스마트폰 퍼포먼스 문제 해결

(Solving Smartphone Performance Problems)

작성자: 이테이 둡데바니 (Itay Duvdevani)

작성일: 2012년 10월 3일

스마트폰 게임에 퍼포먼스 이슈가 있는가? 모미니스(MoMinis)의 프로그래머 이테이 둡데바니가 동료들과 함께 아주 단순한 소팅 문제를 창조적으로 해결한 방법을 설명하고, 그 과정에서 스마트폰 게임 엔진을 업그레이드한 기술적인 방법을 소개한다.

모미니스 플랫폼(MoMinis Platform)의 폰 관련 작업은 재미있는 일이다. 우리 콘텐츠 개발자들은 게임 메카닉, 버그, 퍼포먼스 튜닝 등을 시뮬레이터상에서 하는 경우가 대부분이어서, 새 타이틀을 실제 폰에서 테스트할 때가 되면 항상 약간 긴장하게 된다.

게임을 컴파일 하게 되면 게임의 로직을 깨뜨리는 버그를 발견할 수도 있고(시뮬레이터나 컴파일러에서), 시뮬레이터에서 제대로 시뮬레이팅되지 않은 퍼포먼스 문제를 찾아낼 수도 있다.

게임이 시뮬레이터에서는 작동했는데 폰에서는 작동하지 않을 때는 바로 우리 팀으로 넘어온다. 문제를 정확히 찾아낼 수 있는 이는 컴파일러 개발자인 우리들뿐이기 때문이다.

<젤리 점프>의 문제

우리 컨텐츠 개발자들은 새로운 타이틀 <젤리 점프(Jelly Jump)>를 만들고 있었다. 몇 주 동안 개발을 마치고 실제 폰에서 테스트를 시작할 수 있을 정도의 완성상태였다. 릴리즈까지는 일주일이 남아있었다. 고사양의 폰에서는 무척 재미있고 부드럽게 잘 실행되었다. 그런데 중간 사양이나 저사양의 디바이스에서는 화면이 불규칙적으로 깨져서(jitter) 게임을 플레이 할 수가 없었다. 우리 팀은 이 문제를 해결하기 위해 나섰다.

이 문제는 비교적 성능이 괜찮은 디바이스(800MHz 스냅드래곤, 아드레노 200GPU)에서 발견되었다. FPS가 낮아서 생기는 문제는 아니었지만, FPS가 몇 초에 한 번씩 흐트러졌고 무작위로 발생하는 것 같았다. 게임이 40-50FPS로 실행되다가 1초 정도 10-5FPS로 떨어졌다가 다시 계속되는 식이었다.

우리는 컨텐츠 개발자들로부터 테스트할 때 그래픽 에셋을 제거하면 게임이 더 잘 실행되더라는 귀띔을 받았다.

컴파일 과정 중 에셋-생성 단계가 게임 에셋을 텍스쳐 아틀라스(Texture Atlas)로 할당하게 된다. 이건 자동화된 단계이기 때문에 텍스쳐 스와핑을 최소화하기 위해서 어떤 오브젝트들을 같이 그릴 것인지 몇 가지 발견적 방법(heuristic)을 통해 예측한다. 이런 발견적 방법들은 지금은 아주 기초적인 것으로 통계 분석에만 의지한다. 퍼포먼스를 위한 최적화 기능은 아직까지 없다. 우리는 에셋이 비효율적인 방식으로 그루핑되어서 특정 오브젝트를 그릴 때마다 메모리 부족으로 텍스쳐 스래싱(Thrashing)이 발생하는 것이 아닌가 의심해보았다.

이것이 문제였다. 여기에 대해서는 '마법 같은 솔루션'도 없고 어떻게 피해갈 방법조차 없었다. 꼼수도 부릴 수가 없었다. 단순히 에셋 생성 파이프라인에 비해 게임 그래픽이 너무 많았다.

'어떻게' 이 문제를 해결할 것인지 며칠을 씨름했지만 실제적인 해결책이 없었다. 우리는 가정에 대해 의심하기 시작했고 텍스쳐에도 문제가 있었는지 테스트해 보았다. 에셋 생성 파이프라인 설정에서 그래픽을 모두 ¼ 사이즈로 줄인 다음, 에셋이 모두 한 텍스쳐에 들어가도록 했다. 놀랍게도 여전히 문제가 발생했다. (그리고 되게 보기 싫었다.)

게임의 '로직(logic)'이 병목을 만든다는 뜻이었다. 우리는 달빅(Dalvik) 프로파일러로

게임을 프로파일링 해보았다. 프로파일러는 FPS가 낮은 문제를 해결할 때는 매우 유용하지만 불행히도 FPS가 깨지는 문제를 해결할 때는 거의 쓸모가 없었다. 언제 깨질지를 미리 정확히 알고 있지 않는 한 말이다. 하긴 그걸 알고 있었으면 당연히 뭐가 문제인지도 알았겠지.

컨텐츠 개발자들과 같이 앉아 게임의 로직을 들은 다음, 직감에 따라 코드에 디버그 프린트를 몇 개 심어보았는데, 놀라운 결과가 나왔다. 한 로직을 반복하면서 대량의 오브젝트를 생성할 때 Z오더(Z-order) 코드가 생각과 다르게 작동하고 있었던 것이다!

Z Order를 반복하는 이유는?

우리 게임 엔진은 게임 로직을 컴파일 하는 일 말고도, 화면을 예쁘게 그리면서 터치 입력을 받아들이는 것을 책임지고 있다.

모미니스 플랫폼 같은 2D 엔진에서는 오브젝트를 Z 오더에 따라 정렬하고 가로질러(뒤에서 앞으로, 혹은 앞에서 뒤로 정렬) 터치 인풋을 받을 오브젝트와 먼저 그릴 오브젝트를 정한다.

터치 핸들러

터치 핸들러는 스크린 단에서의 터치 이벤트를 운영체제(OS)에서 받아 게임 단의 이벤트로 바꾸는 간단한 어댑터다. 따라서 내가 화면상의 특정 위치를 클릭하면, "사용자가 (x,y) 위치를 클릭했다" 대신에 그 위치에 있는 오브젝트가 논리적인 터치 다운 이벤트를 받아들이게 되는 것이다.

우리는 게임 개발자들이 낮은 수준의 터치 이벤트를 핸들링해서 어떤 오브젝트가 응답하게 해야 할 것인지 관리하지 않아도 되도록 가상의 레이어를 제공한다. 우리의 개발 플랫폼을 초보 개발자들도 사용할 수 있게 하고 번거로움을 최소로 하기 위해서는 매우 중요한 부분이다.

게임 오브젝트가 여러 개 겹쳐 쌓여있고 터치 이벤트가 여러 오브젝트가 겹쳐있는 곳으로 들어오는 건 게임 개발자 수준에서 처리하기 어려운 일일 수 있다. 그래서 우리는 터치 이벤트를 가장 위에 있는 오브젝트, 즉 손가락이 닿는 곳에 보이는 오브젝트로 전달해준다. 그뿐이다.

이 때 가장 쉬운 방식은 전체 오브젝트를 앞에서부터 순서대로 스캔해서 이벤트를 처리할 수 있는 첫 번째 것을 찾는 것이다.

렌더 큐(The Render Queue)

모미니스 게임 엔진은 안드로이드와 아이폰 모두 OpenGL ES를 바탕으로 하고 있다. 호환성 문제 때문에 우리는 OpenGL ES 1.1 API만을 사용한다.

필레이트(fill rate)를 최대화 하기 위해서 우리는 오브젝트를 투명한 것과 불투명한 것의 두 그룹으로 나눈다. 그리고 먼저 불투명한 것들은 앞에서 뒤로 그리고, 안 보일 부분들은 나중에 그리는 대신 Z 버퍼에 놔두어서 전체 레스터라이제이션(rasterization) 파이프라인을 절약한다.

불투명한 오브젝트를 먼저 다 렌더링 한 다음에 투명한 것들을 그리는데, 투명한 그룹은 알파 블렌딩을 제대로 맞추기 위해 뒤쪽에서 앞으로 그려야 한다.

플랫폼 개발자들에게 어려운 점

우리 클래스 플랫폼 개발 환경과 그 위에서 만드는 게임들이 요구하는 조건은 까다롭다.

- 프로그래머가 아닌 초보 개발자도 좋은 게임을 쉽게 만들 수 있어야 한다.
- 게임의 퍼포먼스가 좋아야 한다.
- 게임이 보기 좋아야 한다.
- 최종 게임 사이즈는 작아야 한다.

- 개발자들이 포팅할 때 문제가 가능한 적어야 한다.(없으면 더 좋다)

이런 요구 조건들은 거의 항상 충돌해서 우리 게임 엔진 개발자들을 힘들게 한다.

콘텐츠 개발자가 완전히 독립적인 게임을 만들고 있다면, 게임 로직은 금세 꽤 복잡해진다. 파워업이나 보너스나 스페셜 애니메이션과 사운드 등이 기본 게임 메카닉을 일시적으로 바꾸고, 어떤 게임에서는 사용자의 발전에 따라 다이내믹하게 레벨을 생성해야 할 수도 있다. (<젤리 점프>에서도 그랬다)

우리 플랫폼 디자이너들이 모두 숙련된 프로그래머인 것도 아니고, 게임 엔진이 실행될 때 들고나는 것을 모두 알고 있지도 못한다. 그래서 게임 로직을 때때로 좀 이상하게 만들 수도 있다. 또 플랫폼도 아직 발전하는 단계이다 보니 콘텐츠 개발자들의 관행들이 상황을 더 나쁘게 만들기도 하는데, 플랫폼 기능의 제약상 다른 대안이 없을 때가 대부분이다.

그래서 때때로 콘텐츠 개발자가 게임플레이 중 짧은 시간에 보조 오브젝트를 많이 만들거나 없애려고 하면 인스턴스 관리가 매우 어려워진다.

매번 정렬하는 문제

이 엔진은 원래 J2ME 로 작성된 것이다 보니 CLDC 에서 제공하는 방식만 사용할 수 있으며 클래스 개수나 코드 사이즈에도 제한이 있어 최적화와는 거리가 멀게 구현될 때가 있다. 모든 메소드가 동기화되어 있는 java.util.Vector 클래스에 관련된 코드가 한 예이다.

게임을 관리하는 엔터티가 모든 오브젝트 인스턴트를 Z 오더로 정렬되는 배열(array)에 담아놓는다. 오브젝트가 하나 만들어지거나 없어질 때, 또는 Z 오더가 변경될 때마다 우리는 이 오브젝트를 배열에서 빼고 (그 뒤의 엘리먼트를 모두 한 인덱스씩 당긴다) 이진검색(binary search)를 해서 맞는 위치에 다시 넣는다.(그 뒤의 엘리먼트를 모두 한 인덱스씩 민다)

오브젝트를 위해 배열을 스캔하는 것이 O(n) 오퍼레이션이므로, 엘리먼트를 넣거나 빼는 것도 다시 O(n) 오퍼레이션이고, 맞는 위치를 찾는 것은 O(log n)

오퍼레이션이다. 한 오브젝트를 위한 기본적인 액션의 실행시간은 그 시점에 존재하는 오브젝트의 수에 비례하여 늘어난다.

당연히 실제 게임은 훨씬 더 복잡하고 오브젝트도 많기 때문에 이게 문제가 되었다. <젤리 점프>는 몇 초에 한 번씩 40~60 개의 오브젝트를 만들었고, 여기에는 500~700 개의 오브젝트가 있었다. 게다가 오브젝트가 하나 만들어질 때 그 Z 오더도 Z 오더가 허용하는 최대치로 설정되었다. 하지만 곧 오브젝트의 컨스트럭터가 레이어의 Z 오더를 설정해서 엔진이 수행해야 하는 Z 오더 오퍼레이션을 두 배로 만들어버린다. 이것으로 화면 깨짐을 설명할 수 있었다. 오브젝트를 만드는 게 논리적으로 매우 긴 과정을 거쳐야 해서 FPS 를 순간적으로 떨어뜨리는 것이다.

초기 최적화 시도

우리는 이 문제를 빨리 고쳐야 했다. 이것이 게임 릴리즈를 가로막는 가장 큰 원인이었다.

처음에 우리는 보수적인 접근법을 취했다. 릴리즈가 가까우니 변경을 가능한 한 최소화하려고 했다. 우리의 결론은 컬렉션을 항상 정렬할 이유는 없다는 것이었다. 렌더링 할 때와 터치 핸들링하기 전에만 하면 되었다. 우리의 첫 번째 시도는 이 두 경우의 정렬 방식을 다르게 해보는 것이었다.

이건 오브젝트 생성의 문제라고 봐야 했다. O(n)을 O(1)으로 줄이고 새 오브젝트를 그냥 리스트 제일 끝에 붙인 다음 정렬하기 직전에 쿼리로 Z 오더를 하는 것이다. 이로써 Z 오더 수정도 절반 정도로 줄어들 것이다. 컬렉션에 새 오브젝트를 더 이상 추가하지 않고 이전처럼 움직일 것이기 때문이다.

우리는 표준 접근법을 취했고 고전적인 퀵 정렬(quicksort)를 사용해서 적시에 배열 정렬을 하려고 했다. 불행히도 게임 엔진 개발자와의 계약사항에 같은 Z 오더 안의 오브젝트들은 Z 값이 변경될 때 따라서 정렬되어야 한다는 내용이 있었다. 다시 말해 우리의 정렬 알고리즘이 안정 정렬(stable)이어야 했는데, 퀵소트는 불안정 정렬이었다. (정말 이 문제가 게임 그래픽에 나타난 그 불가사의한 현상의 원인이라는 것을 알아내는데 한참 걸렸다)

쉽게 고칠 수 있는 문제라고 생각했던 우리의 오만함을 조금 꺾고, 안정 정렬이면서 효율적이고 좋은 정렬 알고리즘을 찾아보았다. 다음으로 시도해 본 건 트리 정렬(tree sort)이었다. 트리 정렬은 안정 정렬이긴 하지만 굉장히 느렸다. 이미 거의 순서대로 되어 있는 컬렉션에서 트리 정렬을 사용하고 있는 게 맞는지 알 수가 없었다. 이건 트리 정렬의 아킬레스건이었다. 이런 경우에 트리 정렬은 전혀 효율적이지 않다.

웹 검색을 통해서 우리는 별별 종류의 정렬 알고리즘과 그 변형을 찾을 수 있었다. 그 중 어떤 건 효과가 있을 것이고 다른 어떤 건 소용 없을 테지만 우리는 각각을 공부하고 이해해서 우리 니즈에 맞는 걸 결정할 시간이 없었다.

그래서 우리는 퀵소트로 돌아갔다. 이번에는 Z 오더에 바이어스(bias)를 적용해 각 오브젝트의 Z 오더가 변경되었는지를 나타내는 인덱스를 주었다. 완벽하진 않지만 주어진 시간을 감안하면 합리적인 선택이었다.

결론적으로 이 최적화로는 릴리즈 하지 못했다. 수정한 것을 충분히 테스트할 시간이 없었고, Z 오더로 부하가 걸리지 않도록 게임에서 한번에 여러 오브젝트를 만들지 않는 방식으로 문제를 피해갈 수 있었다. 전체 화면의 젤리를 한번에 만드는 대신, 한 줄씩 차례로 만들도록 컨텐츠 개발자들에게 안내해 주었다.

O(1) 항상 정렬하는 컬렉션

<젤리 점프>는 릴리즈 했지만, 다음 게임을 위해서는 이 문제를 고쳐야 했다.

Red Hat Linux 8 을 커널 2.4.18(이나 비슷한)로 쓰던 시절에 당시 개발 중인 리눅스 2.6 이 완전 새로운 O(1) 스케쥴러[1]를 사용한다는 기사를 읽은 기억이 있다. 기사 내용은 이 스케쥴러가 시스템의 로드와 무관하게 여러 개의 업무(task)를 상수 시간 복잡도(constant time complexity)로 스케쥴링 할 수 있다는 것이었다. 이 스케쥴러는 2.6.23 이 나올 때까지 보류되었다가 O(log on) 복잡도의 스케쥴러로 대체되었다.

[1] 참조링크: http://www.ibm.com/developerworks/linux/library/l-scheduler/

이 O(1) 스케쥴러를 만들게 된 것은 레드햇의 프로그래머 잉고 몰나(Ingo Molnar)가 우선 순위 정보를 업무 자체에 저장하고 우선순위에 따라 컬렉션을 정렬하는 방식을 취하지 않기로 했기 때문이었다. 대신 업무의 우선순위 정보를 컬렉션 자체에 저장했다. 잉고 몰나는 각각의 우선 순위에 따라 다른 큐(queue)를 만들어서 각 큐의 헤드로 직접 접근했다.

우선 순위의 숫자는 미리 정해져 있으므로, 우선 순위 큐를 모두 스캔해서 삽입하거나 큐에서 빼는 일이 즉시 이루어질 수 있다.

업무는 거의 예측이 불가능하지만 그래도 스케쥴링을 위해서 순서대로 정렬되어 있어야 한다는 점에서 우리 게임의 오브젝트들과 비슷하다. 그래서 우리는 이 리눅스 솔루션에 기반해서 새로운 접근을 시도했다.

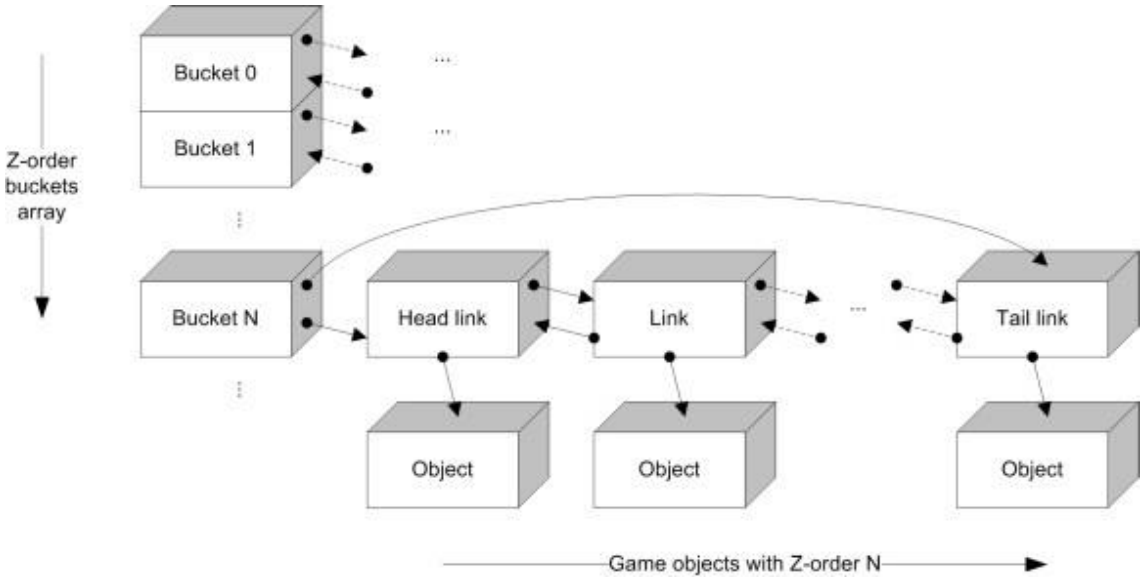
우리 컬렉션의 요구 조건을 살펴보자면 이렇다.

- 컬렉션은 O(n) 이내에 앞 뒤 어느 방향으로든 순서대로 가로지를 수 있어야 한다.
- 같은 Z 오더의 오브젝트는 마지막 Z 액션에 따라 하위 정렬(sub order) 되어야 한다.
- Z 오더의 삽입, 삭제, 변경은 O(1) 이어야 한다.
- 메모리 복잡도가 낮아야 한다.
- Z 오더 값은 모두 legal (음수나 정수가 아닌 값을 포함)이어야 한다. –하위 버전 호환 문제
-

단순화

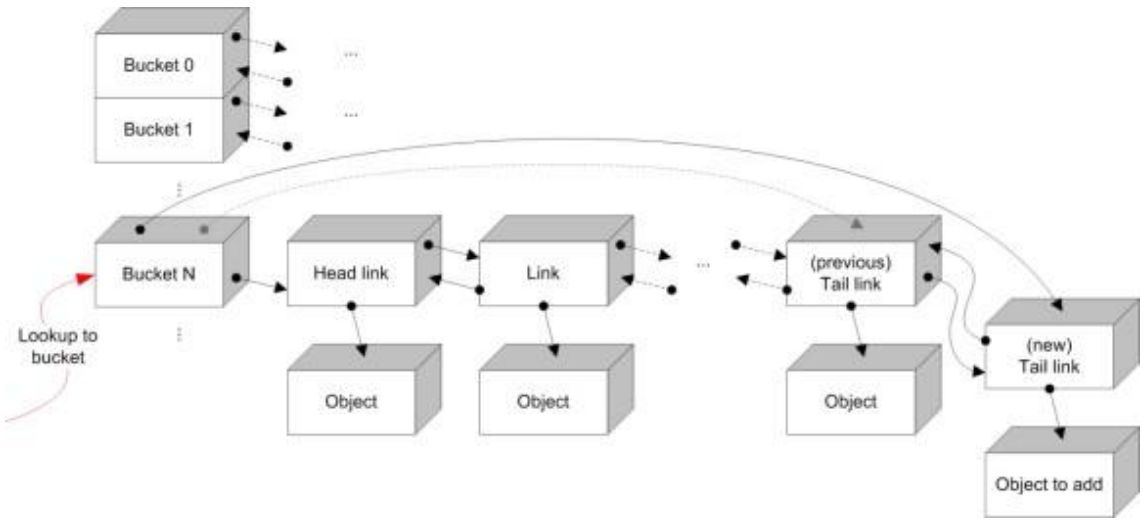
우리는 비교적 단순한 케이스부터 시작하고, 마지막 조건은 무시했다. 우리는 한 오브젝트의 Z 오더를 범위 내의 자연수(말하자면 0 에서 100)으로 정의하고 Z 오더 숫자가 낮은 오브젝트가 뒤쪽으로 가게 했다.

배열 내의 Z 오더마다 각각의 "버킷(bucket)"을 가져가는 방식으로, Z 오더가 있는 오브젝트를 입력 순으로 보관하는 것이었다. 우리는 "버킷"을 양방향 연결목록(double-ended linked-list)로 구성하여 새 오브젝트의 O(1) 입력을 끝에 붙이고, 양쪽에서 가로지를 수 있게 했다.



간단한 데이터 구조

Z 오더 버킷에 새 오브젝트를 추가하려면 버킷 배열에서 Z 오더를 찾고, 목록의 꼬리에 오브젝트를 덧붙이면 되었다.



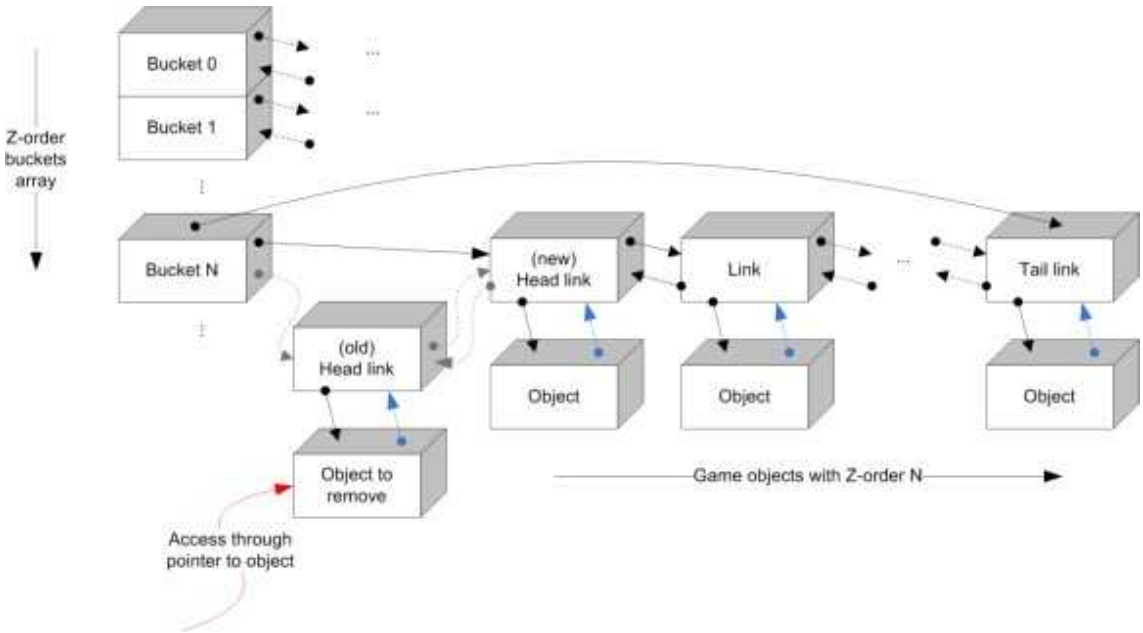
버킷에 오브젝트 추가

오브젝트를 목록의 꼬리에 덧붙임으로써, 마지막 Z 오더 액션에 따른 하위 정렬을 유지할 수 있다. (우리는 뒤에서 앞으로 가로지르는 방식을 취했다)

전체 구조를 가로지르는 건 간단한 일이다. 처음(이나 마지막) 버킷에 들어있는 오브젝트들을 스캔하고 다음(이나 이전) 버킷으로 넘어가서 버킷이 더 이상 없을 때까지 반복하면 된다.

그렇지만 우리는 여전히 버킷에서 오브젝트들을 꺼낼 때 표시를 해야 했다(오브젝트가 파괴되었거나 Z 오더가 바뀌어서 다른 버킷으로 옮겨갔거나). 버킷 목록에서 전체 컬렉션을 스캔 해서 오브젝트 링크의 위치를 찾는 일을 피하려고 오브젝트 자체에 보조 필드를 사용할 것이다. 현재 링크된 목록의 오브젝트를 담고 있는 버킷 링크를 캐시에 저장해서 O(1)에 접근할 수 있다.

우리는 오브젝트로부터 버킷 내 오브젝트의 현재 위치로 백 리퍼런스(back reference)를 만들었다.



버킷에서 한 오브젝트를 제거
(파란색 화살표가 오브젝트에서 링크된 목록으로의 링크다)

여기서부터 복잡해진다. 버킷을 선택하는 것은 인덱스를 이용한 배열검색이기 때문에 링크된 목록의 끝에 추가하는 것은 O(1)이고 링크된 목록에서 링크를 제거하는 것도 O(1)이다(오브젝트 자체에 링크를 부여함). 우리는 필요한 액션 모두에 대해서 항상 정렬된 컬렉션을 O(1) 실행시간으로 보존할 수 있다.

이제 이게 어떻게 구현되는지 살펴보자. 먼저, 게임 오브젝트는 Z 오더를 리포트할 수 있어야 하고, 보조 링크를 가진다.

```
public interface ZSortable {
      public int getZOrder();
      public Unlinkable getCurrentLink();
      public void setCurrentLink(Unlinkable currentLink);
}
```

앱스트랙션(abstraction) 때문에 실제 링크된 리스트의 링크 클래스를 오브젝트에 둘 수는 없지만, 인터페이스를 통해서 링크를 리스트에서 제거할 수 있다.

```
public interface Unlinkable {
      public void unlink();
}
```

다음 단계는 버킷의 내용물을 담아둘 리스트를 구현하는 것이다. 캐싱을 위해서 오브젝트의 링크를 노출시켜야 하기 때문에, Java 의 빌트인 컬렉션은 쓸 수 없다.

```
public interface ZLinkedList extends Iterable<ZSortable> {
      public Unlinkable append(ZSortable object);
      public Iterator<ZSortable> iterator();
      public Iterator<ZSortable> reverseIterator();
}
```

이제 각각의 유효한 Z 오더마다 버킷을 만드는 것만 남았다.

```
public class ZCollection extends Iterable<ZSortable> {

      private final ZLinkedList[] buckets =
            new ZLinkedList[MAX_Z_LEVEL + 1];

      public ZCollection() {
            for (int i = 0; i <= MAX_Z_LEVEL ; ++i) {
                  buckets[i] = new ZLinkedListImpl();
            }
      }

      public void add(ZSortable object) { ... }
      public void remove(ZSortable object) { ... }
      public void change(ZSortable object) { ... }

      public Iterator<ZSortable> iterator() { ... }
      public Iterator<ZSortable> reverseIterator() { ... }
}
```

그리고 이렇게 해서 기본 데이터 구조는 완성이 되었다. 각각의 액션을 검증하고 어떻게 O(1) 오퍼레이션으로 만들었는지 살펴보자.

오브젝트 생성

새 오브젝트를 컬렉션에 추가하려면 그저 현재 Z 오더의 버킷 끝에 덧붙인 다음 오브젝트에 링크를 캐싱 하기만 하면 된다. 그러면 같은 Z 오브젝트의 서브오더도 보장된다.

```
// member of ZCollection
public void add(ZSortable object) {
      // ... check that 'object' isn't null ...
      int zOrder = object.getZOrder();
      // ... assert that: zOrder >= 0 and zOrder <= MAX_Z_LEVEL
      Unlinkable link = buckets[zOrder].append(object);
      object.setCurrentLink(link);
}
```

양방향 링크 목록이기 때문에 목록 끝에 덧붙이는 것은 O(1) 액션이다.

오브젝트 파괴

Z 컬렉션에서 오브젝트를 제거하려면 버킷 리스트에서 링크를 제거하면 된다.

```
// member of ZCollection
public void remove(ZSortable object) {
      // ... check that 'object' isn't null ...
      // ... check that 'object' has a valid link ...
      object.getCurrentLink().unlink();
      object.setCurrentLink(null);
}
```

양뱡항 링크 목록에서 링크를 제거하는 것이 O(1) 오퍼레이션이고 우리는 링크에 직접 접근할 수 있으므로, 오브젝트를 컬렉션에서 제거하는 것도 실행시간 복잡도(runtime complexity)에 해당한다.

Z 오더 변경

오브젝트의 Z 오더를 변경하는 것도 이전 버킷에서 제거하고 새 버킷에 추가하기만 하면 되는 간단한 일이다.

```
// member of ZCollection
public void move(ZSortable object) {
      // ... check that 'object' isn't null ...
      // ... check that 'object' has a valid link ...
      remove(object);
      add(object);
}
```

우리는 오브젝트가 같은 현재와 Z 로 이동할 때도 오퍼레이션을 건너뛰었다. 엔진 계약사항에 따라 이 오브젝트를 나머지 오브젝트보다 상위로 올려야 했기 때문에 일부러 이렇게 한 것이다. 우리는 항상 O(1) 오퍼레이션만 사용하기 때문에 이 역시 O(1) 오퍼레이션이다.

차례로 가로지르기

차례로 가로지르는 것은(앞에서 뒤로든 뒤에서 앞으로든) 간단한 일이다. 우리는 전체 버킷을 마음대로 가로지르고 각각의 버킷에 대해서도 원하는 방향으로 리스트를 가로지를 수 있다.(우리는 보통 뒤에서 앞으로 가로지르지만 버킷이나 버킷의 리스크나 반대로도 할 수 있다.)

링크된 리스트에서 하나 전진하는 것이 O(1) 오퍼레이션이므로 전체 컬렉션을 가로지르는 것은 O(n) 이다.

주의: 나는 구현이 간편한 인터페이스 리스팅만을 소개했다. 전체 코드를 보려면 **https://github.com/mominis/zorder** 와 SimpleZCollection 클래스를 참조하기 바란다.

현실세계

앞에서 소개한 솔루션이 잘 작동하기는 하지만 이대로 우리 엔진에 반영할 수는 없었다. 언제나처럼 하위버전 호환의 문제였다. 앞서 무시했던 마지막 항목에 착수해야 했다.

J2ME 같은 역사적인 이유로 이 엔진은 고정소수점(fixed point) 연산을 하고 "non-integer"와 네거티브 Z 오더를 지원한다. 또한 가능한 Z 오더 전체 정수 범위를 포괄한다. – 한 배열에서 처리하기엔 너무 많다. 게다가 각 오브젝트가 생성될 때 최대 Z 오더 값을 얻는다(2^{31}-1).

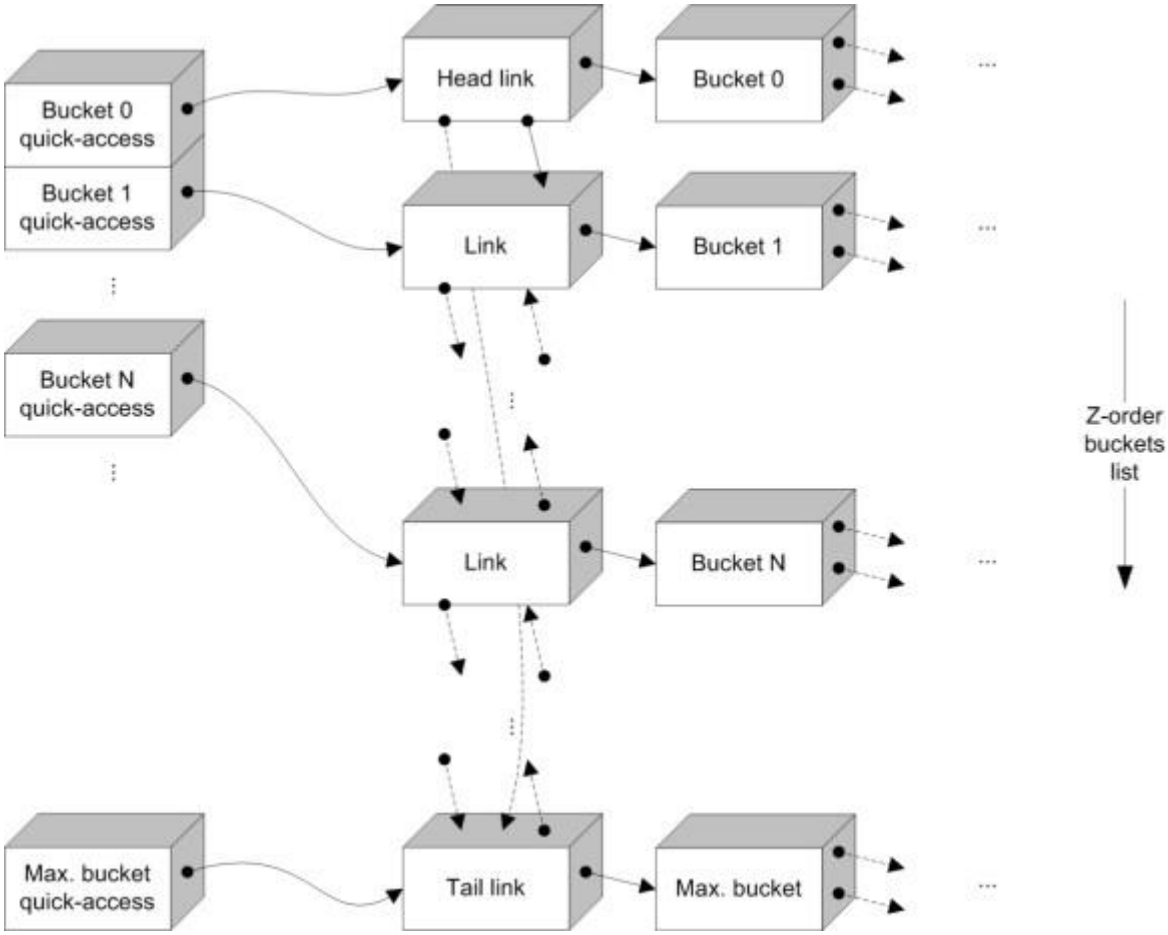
불행히도 우리는 이 문제를 O(1) 실행시간 복잡도나 합리적인 메모리 복잡도로 해결할 방법을 찾지 못했고, 최선의 접근법에 안착해야 했다.

우리는 정수가 아닌 네거티브 Z 레벨을 허용하고, 매우 높거나 낮은 값도 허용했다. 하지만 이럴 경우 범위 내의 자연수 Z 오더만큼 최적의 결과는 나오지 않았다. 우리는 또 디폴트 Z 오더에 대해 특수한 최적화도 제공했다.

우리가 처음 한 일은 버킷 배열을 다른 링크된 리스트로 대체하는 것이었다. 이렇게 하면 정수가 아닌 Z 레벨에 대한 새 버킷을 '최적화된' 버킷 사이에 열 수 있고, 필요에 따라 크거나 작은 버킷을 만들 수 있다.

그리고 최종 Z 오더의 리스트 끝은 최대값 기본 엔트리를 위해 할당했다. 버킷 리스트의 모든 아이템은 정렬 가능한 리스트와 이를 나타내는 Z 오더로 구성되어 있다.

처음에는 버킷 리스트가 각각의 최적화된 자연수 Z 오더와 최대 레벨로 초기화 된다. 그리고 버킷 자체를 가지는 버킷 리스트의 링크가 배열("quick-access" 배열)에 들어간다. 이전과 마찬가지로 링크된 리스트가 사용하는 내부 구조를 캐싱한다.



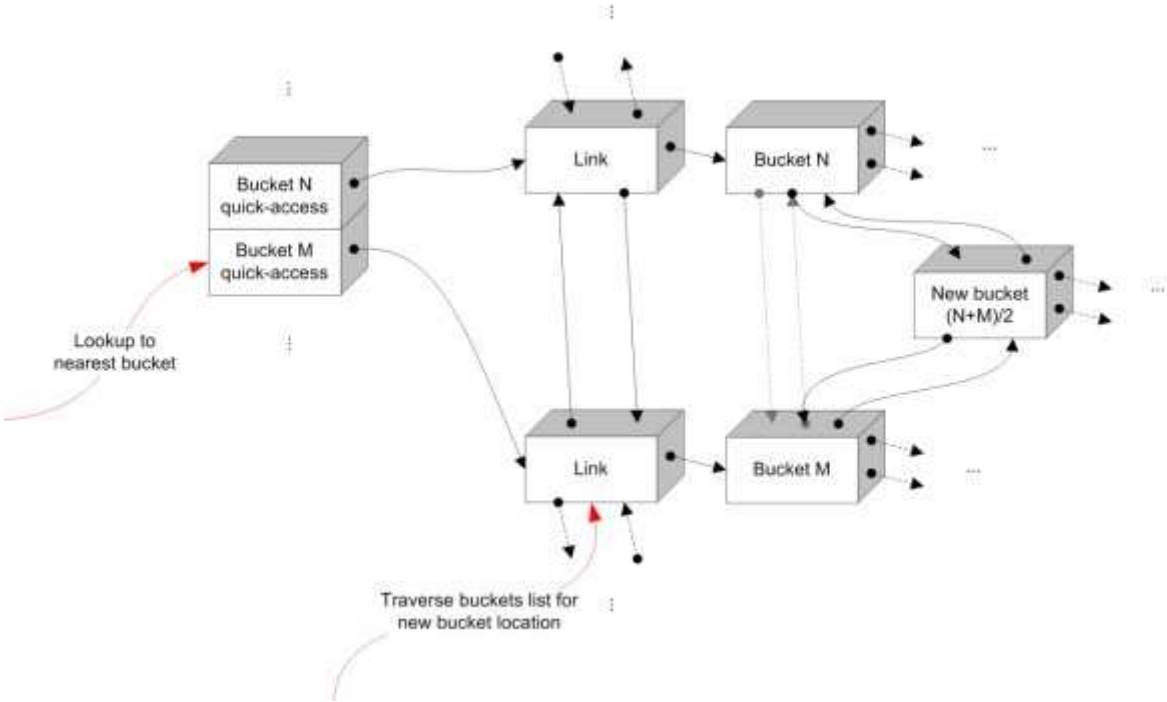
quick-access 배열이 있는 링크된 리스트 내의 버킷들

이렇게 해서 우리는 새 Z 오더 버킷이 사이로 들어오더라도 항상 최적화된 Z 오더 버킷에 빠르게 접근할 수 있다.

새 오브젝트를 추가해야 할 때에는 제일 먼저 그 오브젝트의 Z 오더가 디폴트 값인지를 확인한다. 디폴트 값일 때는 직접 참조하고 있는 버킷에 덧붙인다. 아닌 경우에는 최적화된 Z 오더인지 테스트할 것이다. 최적화된 Z 오더라면 quick access 배열을 검색해 적절한 버킷에 덧붙이기만 하면 된다.

만약 최적화되지 않은 Z 오더라면 다시 O(n) 검색 메소드로 돌아가 새 Z 오더로 기존 버킷들을 검색한다. 우리는 버킷들을 가장 가까운 최적화된 버킷부터

순서대로 가로지르면서 해당하는 Z 오더를 가진 버킷을 찾는다. 찾지 못하면 버킷 리스트의 적절한 위치에 새 버킷을 끼워 넣게 된다.



기존 버킷 사이에 새 버킷을 연다

네거티브 Z 오더도 비슷한 방식으로 지원하는데, 다만 quick access 배열을 사용할 수 없기 때문에 끝에서부터 가로질러 내려오게 된다.

아직 남아있는 작은 숙제 하나는 비어있는 최적화되지 않은 버킷을 버리는 부분이다. 이건 컬렉션을 가로지를 때 해결할 수 있다. 예를 들어 최적화되지 않은 버킷 각각에 카운터를 두어 가로지를 때 버킷이 비어있는 횟수를 기록하는 것이다. 이 카운터가 기준 이상이면 그 버킷을 비울 수 있다.

이렇게 해서 우리는 예전 게임들도 지원할 수 있었으며, 가이드라인을 따르는 새 게임들도 퍼포먼스를 향상시킬 수 있게 되었다.

이 컬렉션의 구현사항 전체를 보고 싶으면 https://github.com/mominis/zorder 에서 FixedPointZCollection 클래스를 참조하기 바란다.

결론

우선, 게임 개발자들 사이에는 퍼포먼스 병목현상이 렌더링 과정이나 게임 로직에서 CPU 측면을 무시할 때 생긴다는 일반적인 개념이 있다. 우리가 <젤리 점프> 문제를 처음 대면했을 때도 우리는 컨텐츠 개발자들이 자체 테스트에서 추측한 것을 받아들여 렌더링 과정에서 문제를 찾으려고 했었다. 렌더링 문제라고 증명하기도 전에 말이다.

몇 시간 동안 찾아도 아무것도 나오지 않다가, 최초의 시험 결과를 무시하고 방법론적으로 접근하기 시작하자 그래픽이 문제가 아님을 곧 알 수 있었다. 버그를 상대하는 개발자로서 버그 리포트를 받았을 때 주목해야 할 것은 '무슨 문제'가 생기느냐이지 '왜' 생겼는지가 아니다. '왜?'를 찾아내는 건 우리 개발자들의 몫이다.

독학으로 프로그래밍을 공부하던 젊은 시절의 나는 컴퓨터공학이라는 걸 믿지 않았었다. 오늘날의 모든 언어는 이미 기본 데이터 구조나 알고리즘이 다 구현되어 있고, 컴퓨터공학 이론 같은 건 '현실에서'는 별 쓸모가 없다고 생각했다. 경험 많은 고참 개발자들 가운데서도 아직 이렇게 생각하는 사람이 있는 걸로 안다. 하지만 지난 몇 년간 문제 해결의 과정에서 20-80 규칙을 더 자주 보게 됐다. 우리의 제일 처음 최적화 시도만 보더라도, Java 가 제공해야 하는 기능으로 퍼포먼스를 향상시킬 수 있었다. 간단한 해결책이 80%를 해결한다- 대부분의 경우엔 이 정도면 충분하다.

하지만 우리는 게임 엔진 개발자이기 때문에 "이 정도면 충분"으로 끝낼 수가 없다. 우리는 최고의 프로그래머여야 한다. 우리는 다양한 디바이스에서 최대한 매끄럽게 돌아가는 어플리케이션을 개발해야 하고, 이용 가능한 CPU, GPU, 메모리를 마지막 한 비트까지 짜내야 한다. (모바일 디바이스의 배터리 문제는 예외지만, 그건 별개 문제니까.) 대부분의 어플리케이션에서 80%는 '충분하다'. 하지만 게임 엔진 개발자는 그 이상을 할 수 있어야 한다. 그래서 특별한 마무리가 반드시 필요하다.

이 경우에 나머지 20%는 여러 가지 접근법을 함께 적용하고 OS-커널 스케쥴러에서 방법을 차용해서 더 나은 해결책을 찾아낸 것이다. 결국 우리는 정렬 알고리즘 하나 없이 컬렉션을 정렬했다! '전통적인' 방법을 차용해서는 여기에 다다를 수 없었을 것이다. 어떤 지식이든, 자기 분야와 무관해 보일지라도 절대 무시하지 말기 바란다. 그것이 빛나는 순간을 만드는 열쇠가 된다.