# 책 발췌: 게임 엔진 아키텍처
# (Book Excerpt: Game Engine Architecture)

제이슨 그레고리(Jason Gregory)

가마수트라 등록일(2009. 11. 25)

http://www.gamasutra.com/view/feature/4199/book_excerpt_game_engine_.php

*[가마수트라는 노티 독의 프로그래머 제이슨 그레고리의 책 <게임 엔진 아키텍처Game Engine Architecture>를 발췌해 전재했다. 이 책에는 게임 엔진을 개발할 때 고려해야 할 엄청난 양의 특정 데이터들이 실려 있다. 이번 발췌, 특히 원서의 14장부터의 발췌 내용은 게임 엔진이 물체를 다루는 법에 대해 다루고 있다. 더 자세한 정보는 이 책의 공식 사이트를 참조하라.]*

## 실시간으로 게임 물체 업데이트

단순한 것과 복잡한 것을 막론하고 모든 게임 엔진은 시간에 따라 모든 게임 내 물체의 내부 상황을 업데이트할 방법이 있어야 한다. 게임 내 물체의 상태는 그 물체의 특성값으로 정의할 수 있다. 일부에서는 특성이라고 부르지 않고 속성이라고 부르기도 하며, C++ 언어에서는 데이터 멤버(데이터 구성인자)라고도 부른다. 예를 들어 <퐁>의 공 상태는 스크린 상의 x 및 y 좌표. 그리고 속도(속력 및 진행방향)를 통해 나타낼 수 있다. 게임은 동적 시간 기반 시뮬레이션이므로 게임 내 물체의 상태는 특정한 시간적 순간의 게임 설정을 나타내는 것이다. 다른 말로 하면 게임 내 물체의 시간개념은 연속적이라기보다는 불연속적이다. 그러나 앞으로도 살펴보겠지만 흔히 빠질 수 있는 위험을 막기 위해 물체의 상태는 연속적으로 변한다고 가정하고, 이것이 엔진에 의해 불연속적으로 샘플링된다고 보는 것이 유용할 것이다.

다음 논의에서 우리는 임의의 시점 $t$의 물체 $i$의 상태를 나타내기 위해 $\mathbf{S}_i(t)$라는 기호를 사용할 것이다. 이 기사에서의 벡터 기호 사용은 수학적으로 엄밀히 정확하지는 않다. 그러나 이 벡터 기호의 사용으로 인해 게임 내 물체의 상태는 다양한 데이터 유형의 모든 정보를 포함한 이질적인 n-차원 벡터의 움직임과 비슷하다는 것을 알 수 있을 것이다. 여기서 쓰이는 상태라는 말은 유한 상태 기계에서 사용되는 상태라는 말과는 다르다. 게임 내 물체를 하나 또는 여러 개의 유한 상태 기계의 관점으로 구현할 수도 있다. 그러나 그 경우 각 유한 상태 기계의 현재 상황을 명확하게 규정하는 것은 그저 게임 물체의 전반적인 상황 벡터 $\mathbf{S}(t)$의 일부를 나타낼 뿐이다.

대부분의 하위 레벨 엔진 하위체계(렌더링, 애니메이션, 충돌, 물리, 음향 등)는 주기적인 업데이트가 필요하다. 그리고 게임 물체 체계라고 예외는 아니다. 제7장에서 살펴보았듯이 업데이트는 보통 게임 루프라고 불리우는 단일 마스터 루프, 또는 별도의 스레드 내에서

실행되는 여러 게임 루프를 통해 이루어진다. 거의 모든 게임 엔진은 게임 물체 상태 업데이트를 주 게임 루프의 일부로 실시한다. 다른 말로 하면 주기적인 서비스가 필요한 다른 엔진 하위체계와 마찬가지로 게임 물체 모델을 취급한다는 것이다.

그러므로 게임 물체 업데이트는 이전 시간의 물체 상황 $S_i(t - \Delta t)$에 대한 현재 시간의 물체 상황 $S_i(t)$를 결정하는 과정이라고 볼 수 있다. 모든 물체 상황이 업데이트되고 나면 현재 시간 $t$는 새로운 이전 시간 ($t - \Delta t$)가 된다. 그리고 이 과정은 게임이 실행되는 한 계속 이루어진다. 보통 게임 엔진은 하나 이상의 시계를 보유하고 있다. 이 시계들 중 하나는 실제 시간을 정확히 지키고 다른 시계들은 실제 시간을 따르거나 그렇지 않거나 한다. 이들 시계들은 엔진에 절대 시간 $t$ 및 또는 게임 루프의 반복 사이의 시간 변화 $\Delta t$를 알려준다. 게임 물체 상황의 업데이트를 주도하는 이 시계는 보통 실제 시간으로부터 분기하는 것이 가능하다. 이 덕분에 게임 물체의 움직임을 일시 정지하거나, 감속하거나, 가속하거나, 역으로 움직이는 등 게임 디자인의 필요를 충족시키는 모든 동작을 실시하는 것이 가능하다. 이러한 기능은 게임의 디버깅 및 개발에도 매우 중요하다.

제1장에서도 언급했다시피, 게임 물체 업데이트 시스템은 컴퓨터 과학에서 말하는 동적 실시간 행위자 기반 컴퓨터 시뮬레이션의 사례이다. 또한 게임 물체 업데이트 시스템은 불연속 이벤트 시뮬레이션의 여러 국면을 보여준다(자세한 것은 14.7절을 참조하라). 컴퓨터 과학에서 잘 연구된 이 영역들은 상호작용형 엔터테인먼트 영역 밖에도 많이 적용할 수 있다. 게임은 가장 복잡한 행위자 기반 시뮬레이션 중 하나이다. 이제부터 살펴보듯이 동적인 상호작용형 가상 환경 내에서 시간에 따른 게임 물체 상태 업데이트를 제대로 하기란 놀랄 만큼 어렵다. 게임 프로그래머들은 행위자 기반 및 불연속 이벤트 시뮬레이션이라는 더욱 광대한 영역을 공부함으로써 게임 물체 업데이트에 대해 많은 것을 배울 수 있다. 그리고 이 두 영역의 연구자들도 게임 엔진 디자인에서 뭔가를 배워갈 수 있을 것이다.

모든 상위 레벨 게임 엔진 체계가 그렇듯이 모든 엔진은 약간씩(경우에 따라서는 크게) 다른 접근방식을 취하고 있다. 그러나 앞서와 마찬가지로 대부분의 게임 팀이 접하는 문제는 거의 비슷하다. 그리고 거의 모든 게임 엔진에서 특정 디자인 양상이 꾸준히 나타나는 경향이 강하다. 이 절에서 우리는 일반적인 문제 및 그에 대한 일반적인 해결책을 찾아볼 것이다. 여기 제시된 것과는 많이 다른 해결책을 사용해야 하는 게임 엔진은 물론, 여기서 미처 다루지 못한 독특한 문제를 일으키는 게임 디자인도 있을 수 있다는 점을 염두에 두라.

### 14.6.1. 유효하지 않은 간단한 접근법

여러 게임 물체의 집합의 상태를 업데이트하는 가장 간단한 방법은 집합에 대해 절차를 반복하고, Update() 같은 이름을 붙인 각 물체에 가상 함수를 호출하는 것이다. 이는 주 게임 루프의 반복, 예를 들면 프레임이 바뀔 때마다 일반적으로 실시되는 절차이다. 게임 물체 등급은 Update() 함수의 맞춤형 구현을 제공함으로써 해당 유형의 물체 상태를 다음 불연속 시간 지수에 맞게 발전시키는데 필요한 임무를 수행할 수 있다. 이전 프레임의 시간 델타는 업데이트 기능에 패스되어 물체가 시간의 흐름에 따른 적절한 설명을 받을 수 있도록 한다. 아주 간단히 말하면 Update() 함수의 특징은 다음과 같이 나타낼 수 있다.

```
virtual void Update(float dt);
```

우리는 앞으로 진행될 논의의 목적에 맞게 엔진이 단일 물체 계층을 사용하고 있다고 가정하겠다. 이 속에서 각 게임 물체는 단일 계층의 단일 인스턴스로 표시된다. 그러나 이

개념을 거의 모든 물체 중심 디자인으로 쉽게 확장할 수도 있다. 예를 들어 구성요소 기반 물체 모델을 업데이트하려면 각 게임 물체를 만드는 모든 구성요소에 Update()를 호출하던가, 또는 허브 오브젝트에 Update()를 호출하여 그것이 옳다고 보는 관련된 구성요소를 적절하게 업데이트할 수 있다. 우리는 특정한 종류의 Update() 함수를 모든 프레임의 각 속성 인스턴스에 호출함으로써 이 개념을 속성 중심 디자인에도 확장할 수 있다.

그러나 자세히 뜯어보면 복병이 숨어 있다고들 한다. 따라서 여기서 두 가지의 중요한 세부사항을 조사해 보기로 하겠다. 우선, 모든 게임 물체 집합을 유지하려면 어떻게 해야 할까? 그리고 Update() 함수가 작동하게 하려면 어떻게 해야 할까?

#### 14.6.1.1. 활성 게임 물체의 집합을 유지

활성 게임 물체의 집합을 유지하는 데는 보통 개체 관리자 계층이 사용된다. 이 계층은 보통 게임세계 또는 게임 물체 관리자라는 이름이 붙어 있다. 게임이 진행되면서 게임 물체는 스폰되고 파괴되므로 게임 물체 집합은 보통 동적이어야 한다. 따라서 게임 물체에 대한 포인터, 지능형 포인터, 핸들 등의 링크 리스트를 쓰는 것이 쉽고도 효율적인 접근방식이다. 물론 일부 게임 엔진에서는 게임 물체의 동적 스폰 또는 파괴를 허용하지 않기도 한다. 이런 게임 엔진들은 링크 리스트 대신 정적 규모의 게임 물체 포인터, 지능형 포인터, 핸들 어레이를 사용할 수 있다. 아래에서도 살펴보듯이 대부분의 엔진은 간단한 링크 리스트가 아닌 훨씬 복잡한 데이터 구조를 사용해 게임 물체를 추적한다. 그러나 당분간은 간단하게 살펴보기 위해 데이터 구조를 링크 리스트로 시각화하기로 하겠다.

#### 14.6.1.2. Update() 함수의 책임

게임 물체의 Update() 함수는 기본적으로 과거의 상태 $S(t - \Delta t)$에 대한 현재 불연속 시간 지수 $S(t)$의 게임 물체 상태를 결정하는 책임을 진다. 이렇게 할 때 물체에 강체역학을 적용하거나, 사전에 작성된 애니메이션을 샘플링하거나, 현재의 시간 단계 중 벌어진 이벤트에 반응하거나 할 수도 있다.

대부분의 게임 물체는 하나 이상의 엔진 하위 체계와 상호작용을 한다. 이들은 애니메이션화 되거나, 렌더링 되거나, 파편 효과를 발생하거나, 음향을 발생하거나, 다른 물체 또는 고정된 지형과 충돌하는 등의 움직임을 보인다. 이들 각각의 시스템은 시간에 따라 업데이트 되어야 하는 내부 상태가 있다. 업데이트는 보통 프레임당 한 번 이상 일어난다. 이 모든 하위체계를 게임 물체의 Update() 함수 내에서 간단히 직접 업데이트하는 것이 합리적이고 직관적인 것 같다. 전차(tank)라는 물체에 대한 다음 가상 업데이트 함수를 예로 들어 살펴보자.

```
virtual void Tank::Update(float dt)
{
// Update the state of the tank itself.
MoveTank(dt);

DeflectTurret(dt);
FireIfNecessary();

// Now update low-level engine subsystems on behalf
// of this tank. (NOT a good idea... see below!)
```

```
m_pAnimationComponent->Update(dt);

m_pCollisionComponent->Update(dt);

m_pPhysicsComponent->Update(dt);

m_pAudioComponent->Update(dt);
m_pRenderingComponent->draw();
}
```

Update() 함수가 이와 같은 구조를 갖추고 있음을 감안하면, 다음과 같은 게임 물체의 업데이트만으로도 게임 루프를 거의 대부분 움직일 수 있다.

```
while (true)
{
PollJoypad();

float dt = g_gameClock.CalculateDeltaTime();

for (each gameObject)
{
// This hypothetical Update() function updates
// all engine subsystems!
gameObject.Update(dt);
}

g_renderingEngine.SwapBuffers();
}
```

물체 업데이트에 대해 위에 나온 간단한 접근방식이 매력적으로 보이기는 하지만, 상용 수준의 게임 엔진에는 대체로 쓸모가 없는 방식이다. 다음 절에서는 이 간단한 접근방식의 문제점을 살펴보고 각 문제점을 해결하는 일반적인 방식을 알아보기로 하겠다.

### 14.6.2. 성능의 제약과 일괄작업 업데이트

대부분의 하위 레벨 게임 엔진 체계는 극도로 엄격한 성능 제약이 걸려 있다. 이들은 대량의 데이터를 사용해 작동되며 각 프레임마다 가급적 빨리 엄청난 수의 연산을 해내야 한다. 그 결과 대부분의 엔진 체계는 일괄작업 업데이트를 이용한다. 예를 들어 한 번의 일괄작업으로 많은 수의 애니메이션을 업데이트하는 것이 충돌 탐지, 물리 시뮬레이션, 렌더링 등 상관없는 작업과 함께 끼워진 각 물체의 애니메이션을 업데이트하는 것보다 훨씬 효율적이다.

대부분의 상용 게임 엔진에서 각 엔진 하위 체계는 각 물체의 Update() 함수로부터의 게임별 물체 기반보다는 주 게임 루프에 의해 직간접적으로 업데이트된다. 게임 물체가 개별 엔진 하위 체계의 서비스를 요구한다면 이는 서브시스템이 일부 하위 체계에 특화된 상태 정보를 대표에 할당하도록 지시한다. 예를 들어 삼각형 메쉬를 통해 렌더링하고자 하는 게임 물체가 그 용도로 메쉬 인스턴스를 할당하기를 렌더링 하위체계에 요청할 수 있다. 메쉬 인스턴스는 삼각형

메쉬의 단일 인스턴스를 대표하며, 이는 인스턴스가 보이건 보이지 않건 간에 세계 공간 내부의 인스턴스의 위치, 방향, 스케일 및 관련 인스턴스별 정보를 계속 추적한다. 렌더링 엔진은 메쉬 인스턴스 집합을 내부적으로 유지한다. 이는 런타임 성능 극대화를 위해 적당한 경우에도 메쉬 인스턴스를 관리할 수 있다. 게임 물체는 메쉬 인스턴스 물체의 속성을 조작함으로서 렌더링 방식을 제어한다. 그러나 게임 물체는 에쉬 인스턴스의 렌더링을 직접 조절하지는 않는다. 대신 모든 게임 물체는 스스로를 업데이트할 기회를 가지며, 렌더링 엔진은 모든 보이는 메쉬 인스턴스를 하나의 일괄작업 업데이트 내에 그려넣는다.

일괄작업 업데이트에서 개별 게임 물체, 예를 들면 가상의 전차라는 물체의 Update() 함수는 이렇게 나타낼 수 있다.

```
virtual void Tank::Update(float dt)
{
// Update the state of the tank itself.
MoveTank(dt);

DeflectTurret(dt);

FireIfNecessary();

// Control the properties of my various engine
// subsystem components, but do NOT update
// them here...
if (justExploded)
{
m_pAnimationComponent->PlayAnimation("explode");
}
if (isVisible)
{

m_pCollisionComponent->Activate();
m_pRenderingComponent->Show();
}
else
{

m_pCollisionComponent->Deactivate();
m_pRenderingComponent->Hide();
}
// etc.
}
```

그러면 게임 루프는 결국 이런 모양이 된다.

```
while (true)
{
```

```
PollJoypad();

float dt = g_gameClock.CalculateDeltaTime();

for (each gameObject)
{
gameObject.Update(dt);
}

g_animationEngine.Update(dt);

g_physicsEngine.Simulate(dt);

g_collisionEngine.DetectAndResolveCollisions(dt);

g_audioEngine.Update(dt);

g_renderingEngine.RenderFrameAndSwapBuffers();
}
```

일괄처리 업데이트는 성능상의 많은 이익을 제공해줄 뿐만 아니라 다른 이익도 있다.

최대 캐시 일관성. 일관된 업데이트를 하면 엔진 하위체계가 최대 캐시 일관성에 도달할 수 있다. 물체별 데이터가 내부적으로 유지되며 인접한 단일 RAM의 영역 내에서 배열되기 때문이다.
최소 복제 계산. 글로벌 계산은 한 번 실시되면 각 물체에 다시 수행되지 않고 많은 게임 물체에 다시 수행될 수 있다.
리소스 재분배 감소. 엔진 하위체계는 업데이트 중 메모리 및 또는 기타 자원을 할당하고 관리해야 하는 경우가 많다. 개별 하위체계의 업데이트가 다른 엔진 하위체계로 인터리브되면 이러한 자원을 풀어놓아 처리되는 각 게임 물체로 재할당해야 한다. 그러나 업데이트가 일관되면 자원은 프레임 별로 할당되어 배치 내의 모든 물체에 재사용된다.
효율적인 파이프라이닝. 많은 엔진 하위체계가 게임 내 각 물체에 대해 사실상 같은 연산 집합을 수행하고 있다. 업데이트가 일관화되면 새로운 최적화가 가능해지고 특화된 하드웨어 자원이 영향을 받는다. 예를 들어 플레이스테이션3은 SPU로 알려진 수많은 고속 마이크로프로세서의 묶음을 가지고 있는데, 이들 하나하나는 개별적인 고속 메모리 영역을 가지고 있다. 애니메이션 배치를 처리할 때면 한 캐릭터의 포즈를 계산하는 동안 다음 캐릭터의 데이터에 대한 직접 메모리 접근을 SPU 메모리 내에 실시한다. 각 물체를 별도로 처리하면 이러한 종류의 유사성은 얻을 수 없다.
일괄 업데이팅 접근법을 선호하는 이유는 성능상의 이점 뿐이 아니다. 일부 엔진의 하위체계는 물체별 기반에서 업데이트 되었을 때 전혀 작동하지 않는다. 예를 들어 다양한 동적 강체 시스템 내의 충돌을 해결하고자 할 때 각 물체를 독립적으로 생각해서는 만족스러운 해결책이 나오지 않는다. 이러한 물체 간의 상호 충돌은 반복적 접근 또는 선형 체계 해결을 통해 그룹 단위로

해결해야 한다.

---

14.6.3. 물체 및 하위 체계의 독립성

설령 성능에 신경을 쓰지 않는다고 해도, 한 게임 물체가 다른 게임 물체에 의존하고 있다면 간단한 물체별 업데이트 접근법은 무너질 수 밖에 없다. 예를 들어 인간 캐릭터가 팔로 고양이를 안고 있는 경우를 들어보자. 고양이 골격의 월드공간 포즈를 산출해내기 위해서는 우선 인간의 월드공간 포즈를 산출해야 한다. 각 물체의 순서가 업데이트되는 것이 게임의 적절한 함수화에 중요하다는 뜻이다.

엔진 하위체계가 다른 엔진 하위체계에 의존할 경우 관련된 문제가 또 발생한다. 예를 들면 누더기 인형의 물리 시뮬레이션은 애니메이션 엔진과 조화를 이루어 업데이트 되어야 한다. 보통 애니메이션 시스템은 중간의 지역공간 골격 포즈를 만들어낸다. 이러한 관절 변형은 월드공간으로 전환되며 물리 체계 내의 골격 근처에 연결된 강체 체계에 적용된다. 강체는 물리 체계에 의해 시간에 따라 구현되며 관절이 최종적으로 위치하는 곳은 골격 내의 해당 관절의 위치가 된다. 마지막으로 애니메이션 시스템은 최종 월드공간 포즈를 산출하여 매트릭스 팔레트로 스키닝한다. 다시 한 번 말하지만 애니메이션과 물리 체계의 업데이트는 정확한 결과를 얻기 위해 개별 순서에 따라 이루어져야 한다. 이러한 종류의 하위체계 상호간 종속성은 게임 엔진 설계에서 흔하다.

14.6.3.1. 단계적 업데이트

하위체계 상호간 종속성을 설명하기 위해 엔진 하위체계 업데이트를 주 게임 루프 내에서 적절한 순서에 맞추어 코딩할 수 있다. 예를 들어 애니메이션 시스템과 누더기 인형 물리 간의 상호작용을 처리하기 위해 다음과 같이 작성할 수 있다.

```
while (true) // main game loop
{

// ...

g_animationEngine.CalculateIntermediatePoses(dt);
```

```
g_ragdollSystem.ApplySkeletonsToRagDolls();

g_physicsEngine.Simulate(dt); // runs ragdolls too

g_collisionEngine.DetectAndResolveCollisions(dt);

g_ragdollSystem.ApplyRagDollsToSkeletons();

g_animationEngine.FinalizePoseAndMatrixPalette();

// ...

}
```

게임 루프의 적절한 시간에 게임 물체의 상태를 업데이트 할 때는 주의를 기울여야 한다. 이는 프레임별 게임 물체마다 단일 Update() 함수를 호출하는 것만큼 단순하지 않은 경우가 많다. 게임 물체는 다양한 엔진 하위 체계에서 수행한 연산의 중간결과에 의존하는 경우가 많다. 예를 들어 게임 물체가 애니메이션 시스템이 업데이트를 실행하기 이전에 플레이된 애니메이션을 유구할 수도 있다. 그러나 또한 이 물체는 누더기 인형 물리 체계 및 또는 최종 포즈와 매트릭스 팔레트가 생성되기 이전 애니메이션 시스템에서 생성한 중간 포즈를 절차적으로 조절해야 할 수도 있다. 이는 물체를 두 번 업데이트해야 한다는 뜻이다. 한 번은 애니메이션이 중간 포즈를 연산하기 전, 그리고 또 한번은 그 이후에 업데이트해야 한다.

많은 게임 엔진이 게임 물체가 프레임 내의 다양한 포인트에서 업데이트하는 것을 허용하고 있다. 예를 들면 어떤 엔진에서는 게임 물체를 세 번 업데이트할 수 있다. 한 번은 애니메이션 블렌딩 이전, 또 한 번은 애니메이션 블렌딩 이후와 최종 포즈 생성 이전 사이. 그리고 또 한 번은 최종 포즈 생성 이후이다. 이는 각 게임 물체 계층을 후크로 작용하는 3개의 가상 함수와 함께 제공함으로서  가능하다. 이러한 체계에서 게임 루프는 이렇게 끝이 난다.

```
while (true) // main game loop
{

// ...

for (each gameObject)
```

```
{
gameObject.PreAnimUpdate(dt);
}
g_animationEngine.CalculateIntermediatePoses(dt);
for (each gameObject)
{
gameObject.PostAnimUpdate(dt);
}
g_ragdollSystem.ApplySkeletonsToRagDolls();
g_physicsEngine.Simulate(dt); // runs ragdolls too
g_collisionEngine.DetectAndResolveCollisions(dt);
g_ragdollSystem.ApplyRagDollsToSkeletons();
g_animationEngine.FinalizePoseAndMatrixPalette();
for (each gameObject)
{
gameObject.FinalUpdate(dt);
}
// ...
}
```

게임 물체에 많은 업데이트 단계를 성공적으로 부여할 수 있다. 그러나 모든 게임 물체에 반복을 실시하고, 각개 물체에 가상 함수를 호출하는 것은 매우 비싸다는 것을 명심해야 한다. 또한 모든 게임 물체에 모든 업데이트 단계가 필요한 것은 아니다. 특정 단계가 필요없는 물체에 반복을 실시하는 것은 CPU 대역폭의 낭비일 뿐이다. 반복 비용을 줄이는 방법 중 하나는 게임 물체의 다중 링크 목록을 유지하는 것이다. 링크 목록 하나는 각 업데이트 단계를 맡는다. 특정 물체가 이러한 업데이트 단계에 포함되어야 할 경우 이 물체는 해당되는 링크 목록에 자신을 추가시킨다. 이렇게 하면 특정 업데이트 단계에 관심이 없는 물체를 반복하는 것을 피할 수 있다.

---

#### 14.6.3.2. 버킷 업데이트

물체 상호간 의존성이 있는 상태에서 위에 기술한 단계적인 업데이트 기술은 약간 조정되어야 한다. 물체 상호간 의존은 업데이트 순서를 규정하는 규칙과 충돌할 수 있기 때문이다.

예를 들어 물체 B가 물체 A에 붙들려 있는 상황을 생각해 보라. 더 나아가서 물체 A를 최종 월드 공간 포즈와 매트릭스 팔레트까지 완전히 업데이트 한 다음에만 물체 B를 업데이트할 수 있다고 가정해보자. 이는 애니메이션 시스템에 최대의 처리율을 보장하기 위한 모든 게임 물체의 배치 애니메이션 업데이트 필요성과 충돌한다.

물체 상호간 의존성은 의존성 트리의 숲으로 시각화할 수 있다. 게임 물체는 이 숲의 뿌리를 대표하는 상위가 없다. 따라서다른 어떤 물체에도 의존성이 없다. 물체는 이 숲의 나무 속의 첫 하위분류에 있는 이러한 뿌리 물체에게 직접 의존한다. 첫 하위분류에 의존하는 물체는 두 번째 하위분류가 되는 식이다. 이는 그림 14.14.에 소개되어 있다.

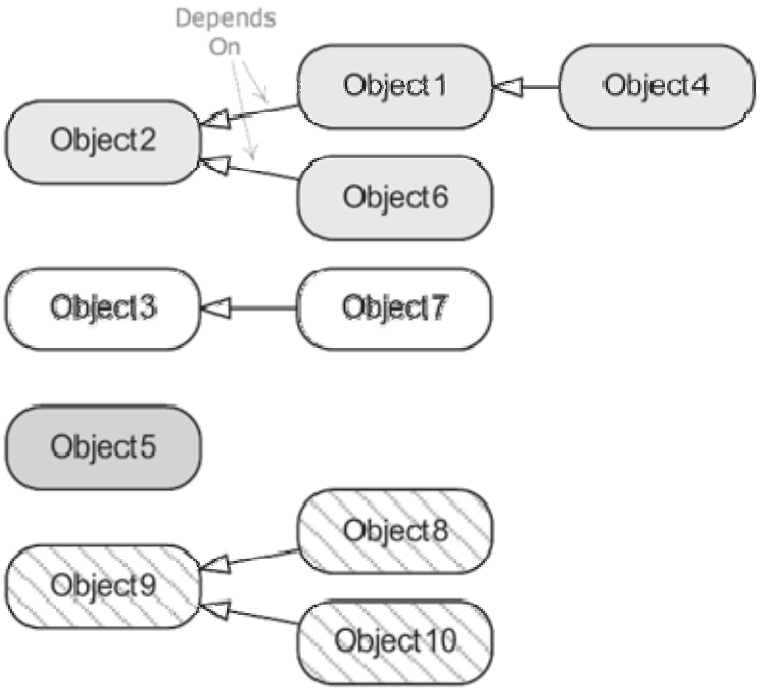


그림 14.14. 물체 상호간 업데이트 순서 의존성은 의존성 트리의 숲으로 나타낼 수 있다.

업데이트 순서 요청 충돌 문제를 해결하는 해결책 중 하나는 물체를 독립된 그룹 안으로 모으는

것이다. 더욱 마땅한 이름이 없는 관계로 이를 버킷이라고 부르기로 한다. 첫 번째 버킷은 숲의 모든 뿌리 물체를 포함한다. 두 번째 버킷은 모든 첫 번째 하위분류를 포함한다. 세 번째 버킷은 모든 두 번째 하위분류를 포함한다. 뭐 이런 식이다. 각 버킷에서 게임 물체와 게임 엔진에 대한 완벽한 업데이트를 실행하고 모든 업데이트 단계를 완성한다. 그 다음 버킷이 남지 않을 때까지 각 버킷에 대한 절차 전체를 반복한다.

이론상으로 의존성 숲의 트리의 깊이는 제한이 없다. 그러나 이 깊이는 실제로는 얕다. 예를 들어 캐릭터에게 무기를 쥐어줄 수 있다. 그리고 이 캐릭터를 움직이는 플랫폼이나 차량에 태울 수도 있다. 이를 구현하기 위해서는 의존성 숲에 3개의 단계, 그리고 3개의 버킷만 설치하면 될 뿐이다. 하나는 플랫폼 또는 차량, 또 하나는 캐릭터, 또 하나는 캐릭터의 손에 들린 무기에게 배정하면 된다. 많은 게임 엔진은 의존성 숲의 깊이를 노골적으로 제한함으로서 정해진 수의 버킷만 사용할 수 있게 한다. 일단 이들이 모두 버킷 접근을 사용한다고 가정하자. 물론 게임 루프를 세우는 데는 그 외에도 여러 방법이 있다.

버킷, 단계, 배치 처리가 된 업데이트 루프는 이런 모습을 하고 있다.

```
void UpdateBucket(Bucket bucket)
{

// ...

for (each gameObject in bucket)
{

gameObject.PreAnimUpdate(dt);

}

g_animationEngine.CalculateIntermediatePoses

(bucket, dt);


for (each gameObject in bucket)
{
```

```
gameObject.PostAnimUpdate(dt);

}

g_ragdollSystem.ApplySkeletonsToRagDolls(bucket);
g_physicsEngine.Simulate(bucket, dt);
// runs
// ragdolls too
g_collisionEngine.DetectAndResolveCollisions

(bucket, dt);


g_ragdollSystem.ApplyRagDollsToSkeletons(bucket);
g_animationEngine.FinalizePoseAndMatrixPalette

(bucket);


for (each gameObject in bucket)
{

gameObject.FinalUpdate(dt);

}
// ...

}
void RunGameLoop()

{

while (true)
{

// ...
```

```
UpdateBucket(g_bucketVehiclesAndPlatforms);

UpdateBucket(g_bucketCharacters);

UpdateBucket(g_bucketAttachedObjects);

// ...

g_renderingEngine.RenderSceneAndSwapBuffers();

}

}
```

실제로는 이것보다는 좀 더 복잡하다. 예를 들어 일부 엔진 하위체계는 버킷 개념을 지원하지 않는 물리 엔진일수도 있다. 이는 이들이 타사 SDK 또는 버킷 방식으로 업데이트되지 않기 때문이다. 그러나 이 버킷 업데이트는 기본적으로 노티 독에서 <언차티드>를 구현할 때 사용하던 것과 같으며 차기 타이틀인 <언차티드 2>에도 또 쓰이고 있다. 이것은 실용적이고 타당한 수단이기 때문이다.

---

#### 14.6.3.3. 물체 상태 불일치와 원 프레임 오프 래그

게임 물체 업데이트를 다시 살펴보자. 그러나 이번에는 각 물체의 현지 시간 관념의 기준으로 생각해보자. 14.6절에서 시간 t의 게임 물체 상태 i는 상태 벡터 Si(t)로 표시할 수 있다고 말했다. 게임 물체를 업데이트할 때는 이전의 상태 벡터 Si(t1)를 새로운 현재 상태 벡터 Si(t2)로 바꾸게 된다(t2의 위치는 t1 + Δt이다).

이론적으로는 그림 14.15에 나타난 것처럼 모든 게임 물체의 상태는 시간 t1에서 t2로 순식간에 병렬적으로 바뀌게 된다. 그러나 실제로는 물체를 한 번에 하나씩 업데이트할 수 밖에 없다. 모든 게임 물체를 루프하고 차례대로 하나씩 업데이트 함수를 호출할 수 밖에 없는 것이다. 이 업데이트 루프 중도에 프로그램을 멈추려면, 게임 물체의 절반의 상태가 Si(t2)로 업데이트되고,

나머지 반은 이전 상태인 Si(t1)에 머물러 있게 된다. 이는 현 시간부로 업데이트 루프에 있는 게임 물체들 중 두 개를 호출할 때 될 수도 있고 안 될 수도 있다는 것이다. 더구나 업데이트 루프를 인터럽트하는 정확한 위치에 따라 물체는 부분적으로 업데이트된 상태에 머무를 수도 있다. 예를 들어 애니메이션 포즈 블렌딩이 실행 중이지만 물리 및 충돌 해결은 적용되지 않은 상태이다. 이 때문에 다음 규칙이 나온다.

모든 게임 물체의 상태는 업데이트 루프 전이나 후나 일관성이 있다. 그러나 업데이트 루프 도중에는 일관성이 없을 수도 있다.

이는 그림 14.16에 표현되어 있다.

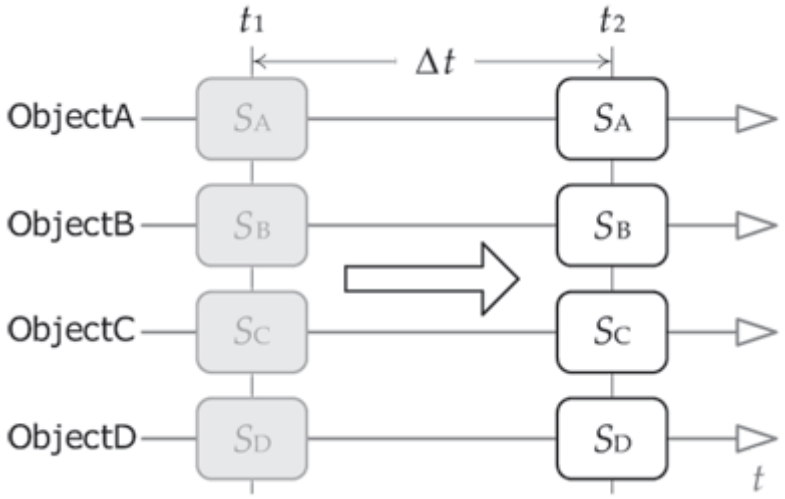


그림 14.15. 이론적으로 게임 루프의 반복 도중 모든 게임 물체의 상태는 즉시 병렬적으로 업데이트된다.

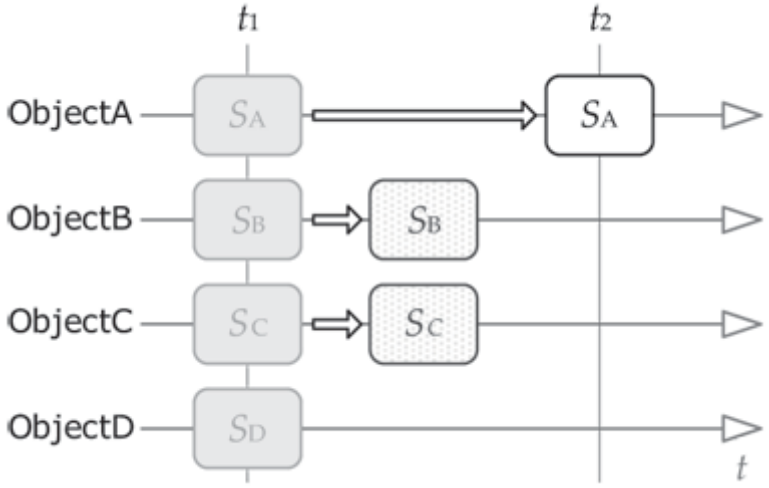

그림 14.16. 실제로는 게임 물체의 상태는 하나씩 업데이트된다. 이는 업데이트 루프의 임의의 순간에 일부 물체는 현재 시각을 t2로 생각하는데 반해 일부 물체는 현재 시각을 여전히 t1으로 인식한다는 것이다. 일부 물체는 부분적으로만 업데이트될 수도 있다. 따라서 이들의 상태는 내부적으로 불일치를 나타내게 된다. 실제로는 한 물체의 상태는 t1과 t2 사이의 한 점에 있게 된다.

업데이트 루프 중의 게임 물체 상태 불일치는 심지어 게임 업계의 프로들에게까지도 혼란과 버그를 일으키는 주원인이다. 가장 큰 문제는 게임 물체가 업데이트 루프 도중 다른 물체에 상태 정보를 문의할 때, 즉 두 물체 간에 의존관계가 있을 때 빈번하게 생긴다. 예를 들어 물체 B가 시간 t에서 자신의 속도를 구하기 위해 물체 A의 속도를 보았다고 하자. 이 때 프로그래머는 물체 A의 이전 상태인 SA(t1)를 읽을 것인지, 또는 새로운 상태인 SA(t2)를 읽을지를 확정해야 한다. 새로운 상태가 필요하지만 물체 A가 업데이트되지 않았다면 업데이트 순서 문제를 겪게 되고, 이는 원 프레임 오프 랙이라는 버그를 일으키게 된다. 이러한 유형의 버그에서는 물체의 상태가 옆 물체의 상태보다 한 프레임이 늦게 된다. 이는 스크린 상에 게임 물체간의 동조 부조로 나타나게 된다.

#### 14.6.3.4. 물체 상태 캐싱

위에서 설명했다시피 이 문제에 대한 해결책 중 하나로는 게임 물체들을 버킷 안에 그룹으로 묶는 것이다 (14.6.3.2절 참조). 단순한 버킷 업데이트 접근법의 문제는 게임 물체가 상태 정보를 문의하는 데 임의의 제한을 부과한다는 데에 있다. 게임 물체 A가 다른 물체 B의 업데이트된 상태 벡터 SB(t2)를 구하려면 물체 B는 이미 업데이트된 버킷 안에 있어야 한다. 이와 마찬가지로 물체 A가 물체 B의 이전 상태 SB(t1)을 구하려면 물체 B는 아직 업데이트되지 않은 버킷 안에 있어야 한다. 또한 물체 A는 같은 버킷 안에 있는 물체의 상태 벡터를 문의할 수 없다. 앞에서도 살펴보았듯이 이들의 상태 벡터는 부분적으로 업데이트되었기 때문이다.

일관성을 높이는 방법으로 각 게임 물체를 업데이트 되는 동안 적소에 겹쳐쓰지 않고, 새로운 상태 벡터 Si(t2) 를 계산하는 동안 이전 상태 벡터 Si(t1) 를 캐시에 입력하기 위해 각 게임 물체를 배열하는 방식이 있다. 이는 두 가지 즉각적인 장점이 있다. 우선 어떤 물체라도 업데이트 순서에 상관없이 다른 물체의 이전 상태 벡터를 안전하게 문의할 수 있다. 그리고 두 번째로 완벽히 일관적인 상태 벡터 (Si(t1))를 언제라도 사용가능하다. 심지어는 새로운 상태 벡터가 업데이트 중일 때도 사용 가능하다. 내가 알기로는 이 기술을 가리키는 표준 용어는 없다. 따라서 더 좋은 이름이 떠오르지 않으면 나는 이를 상태 캐싱이라고 부르겠다.,

상태 캐싱의 또다른 이점은 두 시각 사이의 임의의 순간의 상태의 근사값을 계산하기 위해 이전 상태와 다음 상태 사이에 선형적 보간이 가능하다는 점이다. <더 하보크>의 물리 엔진은 이 목

적을 위해 시뮬레이션 내의 모든 강체의 이전 상태와 현재 상태를 유지한다.

상태 캐싱의 단점은 적소 업데이트 접근법에 비해 두 배의 메모리를 소모한다는 것이다. 또한 문제를 절반밖에 해결하지 못한다. T1시점의 이전 상태는 완벽히 일관성이 있는데 비해 t2의 새로운 상태는 여전히 불일치 상태일 수 있기 때문이다. 그렇긴 하지만 이 기술도 신중하게 사용되면 유용할 수 있다.

---

#### 14.6.3.5. 타임 스탬핑

게임 물체 상태의 일관성을 개선시키는 쉽고 저렴한 방법은 타임 스탬프를 찍어주는 것이다. 그러면 게임 물체의 상태 벡터를 이전 시간 설정 또는 현재 시간 설정에 맞추는 것을 결정하는 문제는 하찮은 것이 된다. 업데이트 루프 동안 다른 게임 물체의 상태를 문의하는  코드는 모두 타당한 상태 정보가 획득되었음을 확인하기 위해 타임 스탬프를 정하거나 분명히 확인할 수 있다.

타임 스탬핑은 버킷 업데이트 중의 상태 불일치를 해결해 줄 수는 없다. 그러나 어떤 버킷이 현재 업데이트 중인지를 반영해주는 글로벌 변수 또는 상태 변수를 설정할 수는 있다. 아마 모든 게임 물체는 자신이 어떤 버킷에 있는지 알고 있을 것이다. 따라서 우리는 문의된 게임 물체가 속한 버킷을 검사해 현재 업데이트된 버킷에 대조하고 불일치 상태 문의로부터 지키기 위해 이들이 같지 않음을 단언할 수 있다.

### 14.6.4.  병렬성 디자인

7.6절에서, 게임 엔진이 최근 게임 하드웨어의 표준인 병렬 처리 자원의 이점을 이용할 수 있는 여러 방법을 소개했다. 그렇다면 병렬점은 게임 물체 상태 업데이트에 어떤 방식으로 영향을 미치고 있는 것일까?

#### 14.6.4.1. 게임 물체 모델의 병렬화

게임 물체 모델은 병렬화하기가 매우 어렵다. 그 이유는 몇 가지 있다. 게임 물체는 다른 물체 및 무수한 엔진 하위체계에서 사용하거나 생성한 데이터와의 상호의존성이 매우 높기 때문이다. 게임 물체는 서로 소통하며 업데이트 루프 도중에도 여러 번 소통하는 경우가 있다. 그리고 이러한 소통 양상은 예측이 불가능하며 플레이어의 입력내용과 게임 세계의 이벤트에 따라 큰 영

향을 받는다. 때문에 다양한 스레드 내의 게임 물체를 업데이트하는 과정은 매우 어렵다. 이는 예를 들자면 성능상 물체 상호간 소통을 지원하기 위한 스레드 동조의 양을 지원하지 못하기 때문이다. 그리고 외부 게임 물체의 상태 벡터를 훔쳐봄으로서 업데이트를 위한 부프로세서의 고립된 메모리에 대한 게임 물체의 직접 메모리 접근이 불가능하다.

즉 게임 물체 업데이트는 이론상으로는 병렬적으로 실행될 수 있다는 얘기다. 이를 실현하기 위해서는 전체 물체 모델을 신경써서 디자인하여 게임 물체가 다른 게임 물체의 상태 벡터를 훔쳐보지 못하게 해야 한다. 모든 물체 상호간 소통은 메시지 패싱을 통해 이루어져야 하며, 게임 물체 간, 심지어는 완전히 분리된 메모리 공간 내에 있는 물체 간이나 물리적으로 다른 CPU 코어 사이에 처리되는 메시지를 패싱하는 효율적인 시스템이 필요하다. 게임 물체 모델을 코딩하기 위해 에릭슨의 얼랑(http://www.erlang.org)과 같은 분산 프로그래밍 언어를 사용한 연구가 여러 건 이루어졌다. 이러한 언어는 병렬 프로세싱과 메시지 패싱, 스레드 간의 컨텍스트 스위칭 처리를 C나 C++보다 더욱 효율적이고 신속하게 하기 위한 내장 지원을 제공한다. 또한 프로그래머들이 규칙을 깨지 않도록 프로그래밍 숙어를 제공함으로서, 동시, 분산, 다중 에이전트 디자인을 가능케 하여 함수 처리를 타당하고 효율적으로 실시할 수 있다.

#### 14.6.4.2. 동시 엔진 하위 체계를 사용한 인터페이싱

섬세하고 동시적이며 분산된 물체 모델은 이론적으로는 타당하며 매우 흥미로운 연구의 영역이 되고 있다. 그러나 현재로서는 대부분의 게임 팀은 이런 모델을 사용하지 않는다. 대부분의 게임 팀들은 대신 물체 모델을 단일 스레드에 남겨두고 구식 게임루프를 사용하여 이들을 업데이트한다. 이들의 관심은 게임 물체가 의존하는 다수의 저레벨 엔진 체계를 병렬화하는 데에 집중되어 있다. 그래야 개발팀은 지출 대비 최고의 효과를 얻을 수 있다. 저레벨 엔진 하위체계는 게임 물체 모델보다 성능이 더욱 중요하기 때문이다. 이는 게임 물체에 사용되는 CPU 파워가 적은 반면 저레벨 하위체계는 각 프레임마다 대량의 데이터를 처리할 수 있기 때문이다. 이는 현장에서 80-20의 법칙이 적용되는 사례라고 하겠다.

물론 단일 스레드 게임 물체 모델을 사용한다고 게임 프로그래머들이 병렬 프로그래밍 문제를 완전히 의식하지 못하는 것은 아니다. 물체 모델은 그들과 동시에 실행되는 엔진 하위체계와 계속 상호작용을 해야 한다. 이 패러다임 전환으로 인해 게임 프로그래머들은 병렬화 시대 이전에 잘 쓰던 특정 프로그래밍 패러다임을 피하고, 그 자리에 새로운 패러다임을 받아들일 수 밖에 없게 되었다.

아마도 게임 프로그래머들이 받아들여야 할 가장 중요한 변화는 비동기적인 사고방식일 것이다. 7.6.5절에서 설명했듯이 이는 게임 물체가 시간이 걸리는 동작 수행을 요청할 때 차단 함수 호출을 막아야 한다. 차단 함수란 호출 스레드 컨텍스트에서 직접 동작하는 함수로, 작업이 완료

될 때까지 스레드를 차단한다. 대신 가능할 때면 언제든지 크고 비용이 많이 드는 작업은 비차단함수 호출을 통해 요청해야 한다. 비차단함수는 다른 스레드, 코어, 프로세서에서 실행될 요청을 전송하고 즉시 호출 함수의 제어로 복귀하는 함수이다. 원래의 물체가 요청 결과를 기다리는 동안 다른 게임 물체를 업데이트하는 등 상관없는 작업과 함께 주 게임 루프를 진행할 수 있다. 나중에 같은 프레임 내에서, 또는 다음 프레임에서 게임 물체는 요청의 결과를 받아 사용할 수 있다.

배치 역시 게임 프로그래머들의 사고방식을 전환시켰다. 14.6.2절에서 언급했다시피 유사한 임무를 배치에 모아 그들을 한꺼번에 실행하는 것이 각 임무를 별도로 실행하는 것보다 더욱 효율적이다. 이는 게임 물체 상태 업데이트에도 적용할 수 있다. 예를 들어 한 게임 물체가 여러 가지 용도로 충돌 월드에 100가지 빛을 쏘아주기를 요청한다고 하자. 이 요청이 대기열을 이루어 하나의 큰 배치로 실행되는 것이 가장 좋을 것이다. 기존의 게임 엔진을 병렬형으로 개장할 수 있다면 코드 역시 요청을 개별적으로 수행하는 것이 아니라, 배치를 구성해 수행하기 위해 빈번하게 다시 쓰여져야 할 것이다.

동기적 사고를 전환하는 데 특히 까다로운 측면은 게임 루프(a)가 요청을 시작하고 (b) 결과를 받아 응용하기를 기다리는 동안 배치되지 않은 코드를 비동기적이고 배치된 접근방식으로 사용하는 것을 결정하는 것이다. 이렇게 하는 동안 스스로에게 다음과 같은 질문을 자문해보면 유용할 것이다.

이 요청을 얼마나 일찍 시작할 수 있는가? 일찍 시작할수록 결과가 필요할 때 맞춰 받아볼 확률이 높아진다. 그리고 이는 주 스레드를 쓸데없이 비동기적 요청이 완료되는 동안 쓸데없이 대기시키지 않음으로서 CPU 활용을 극대화할 수 있다. 따라서 어떤 요청을 받더라도 프레임 내에서 가장 빨리 충분한 정보를 얻어 시작할 시점을 결정하고 나서 시작해야 한다.
또한 이 요청의 결과를 얼마나 오래 기다릴 수 있는가? 아마도 늦어도 후반 작업이 시작될 때까지는 업데이트 루프 내에서 기다릴 수 있을 것이다. 아마도 한 프레임 정도 랙이 생기는 것은 용인할 수 있으며 마지막 프레임의 결과를 사용해 이 프레임의 물체 상태를 업데이트할 수 있을 것이다(AI같은 일부 하위 체계는 몇 초마다 한번씩 업데이트되므로 더 긴 랙 타임도 용인할 수 있다). 짧은 생각, 약간의 코드 리팩터링, 중간 데이터의 일부 추가 캐싱을 감안하면 요청 결과를 사용하는 코드는 여러 환경에서 프레임 후반부까지 연기될 수 있다.