프로그래밍 반응성
(Programming Responsiveness)

Mick West

가마수트라 등록일(2008. 7. 9)

(http://www.gamasutra.com/view/feature/1942/programming_responsiveness.php)

[게임에서 자신의 행동을 제어할 수 없다면, 게임을 탓해야 하는가? 한 기술 관련 기사에서, Neversoft 의 공동 설립자인 Mick West 는 게임에서의 반응 지연 문제를 고찰하면서, 여러 가지 가능한 해결책들을 제시하였다.]

반응성은 게임의 첫인상을 심어줄 수도 있고 망쳐놓을 수도 있는 요인이다. 특히 반응성이 낮은 게임의 “진행이 부진하고,” “반응이 느리고,” 얕고,” “밋밋”하다는 평을 받을 경우에는 더욱 그러하다. 더 나은 게임이라면 "빈틈없는,” 또는 “감응하는” 게임으로 불릴 것이다.

반응성에 기여하는 몇몇 요소들이 있는데, 본 기사는 프로그래머의 관점에서 그 중 몇 가지를 살펴보면서 게임의 반응성을 향상시키기 위한 방법들을 제시할 것이다.

반응 지연

반응 지연(Response lag)은 이벤트를 촉발시키는 플레이어와 피드백(보통은 비주얼)을 받는 플레이어 간의 지연을 말한다. 지연이 너무 길어지면, 게임의 반응이 느린 것으로 느껴진다. 일부 요소들이 이러한 반응 지연의 길이를 향상시킬 수 있다.

게임의 반응이 느릴 경우, 대 여섯 가지의 요인들이 함께 작용한 결과인 경우가 많은데, 한 가지 요인을 조정하는 것만으로는 차이를 인식하지 못할 수도 있으므로, 이 요인들 모두를 다루어, 두드러진 향상을 도모해야 한다.

플레이어나 때로는 설계자들도 게임을 제어하면서 무엇인가 잘못되었다고 느끼지만 이를 말로 표현하기가 어려울 때가 있다. 종종 동기화해야 할 필요를 느끼면서도 그렇게 하지 못하고, “이벤트가 입력 후 0.10 초 만에 일어났다”고 말하는 대신 게임이 “느리거나” “빈틈이 있고,” “어렵다”고 말하곤 한다.

또는 자세하게 말하지 못하고, 그저 게임이 형편없다고 말하면서도 정작 왜 형편없는지는 이해하지 못한다.

비록 테스트 플레이어들이 직접적으로 지적하지는 못한다 하더라도, 설계자들과 프로그래머들은 반응 지연과 그것이 게임에 미치는 부정적인 영향에 대해 알아야 한다.

지연이 일어나는 이유

지연이 발생하는 이유를 이해하기 위해서는 사용자가 버튼을 누르는 시점부터 결과가 화면에 표시되는 시점까지 발생하는 일련의 이벤트들을 이해해야 한다. 그러기 위해서는 게임의 메인 루프 구조를 살펴볼 필요가 있다. 메인 루프는 두 가지 테스크, 곧 로직과 렌더링의 기능을 수행한다.

메인 루프의 로직 부분은 게임 상태를 업데이트하는 한편(게임 객체와 환경의 내부적 표시), 렌더링 부분은 텔레비전 상에 표시되는 프레임을 만든다.

메인 루프의 어느 지점, 보통은 시작 지점에서 사용자로부터의 입력을 받게 되는데, 이것이 메인 루프에서는 때때로 제 3 의 테스크로 여겨지기도 하지만, 보통은 로직 테스크의 일부로 볼 수 있다. 여기서 이를 별도로 다루는 이유는 사물이 발생하는 순서를 고려함에 있어 그 중요성 때문이다.

메인 루프를 체계화할 수 있는 몇몇 방법들이 있다. 가장 간단한 방법이 목록 1 에 제시되어 있는데, 여기서 로직과 렌더링 코드를 단순히 번갈아 가며 호출하는 것이다. 이 때 우리는 일부 프레임 동기화가 Rendering()을 호출할 때에 발생할 것이며, NTSC 콘솔 게임의 경우 보통 60fps 또는 30 fps 의 고정 프레임 비율로 실행할 것으로 예상한다.

목록: 가장 단순한 메인 루프

```
while (1) {
Input();
Logic();
Rendering();
}
```

여기서 메인 루프는 그 일부일 뿐이다. Rendering()의 호출은 렌더링 테스크의 CPU 측면의 작업으로서, 환경과 객체를 반복하고, 변환 요소들을 모으고, 작용하게 하고, 설정하며, GPU 가 실행할 표시 목록을 만드는 역할을 한다.

액츄얼 GPU 렌더링은 CPU 렌더링 이후에 수행되며, 보통은 비동시성이다. 때문에 메인 루프가 다음 프레임을 처리하는 동안, GPU 는 아직도 이전 프레임을 렌더링하고 있는 것이다.

그러면 언제 지연이 있게 되는가? 지연을 발생시키는 요인들을 이해하기 위해서, 사용자가 버튼을 누르는 시점부터 버튼 누름에 대한 피드백까지의 과정에 발생하는 일련의 이벤트들을 이해하지 않으면 안 된다.

최상위 레벨에서, 사용자는 버튼을 누르고, 게임 로직은 버튼 누름을 판독하고 게임 상태를 업데이트한다. CPU 의 렌더링 기능은 이 새로운 게임 상태로 프레임을 설정하고, 그러면 GPU 는 이를 렌더링한다. 최종적으로 새 프레임이 화면 상에 표시된다.

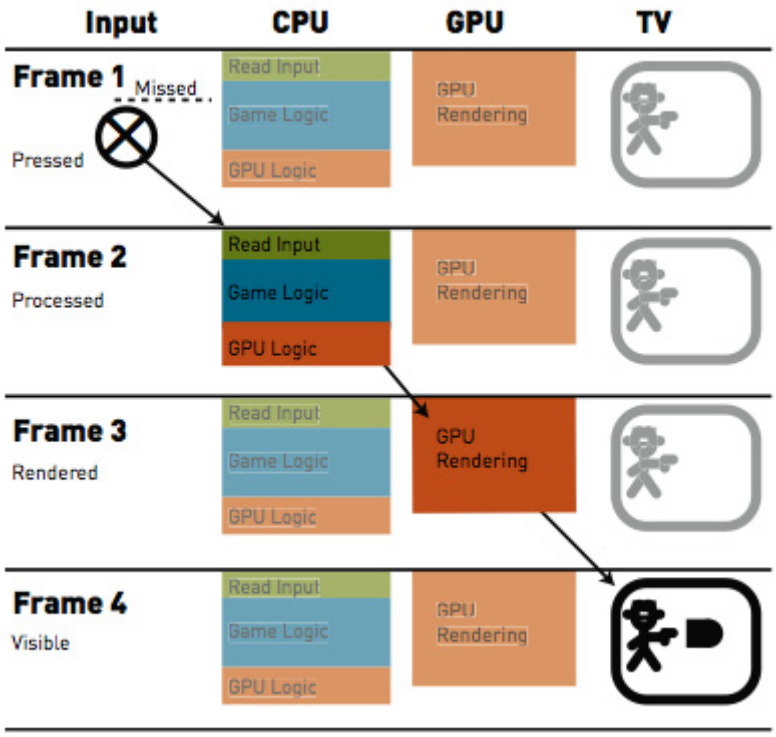

그림 1: 플레이어가버튼을 누를 때, 게임은 세 개의 프레임을 구성하여(최적의 경우) 비주얼 비드백을 만들고, 프로그래밍 문제들이 추가로 프레임 지연을 야기할 수 있다. 액츄얼 지연 시간을 단일 게임 프레임의 길이로 곱한다.

그림 1 은 이 순서를 그래픽으로 보여준다. 프레임 1 의 특정 지점에서 플레이어는 총을 쏘기 위해서 버튼을 누른다. 이 입력값의 처리가 이미 해당 프레임에서 수행되었기 때문에, 이 입력은 프레임 2 에서 판독된다. 프레임 2 는 버튼 누름에 기초해 이 로직 상태를 업데이트한다(총이 발사된다).

프레임 2 에서는 또한 렌더링의 CPU 측면이 이 새로운 로직 상태대로 수행되고, 그런 다음 프레임 3 에서는 GPU 가 이 새로운 로직 상태의 액츄얼 렌더링을 수행한다. 최종적으로 프레임 4 가 시작할 때, 프레임 버퍼를 플립(flipping)함으로써 새롭게 렌더링한 프레임이 표시된다.

그러면 이 지연은 얼마나 긴가? 프레임이 얼마나 긴지에 따라 달라진다(여기서 "프레임"은 메인 루프의 전체 반복 길이를 말한다). 사용자의 입력이 비주얼 피드백에 해석되기까지는 세 개의 프레임이 필요하다.

만일 30fps 에서 실행하고 있다면, 지연은 30 분의 3 즉 초당 10 분의 1 이 될 것이다. 60fps 에서 실행하고 있다면, 지연은 60 분의 3, 곧 초당 20 분의 1 이 될 것이다. 이 계산은 60fps 와 30fps 게임 간의 차이에 대한 일반적인 오해를 예시해 준다. 이 두 프레임 비율 간의 차이는 단지 초당 60 분의 1 이므로, 사람들은 반응성의 차이 역시 60 분의 1 일 것으로 생각하는 것이다.

그러나 사실 60 에서 30 까지는 지연에 아무런 vsync 도 추가하지 않는다. 이는 그저 곱함수로 작용하여, 지연을 일으키는 처리 경로를 두 배로 만들 뿐이다. 그림 1 에서의 예에서, 초당 60 분의 1 이 아닌, 60 분의 3 을 추가한다. 이벤트 경로가 더 길 경우, 그보다 더 많이 추가할 수도 있다.

그림 1 은 가장 가능성이 높은 이벤트 순서를 예시하고 있다. 버튼 누름 이벤트는 가장 짧은 경로를 통해서 비주얼 피드백으로 해석되며, 이벤트 순서에서 이를 분명히 볼 수 있다.

프로그래머로서, 사건이 발생하는 순서에 익숙한 지 여부는 게임에서 사건이 왜 그렇게 작용하는지를 이해하는 데 있어 중요한 요소이다. 지연 프레임이 추가로 도입되기가 아주 쉬운데(초당 60 분의 1 또는 30 분의 1 의 지연을 의미함), 단순히 이벤트가 발생하는 순서에 면밀히 주의를 기울이지 않음으로써 그렇게 될 수 있다.

간단한 예로, 메인 루프에서 Logic()과 Rendering() 호출의 순서를 바꿀 경우 어떤 일이 발생하는지 고려해 보자. 그림 1 의 프레임 2 를 보면, 여기서 GPU 로직(렌더링)이 CPU 로직에 이어 발생하며, 따라서 프레임 2 의 시작에서 입력값이 CPU 로직과 동일 프레임 내에 GPU 로직에도 영향을 미치게 된다.

그러나 GPU 로직이 처음으로 수행될 경우, 입력값은 다음 프레임에서 추가 지연 프레임이 도입되기까지는 GPU 로직에 영향을 주지 않는다. 이는 신참들의 실수로서, 프로그래머들은 이런 일이 절대로 일어나지 않도록 주의해야 한다.

추가 지연 프레임은 게임 로직 내의 작동 순서의 결과로 보다 미묘하게 도입될 수도 있다. 제시된 예에서는, 총을 쏘는 경우가 이에 해당된다.

아마도 물리 엔진을 사용하는 영역에서 객체들을 증가시키고 이러한 업데이트로 인한 이벤트들(충돌 이벤트 등)을 처리하기 위해 엔진을 설정한다. 이러한 상황에서, 입력값 및 로직의 순서는 목록 2 와 같다.

목록 2: 물리 업데이트에 이은 이벤트 처리

```
void Logic() {
HandleInput();
UpdatePhysics();
HandleEvents();
}
```

메시지를 통한 이벤트 처리는 시스템들을 분리시키는 매우 좋은 방법이며, 프로그래머는 플레이어 콘트롤 이벤트에 대하여 이 방법을 사용하기로 결정할 수 있다. 총을 쏘는 이벤트를 위해서는, HandleInput() 함수는 총을 발포하라는 이벤트를 일으킬 것이다.

HandleEvents() 함수는 이 이벤트를 취하여 실제로 총이 발포되도록 한다. 그러나 물리적 업데이트가 이 프레임에 대해 이미 발생하였기 때문에, 이 효과는 다음 프레임까지는 통합되지 않아 추가 지연 프레임을 도입한다.

그 밖의 지연 원인들

그보다 낮은 수준의 액션 명령은 훨씬 더 많은 지연을 초래할 수 있다. 점프를 예로 들면, 그 피드백은 캐릭터가 실제로 움직이게 하는 것이다. 게임에서 무엇인가가 움직이게 만들기 위해서는 속도를 직접 설정하거나 힘을 가할 수 있는데, 가속이나 순간적인 추진력이 그 예이다.

속도 변화가 적용되기 전에 물리 엔진이 위치를 업데이트할 경우에는 이 시나리오에 문제가 생기는데, 이는 많은 게임 프로그래밍 입문서에서 공통적으로 나타나는 조건이다.

점프하는 객체의 속도는 입력 이벤트가 처리되는 것과 동일한 프레임에서 업데이트되지만, 이 객체는 사실상 다음 게임 프레임이 되기까지 이동하지 않기 때문에 여기서 추가로 지연 프레임이 발생하게 된다.

기억해야 할 것은 별개로 놓고 보면 식별하기 힘들지만, 결합했을 때의 그 영향이 게임 제어를 어렵게 만드는, 누적 요인들이 야기하는 문제점들이라는 것이다.

상기 열거한 세 가지 실수가 한꺼번에 일어났다고 가정해 보자. 로직 전에 렌더링을 하고, 물리 상태를 진행한 후에 로직 이벤트를 처리하고, 속도 전에 위치를 업데이트하였다. 이렇게 메인 루프에 3 회에 걸친 반복이 일어나고, 앞서 언급한 세 번의 지연이 일어나 버튼을 누르는 플레이어와 화면 상의 결과를 보는 플레이어 간에는 여섯 개의 지연 프레임이 나타나게 된다.

초당 60 프레임, 곧 1 초의 10 분의 일에서, 게임이 30fps 로 실행되고 있을 때, 그 지연은 1 초의 5 분의 일, 곧 200 밀리 초로 두 배 늘어나게 된다.

지연을 일으키는 다른 요인들도 있는데, 애니메이션에 의해서 이동이 발생할 경우, 속도 변화가 애니메이션의 특정 시점에 투입된다. 예를 들어, 애니메이터가 점프 임펄스(impulse)를 애니메이션에 가할 경우, 보기는 더 좋을 수 있으나, 그다지 좋은 느낌이 연출되지는 않는다.

애니메이터는 플레이어가 즉각적인 피드백을 필요로 할 때에 속도 임펄스가 애니메이션의 첫 프레임에 있도록 조정함으로써 이를 수정할 수 있다. 그러나 문제는 애니메니션을 촉발하는 것이 어떻게 실제 움직임으로 해석되는가 하는 것이다.

아마도 애니메이션의 업데이트는 Render() 함수에 의해 처리될 것이다. 애니메이션으로 촉발된 이벤트는 루프 시점까지 처리되지 않을 것이며, 이로 인해서 또 다른 프레임이 추가된다.

그에 더하여, 애니메이션을 촉발하는 것으로는 다음 프레임까지는 프레임을 진행하지 않을 수 있으며, 이로 인해 프레임의 이벤트 실행이 지연된다. 여기서 프레임 여섯 개에서 여덟 개까지 지연이 증가할 수 있으며, 초당 60 프레임에서조차 플레이 하기 힘든 수준이 된다.

그 외에도 추가 지연 프레임이 게임이 삽입될 수 있는 경우는 다양하다. 별도의 쓰레드(또는 물리 처리 단위)에 물리 과정을 파이프라이닝(pipelining)할 수도 있다.

프레임 속도를 부드럽게 하기 위해서 3 중 버퍼링을 사용하고 있다면 어떠한가? 시스템에서 몇 차례 발생하는 추상 이벤트를 사용하여 실제 이벤트로 변형시킬 수 있다. 이벤트를 기다리는 방식으로 추가 프레임을 덧붙이는 스크립트 언어를 사용할 수도 있을 것이다.

다양한 시간 및 이벤트 개념을 제거함으로써 게임 로직을 상당히 유동적이게 만들 수도 있으나, 이렇게 할 경우, 프로그래머는 정확성을 잃게 되어 추가 지연 프레임이 발생하기가 훨씬 더 쉬워질 수도 있다.

반응성, 반응 시간과 다름

반응성과 관련된 가장 큰 오해 중 하나는 이것이 인간의 반응 시간과 관련되어 있다는 생각이다. 인간은 물리적으로 비주얼 자극에 반응하여 10 분의 1 초 내로 손가락을 움직일 수가 없다.

게임 플레이어의 최고 반응 시간은 0.15 초에서 0.30 초 사이로 다양하며, 이는 "침착성" 여부에 따라 달라진다. 게임 반응성에 대해 논할 때에 흔히 정량화 문제가 제기되는데, 그 연관성은 유의미하지 않다.

이는 플레이어가 게임에 얼마나 빨리 반응하는가가 아니라, 게임이 플레이어에게 얼마나 빨리 반응하는가의 문제이다. 이것은 반응 시간이 아니라, 동기화의 문제인 것이다.

*Guitar Hero*을 예로 들면, 기호들이 제시될 때 정확한 시점에 정확한 버튼을 눌러야 한다(목표 객체는 특정 지역 내에 있음). 앞으로 있게 될 이벤트를 예상하고 있으며, 그에 대한 아무런 반응도 없다.

반응성 부족의 문제점은 게임이 플레이어에 충분히 빠르게 반응하지 않고 목표 객체가 이벤트 발생 시점에 목표 지역 이외의 곳으로 이동할 때 발생한다.

정확한 시점에 버튼을 누르면, 객체가 폭발하기 전에 단 몇 픽셀 이상도 더 움직이지 않을 것으로 예상할 것이다. 그러나 객체가 프레임 당 적어도 몇 픽셀 정도는 이동하기 때문에, 단 몇 개의 지연 프레임만으로도 객체는 목표 밖으로 멀어질 수 있다.

많은 액션 게임들이 예상과 버튼 누름 패턴에 기반해 있다. 스케이트보드 게임에서는 레일의 끝 지점에 닿기 직전에 점프해야 한다. 1 인칭 슈팅게임에서, 누군가 총 앞에서 이동하는 순간 발사할 것이다.

다시 한번 강조하지만, 이것은 반응 시간의 문제가 아니다. 보통은 적어도 발사하기 0.5 초 전에는 목표물을 볼 것이며, 총을 움직이거나, 총 앞으로 목표물이 이동할 때까지 기다릴 것이다.

반응성에 대한 이러한 문제의 본질은 직관적으로 파악하기가 어렵기 때문에, 프로그래머로서 이 문제들을 충분히 이해하는 것이 중요하다. 가장 중요한 것은 비주얼 피드백 이전에 버튼을 누르는 행동의 로직과 렌더링을 통해 프레임별 경로를 명확하게 기술할 수 있어야 한다는 점이다. 일단 이것이 가능해지면, 가능한 최적의 경로에 가깝게 조절할 수 있다.

*[**편집자 각주**: 본 기사는 본래 게임 개발자 잡지에 간행된 것으로서, Gamasutra의 편집자들이 커뮤니티에 가치가 있다는 판단 하에, 독자적으로 선별한 것이다. 본 기사는 Intel's Visual Computing microsite의 플랫폼 및 벤더 독리형으로 Intel 사에 의해서 간행이 허용되었다.]*