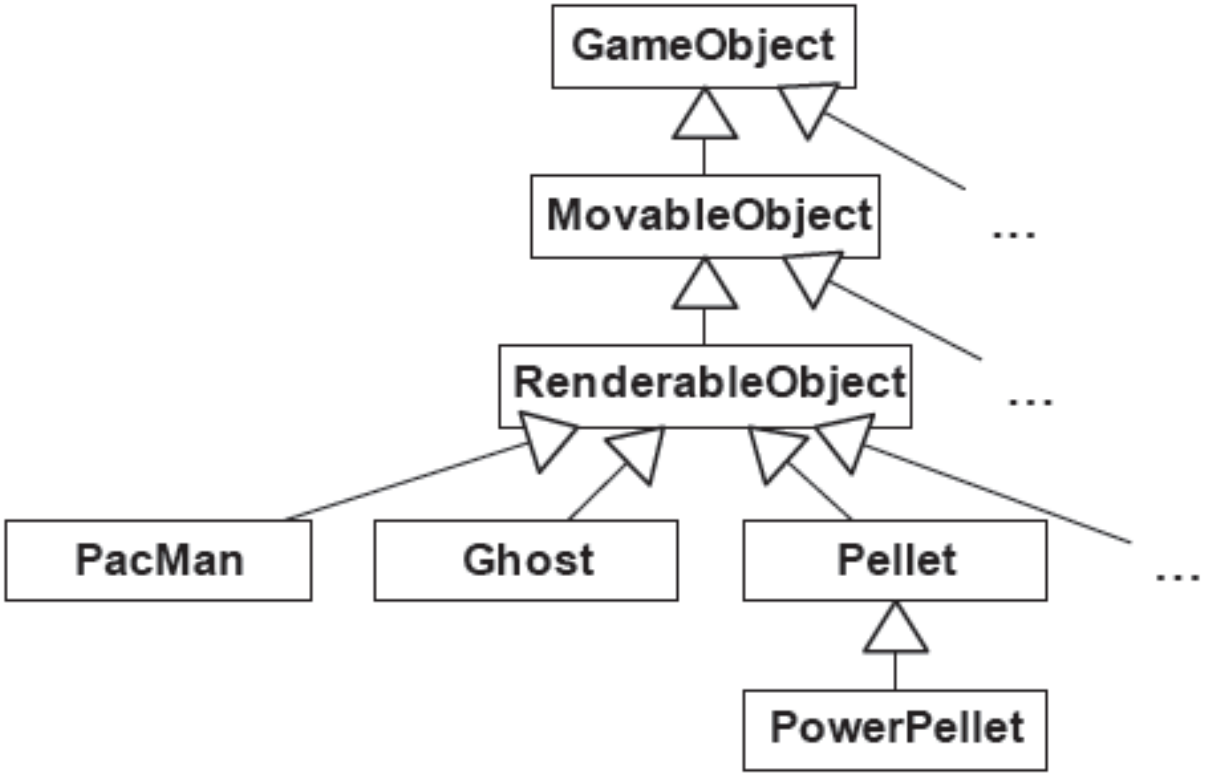
- **유한 상태 기계 지원**
  - 게임 객체를 FSM으로 모델링

- **네트워크 복제(Replication)**
  - 네트워크 게임에서 PC 한 대만 게임 객체를 소유하고 관리
  - 다른 PC에서는 복제된 객체가 존재

- **게임 Save 와 Load**
  - 게임 상태의 저장, 게임 객체들의 serialization

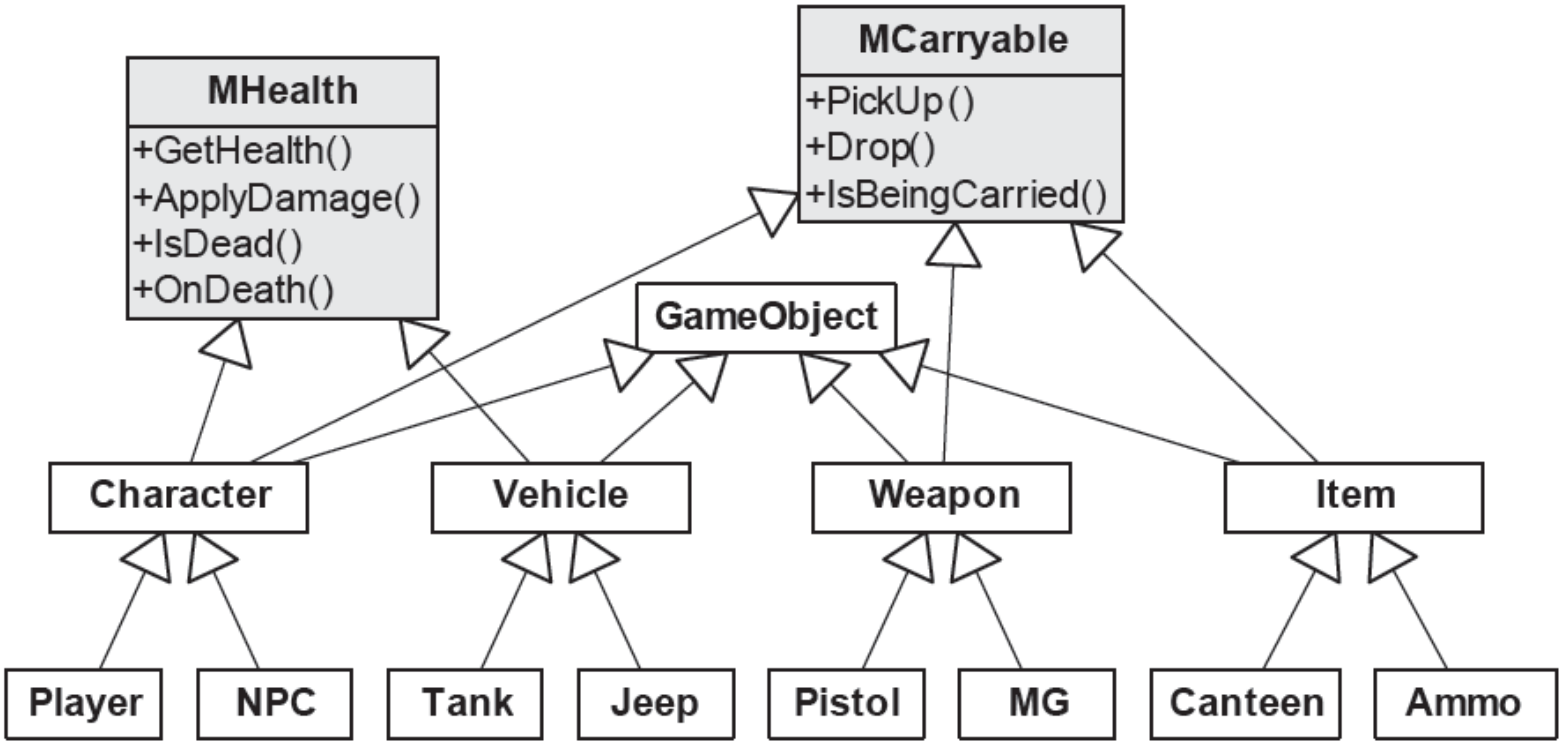
■ 거대 단일 클래스 계층: 팩맨

- 게임 객체 타입을 분류하는데 생물 분류학과 비슷한 분류법을 사용함.
- 클래스 계층이 커지면서 구조는 점점 깊어지는 동시에 넓어짐.
- 하나의 공통 베이스 클래스를 거의 모든 게임 객체들이 상속.



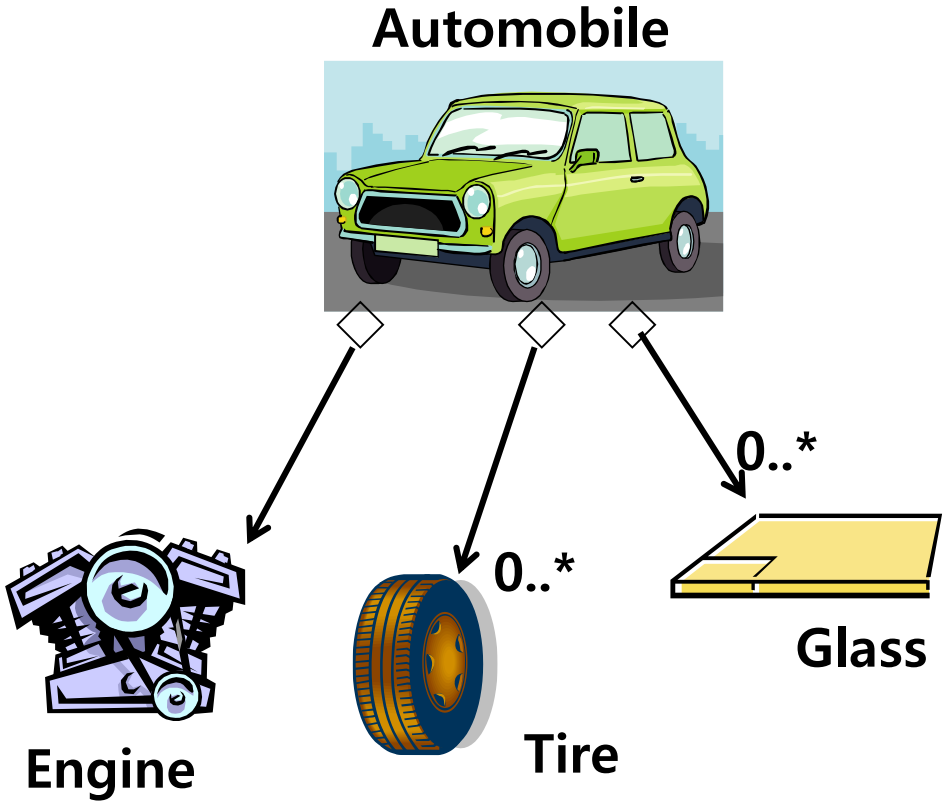
믹스 인 클래스(Mix-In Class)

- 한 클래스는 여러 개의 부모 클래스를 가질 수 있지만, 오직 한 개의 조부모 클래스만 가질 수 있음.
- 중심적인 클래스 구조는 유지하되, 믹스인 클래스(베이스 클래스가 없는 독립된 클래스)는 여러 개 상속 할 수 있음.
- 공통 기능들을 믹스 인 클래스로 모은 후, 상속받게 할 수 있음.



조합(Aggregation) - 집합

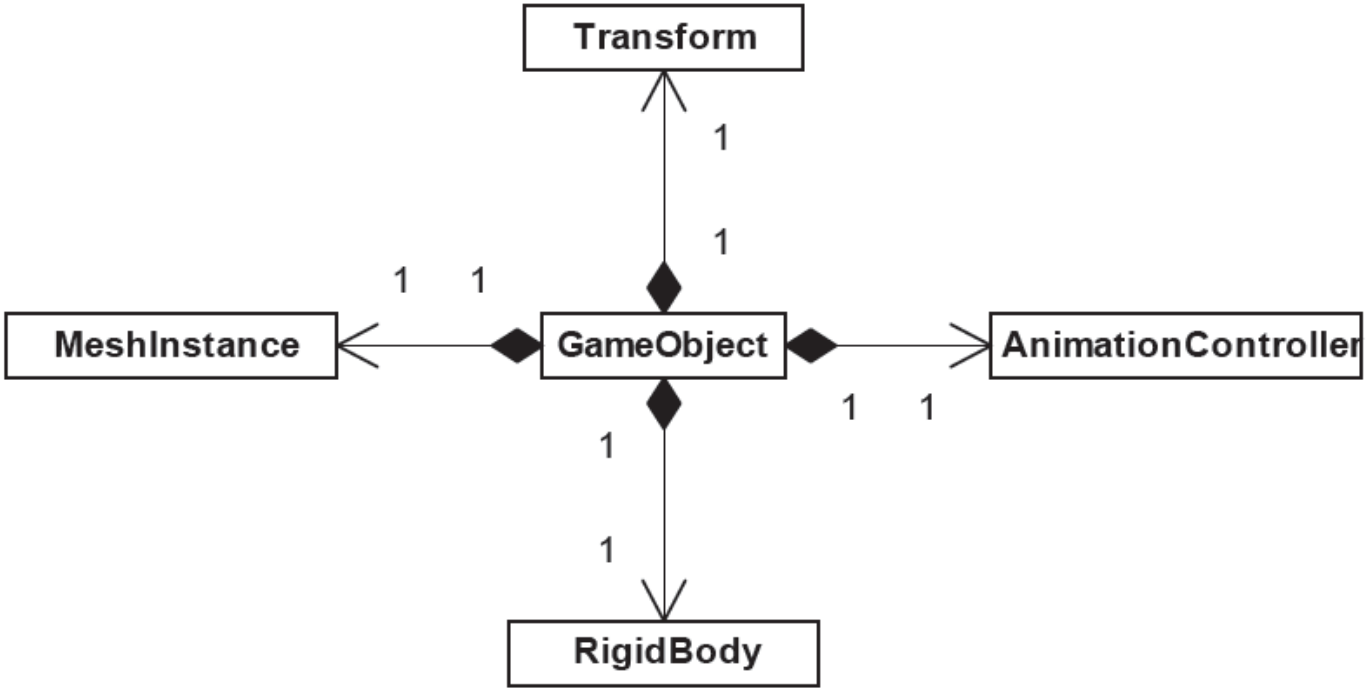
- 포함된 인스턴스의 독자적 행동이 가능하다면? - 포인터 또는 참조를 활용.




```
class Automobile {
   Engine *theEngine;
   Tire *theTire[4];
   Glass *theGlass[6];
...
};
```

컴포넌트 객체의 합성 구조

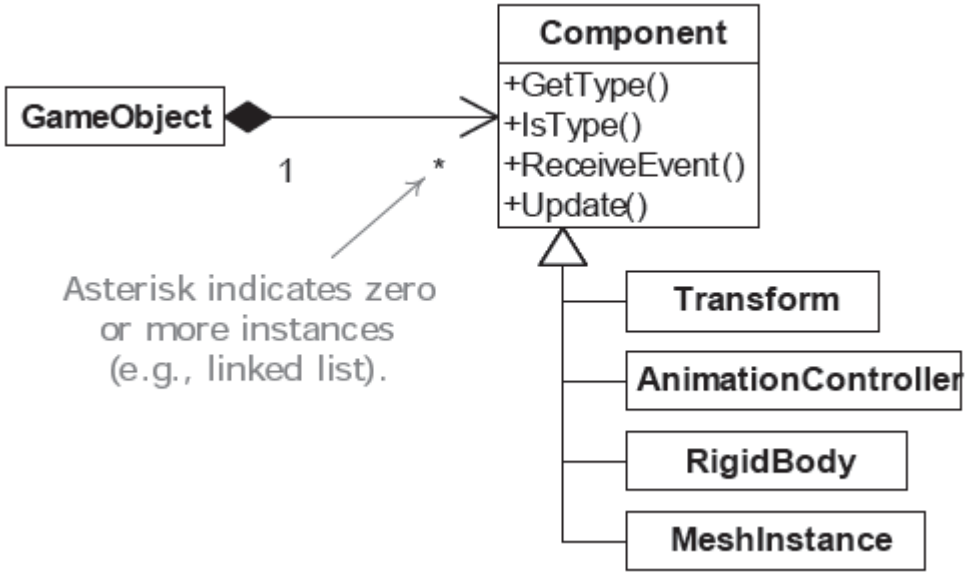
- 서비스 객체
- GmaeObject의 여러 기능을 독립된 클래스로 분리
- 한 클래스는 한 가지 잘 정의된 서비스만 지원
- 허브 클래스(Game Object)가 컴포넌트 객체들의 수명을 관리
- 게임 객체들을 GameObject를 상속받아서 정의
  - 상속된 클래스의 생성자에서 각자 필요에 따라 컴포넌트를 생성하면 됨.



Generic Component

- **Generic Component(일반컴포넌트)**
  - 컴포넌트들의 베이스 클래스

- **GameObject 루트 클래스가 제너릭컴포넌트의 리스트를 관리**
- **GameObject 객체는 구체적으로 어떤 타입의 컴포넌트가 있는지 신경 쓸 필요가 없음.**
- **장점은?**
  - 새 타입의 컴포넌트를 만들 때, GameObject 클래스를 수정할 필요가 없음.
  - 게임 객체에서 가질 수 있는 컴포넌트의 수에 제한이 없음.



상태(State)

- A state is a condition in which an object can reside during its lifetime while it satisfies some condition, performs an activity, or waits for an event.

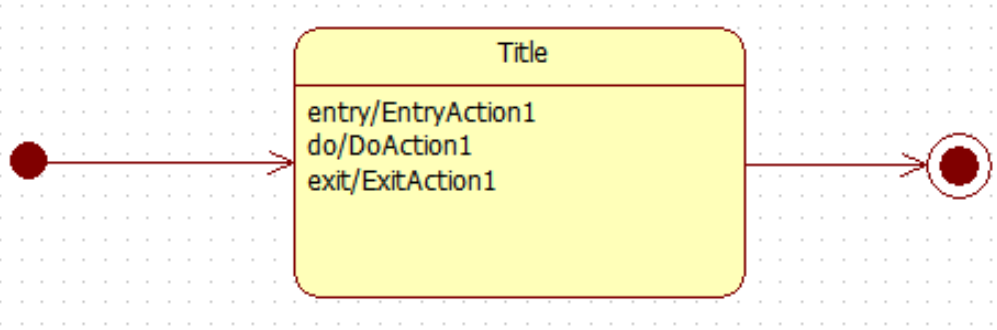
- 오브젝트가 놓여있는 상황으로써, 정해진 조건을 만족하는 동안, 요구되는 태스크를 지속적으로 수행하면서, 이벤트를 기다리며 머무르고 있게 됨.

- Entry Action - 오브젝트가 어떤 상태에 진입할 때, 처음으로 수행되는 일.

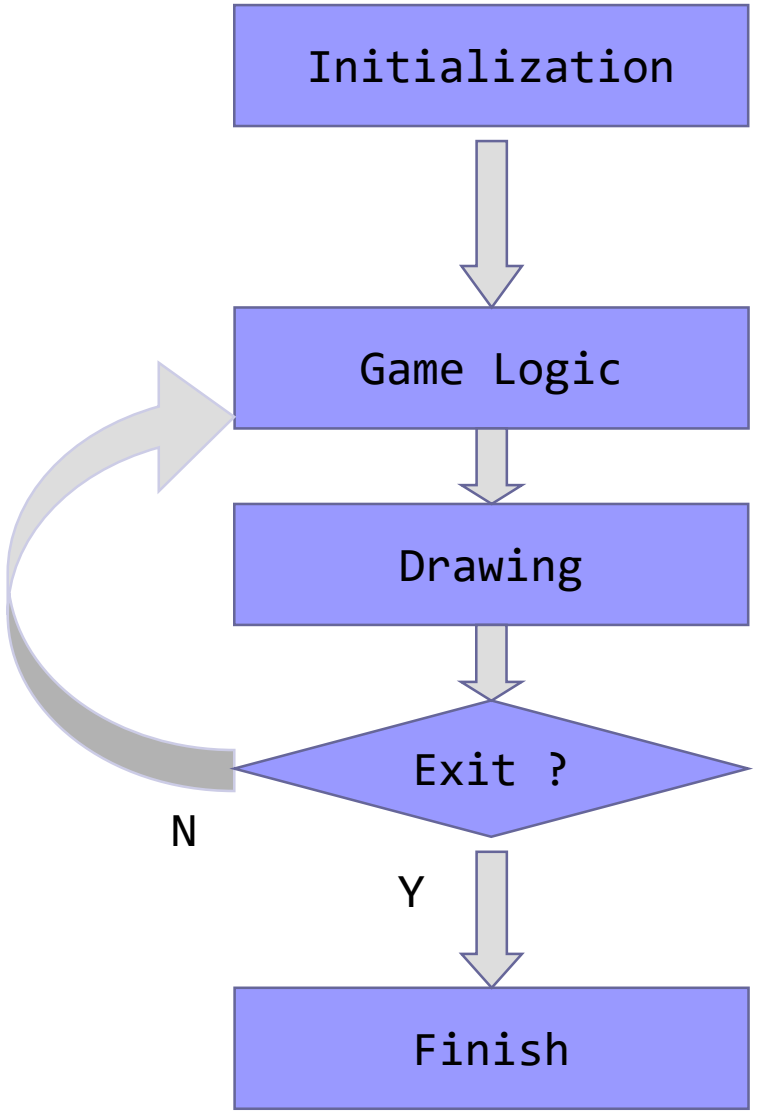
- Exit Action - 오브젝트가 어떤 상태를 빠져나갈 때, 마지막으로 수행되는 일.

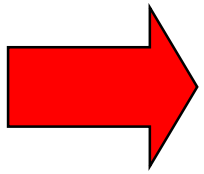
- Event - 상태를 변경시키는 내/외부의 자극

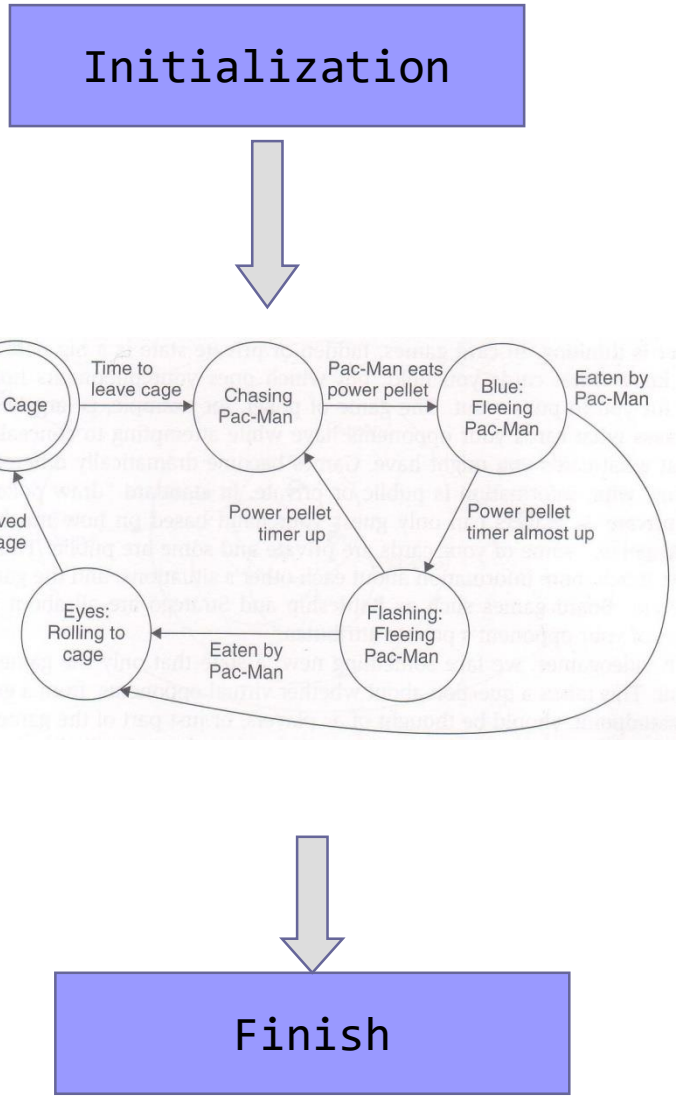
- Do Activity - 오브젝트가 특정한 상태에 머무를 때, 수행하게 되는 일.



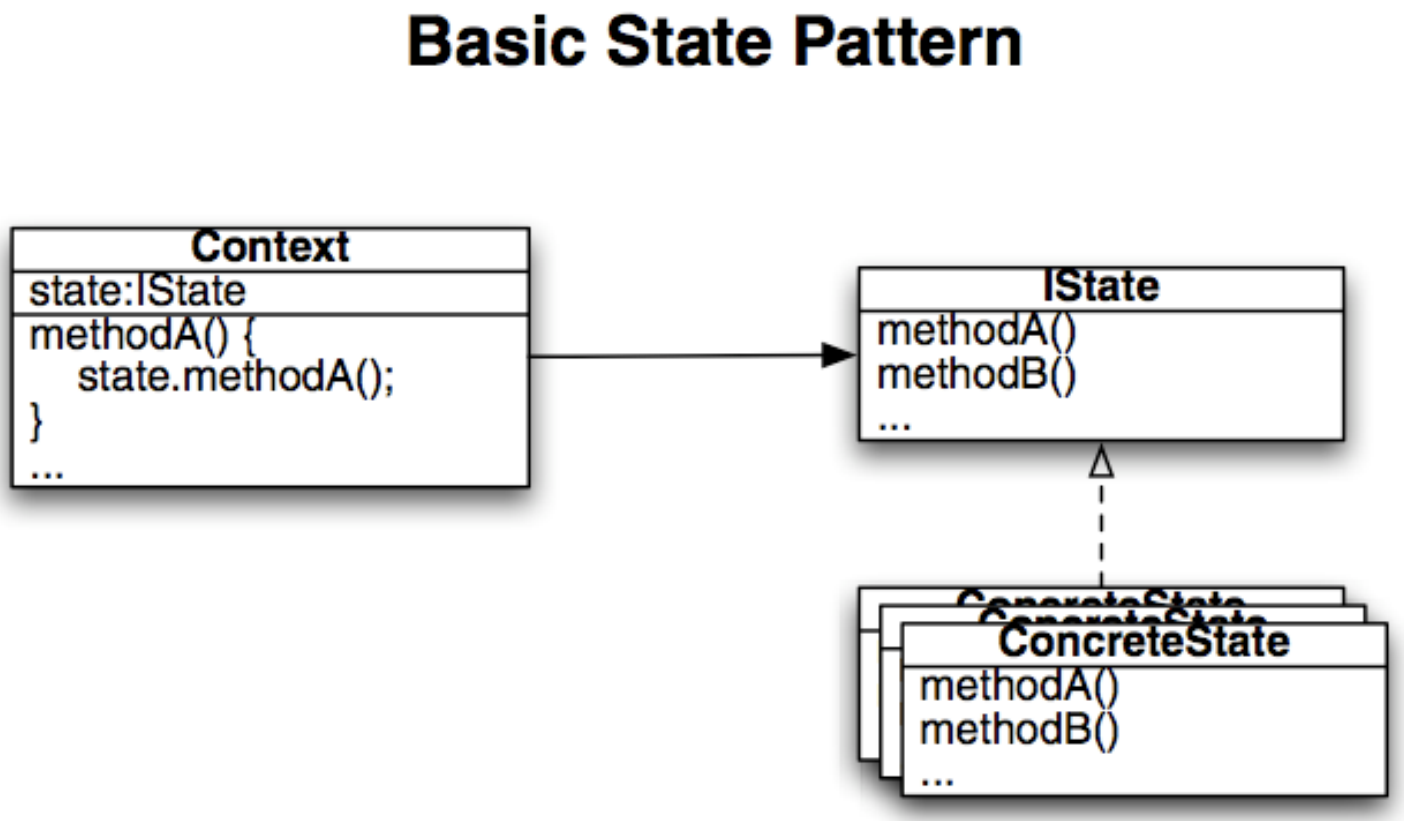
게임 루프에서 게임 상태의 구현?







스테이트 패턴(State Pattern)의 이용



State Pattern 구현