화재, 피, 폭발 - <프로토타입 2>의 과장된 효과 기술

(Fire, Blood, Explosions: Prototype 2's Over-the-Top Effects Tech)

작성자: 케이스 오코너(Keith O'Conor)

작성일: 2012년 10월 23일

<게임 개발자 잡지> 2012년 4월호에 실린 이 기사에서, 래디컬 엔터테인먼트(Radical Entertainment)의 시니어 렌더링 코더(senior rendering coder)인 케이스 오코너(Keith O'Conor)는 게임의 파티클 시스템(particle system) 요소에 대해 설명한다. 특히 오픈 월드 어드벤처 게임에서 잘 실행되는, 멋진 효과를 만드는 방법에 대해 상세히 살펴볼 것이다.

<프로토타입(Prototype)> 세계의 특징 중 하나는 지나치게 혼란한 오픈 월드다. 우리는 혼란을 만들기 위해 대단히 많은 양의 파티클 효과를 이용했다. 화재, 피, 폭발, 그리고 무기 충격 효과로 환경을 채운 것이다.

제임스 헬러 병장(Sgt.James Heller/ 메인 캐릭터)은 환경 내부 어디라도 갈 수 있다. 빌딩 측면을 달려 올라갈 수도 있고, 옥상을 가로질러 활공하거나, 심지어 납치한 헬리콥터로 도시를 가로지를 수도 있다. 이 때문에 우리는 어느 때든지 알아볼 수 있는 수백 개의 복잡한 효과들과 수천 개의 파티클을 지원하기 위한 이펙트 시스템이 필요했다.

이 글에서는 <프로토타입>이나 <프로토타입 2>에 필요한 수준의 이펙트를 밀어넣을 수 있게 하는 기능들을 더하고 향상시킴으로써, 래디컬 엔터테인먼트에서

<스카페이스(Scarface)>와 <헐크: 최후의 파괴(Hulk: Ultimate Destruction)>를 위해 개발한 시스템을 우리가 어떻게 구축했는지 설명하려고 한다.

파티클 모의 실험과 제작하기

우리의 파티클 시스템들은 전적으로 컴포넌트 기반의 특징 세트로 구성되어 있다. 각 특징은 파티클을 처리하는 방식의 한 측면을 의미한다. 예를 들어, 중력이나 다른 어떤 힘에 의해 위치가 바뀌고, 중심점 주위를 돌고, UV에 생기를 불어넣고(animating UV), 계속해서 크기를 바꾸는 것과 같이 말이다.

이펙트 아티스트(effect artist)는 개별 파티클 시스템을 만드는 특징을 뭐든 선택할 수 있으며, 각각의 선택된 특징은 그가 수정하거나 생기를 불어넣을 수 있는 (속도, 중량, 색상 같은) 관련 속성을 드러낸다. 이는 마야의 표준 애니메이션 툴셋으로 이루어지는데, 같은 시뮬레이션 코드를 이용해 마야 플러그인(Maya plug-in)으로 엮음으로써 마야와 게임이 모두 끊임없이 작동하게 했기 때문이다.

아티스트가 마야에서 파티클 시스템의 스타일과 작동에 만족하면, 게임에 실을 수 있는 효과로 내보내진다. 그러면 이 효과는 우리의 인게임 에디터 "더 짐(The Gym)"을 이용해 게임플레이에 쓰이는데, 이 복잡한 상태 기계 에디터는 디자이너들이 게임의 모든 측면을 통제할 수 있게 해준다.(자세한 사항은 우리의 <GDC2006> 프레젠테이션[1]을 보라.)

특정 상황에서 플레이하기 위한 효과 스크립트를 쓸 때, 이펙트 아티스트는 추가 컨트롤 설정, 가령 오버라이드(override)와 편향(biase)에 접근한다. 각 기능에 대한 추가 속성으로, 이펙트 아티스트는 활발한 가치에 편향(증폭)되거나 그것을 완전히 무시할 수 있다. 이는 하나의 연관된 효과가 다양한 상황에서 쓰일 수 있도록 했다. 예를 들어, 아티스트가 표준 연기 효과를 이용해 작고 가볍고 빠르게 움직이게 하거나, 크고 짙고 검은 연기를 선택할 수 있다. 그저 배출량, 색상, 속도 같은 속성을 무시하거나 증폭시킴으로써 말이다. (그림 1의 예제 참조)

---

[1] 참조 링크: http://www.gdcvault.com/play/1013444/The-Gym-Where-The-Incredible

그림 1. 원래 효과(왼쪽)와 편향, 오버라이드를 다르게 준 세 가지 변형

아티스트들은 다수의 상황에서 동일한 포괄적인 효과를 사용한다거나 같은 효과의 비슷한 버전을 많이 싣고 제작하는 대신에 '더 짐'을 이용해 게임 내에서 사용되는 상황에 따라 각 효과를 조정할 수 있다. 이는 게임 내 조명과 애니메이션으로 효과를 살릴 수 있게 함으로써 메모리 사용량을 줄이고 아티스트의 작업 흐름을 향상시킨다. 편향과 오버라이드는 또한 이 글의 후반부에 설명할, 지속적인 LOD 시스템(level-of-detail system)의 주요 부분이다.

각 파티클 시스템의 속성은 모든 파티클의 위치, 수명, 속도 등등을 각각의 배열에 밀집해 저장한다. 이렇게 데이터 지향적인 디자인을 통해 모든 프레임의 시뮬레이션 상태를 업데이트할 때 캐시 효율성이 높은 방식으로 데이터에 접근할 수 있게 해준다. 이 부분은 파티클 시뮬레이션을 할 때 CPU 실행에 큰 충격을 줄 수도 있다.

이 방식으로, 우리는 복잡한 작용으로 수천 개의 파티클들을 시뮬레이션 할 때 CPU 시간을 조금밖에 사용하지 않는다. 또한 비교적 간단하게 PS3의 비동기 SPU를 실행할 수도 있다. 각 기능을 업데이트할 때, 관련 없는 데이터는 하나도 없이 그 기능에 필요한 속성만 기억장치에 직접 접근해 가져온다는 뜻이다.

파티클 위치 분리는 다른 실행과 같은 이점을 가진다. 빨라지고, 정확한 알파 혼합 렌더링을 위한 캐시 효율성 높은 카메라 관련 분류처럼. 이는 또한 다른 기능들도 가능하게 하는데, 말하자면 다른 시스템의 파티클 처리 프로세스에 넣음으로써, 시뮬레이션 업데이트의 위치 출력 기능 구분을 이용해 다른 파티클들을 내보낼 수 있다.

메모리 사용과 조각화 줄이기

가용 메모리를 쪼개면, 많은 단기 파티클 효과를 언제나(예를 들면 심각한 전투 상황에서) 진행할 수 있게 된다. 그래서 인접한 미사용 메모리의 양을 제한하는 "스위스 치즈(Swiss cheese)" 효과로 이어지며 메모리의 많은 작은 조각들이 본질적으로 무작위로 할당되고 마련될 때 분열이 일어난다.

다시 말해, 힙(heap) 내 미사용 메모리의 총량은 효과를 내는 데는 충분하겠지만, 메모리는 실제 사용하기엔 너무 작은 양의 힙으로 흩어질 것이다.(분열과 할당을 위한 가이드는 스티븐 토비(Steven Tovey)의 훌륭한 글 #AltDevBlogADay [2] 를 읽어보라.) 비록 우리가 분할의 위치를 알기 위해 파티클 할당량을 분할해 사용하고 있지만, 이는 여전히 문제가 된다. 다행히도 우리는 분할을 제한할 수 있는, 그리고 문제가 될 때 다룰 수 있는 작은 트릭이 있다.

우리는 분열과 동적 할당 비용을 피하기 위해 가능한 한 언제나 (처음부터 할당된) 고정된 세그먼트화 메모리 풀을 사용한다. 그 부분은 파티클 시스템 할당에 가장 흔히 사용되는 구조에 크기가 맞춰져 있다. 이 풀이 가득 찰 때만 동적 할당 비용을 실행하면 된다. 이러한 일은 특별히 심각한 전투의 순간이나 많은 파티클 효과가 한번에 실행되는 순간에만 일어난다.

우리의 효과 시스템은 하나의 파티클 시스템이 만들어질 때 메모리 할당을 여러 개 해둔다. 만약 이중 어떤 것이 (분열, 혹은 힙이 다 차서) 실패하면, 효과는 만들어지지 않는다. 효과를 반만 만들거나 이미 만들어진 어떤 할당을 비우거나 (아마 힙을 더 분열시키거나) 하는 대신에 우리는 효과 힙에서 크게 하나를 할당한다.

이게 성공하면 작업을 진행하고 이 메모리를 모든 할당에 사용한다. 만약 실패하면, 효과를 초기화하려는 시도조차 하지 않고, 그냥 플레이를 안 한다. 플레이어의

---

[2] 참조 링크: http://www.altdevblogaday.com/2011/02/12/alternatives-to-malloc-and-new/

관점에선 분명히 바람직하지 않다. 폭발 효과가 실행되지 않으면 폭발하는 차가 매우 이상하게 보일 것이기 때문에, 이는 마지막 방책이다. 대신에, 우리는 힙이 절대 꽉 차지 않게 하거나 처음부터 과하게 분열되지 않도록 한다.

이것이 끝날 때쯤에, 우리가 하는 것 중 하나는 대략 효과의 분류에 따라 "저장소(stores)"에 효과를 분할하는 것이다. 우리는 폭발, 잔잔한 효과, 총알 폭죽, 그밖의 다양한 다른 효과 유형의 저장소를 가지고 있다. 이렇게 효과를 구분함으로써, 언제든지 존재하는 특정 유형의 효과 수를 제한할 수 있다.

우리의 효과 힙은 이와 같이 다른 유형의 효과에 메모리를 허용하지 않기 때문에, 수백 개의 피 튀기는 효과 같은 것들로 가득 차지 않는다. 저장소는 배열로 구조화된다. 저장소가 가득 차고 새로운 효과가 실행될 때, 저장소에서 가장 오래된 효과는 "폐기장(graveyard)" 저장소(오래된 효과가 사라지는 곳)로 옮겨진다. 그 효과들의 배출률은 0으로 정해져 있어서 새로운 파티클들이 배출될 수 없고, 특정 시간(보통 단 몇 초)이 지나면 사라지고 죽게 되며, 그 결과 삭제된다.

효과를 저장소에 분할하는 것은 또한 우리가 효과 유형에 따라 다른 최적화를 실행할 수 있도록 해준다. 예를 들어, 우리는 "폭죽(squib)"에 있는 효과는 모두 불꽃이나 담배연기처럼 작은 단명 효과라는 것을 추측할 수 있다. 따라서, 카메라 바깥 혹은 특정 거리보다 훨씬 멀리에서 이런 효과가 실행될 때, 우리는 그냥 효과를 전부 실행하지 않으며 누구도 이를 알아채지 못한다.

또 다른 예는 "폭발" 저장소의 효과에서 파티클이 카메라에서 너무 가까워 액션의 시야를 가릴 것 같을 때, 그리고 렌더링하기 너무 비쌀 때 사라지는 것이다. 플레이어가 그를 해치려는 적군, 탱크, 헬리콥터 무리에 둘러 쌓였을 때, 이 최적화가 중요해진다.

우리는 또한 인스턴싱 효과(instancing effect)로 메모리 사용을 줄일 수 있다. 우리의 오픈 월드 설정에서는 같은 효과가 다수의 장소에서 실행될 때가 많다. 예를 들면 맨홀 뚜껑에서 나오는 증기와 불타는 빌딩에서 나오는 연기처럼. 이런 경우, 우리는 하나의 개별적인 "모체(parent)" 효과만을 할당하고 시뮬레이션한다. 그러곤 효과가 실행될 때마다 이 모체 효과의 "복제(clone)"를 배치한다. 모체 효과에만 파티클 시뮬레이션을 하면 되기 때문에, 또한 복제할 때에는 작은 양의 부기 데이터(bookkeeping data)만 있으면 되기 때문에, 우리는 메모리와 CPU 사용에 극소량의 충격만 주고 복제를 많이 덧붙일 수 있다. 시각적 반복을 피하기

위해, 각 복제를 교대로 내보이거나 색감을 입혀 약간씩 다르게 보이게 할 수 있다.

버텍스 버퍼 메모리 수요(Vertex Buffer Memory Demands) 관리하기

각 파티클 시스템은 복제됐든 아니든 시뮬레이션에 필요한 메모리에 덧붙여 버텍스 버퍼(vertex buffer)를 저장하기 위한 메모리가 필요하다. 새로운 파티클이 생성되고 오래된 것이 죽음으로써 시스템의 파티클 수가 매 프레임마다 바뀔 수 있기 때문에, 메모리의 양은 버텍스 버퍼 차이와 비슷하게 필요하다.

단순히 시스템이 만들어질 때 가능한 버티클의 최대량을 저장할 수 있는 충분한 공간을 할당할 순 있지만, 만약 효과의 주요 지속 기간이 매우 적은 파티클들만 발생한다면 이는 매우 큰 낭비가 될 것이다. 대신에, 우리가 효과 힙에서 해당 프레임 할당만 실행시키는 것을 대체할 수 있지만 모든 프레임에서 이 버퍼들을 만들고 없애는 것은, 다양한 동적 할당을 하기 위해 뒤틀기를 더하고 메모리 분할의 가능성을 증가시키고 더 많은 처리 오버헤드(processing overhead)가 필요해진다.

우리는 다양한 버텍스 버퍼 힙을 사용하는 대신에, 이중 하나의 프레임에만 필요한 모든 버텍스 버퍼를 할당했다. 왜냐하면 파티클 버텍스들은 돌아다니는 모든 프레임에 만들어지며 지속될 필요가 없기 때문에(게다가 GPU에선 이중으로 저장되므로), 우리는 간단하게 한 줄로 된 할당기(allocator)를 사용할 수 있다.

이는 모든 프레임을 내보내는 할당기이며, 모든 할당은 단순히 빈 메모리의 시작에서 배치된다. 이는 몇 가지 이점을 가지고 있다. 분할이 완전히 없어지고 할당을 실행하는 것이 단순한 원자 포인터 연산(atomic pointer arithmetic)이 된다. 또한 메모리는 절대 비울 필요가 없다. "빈 메모리" 포인터는 각 프레임에서 힙이 시작될 때 그냥 리셋된다.

덧붙여, 이 힙은 파티클 시스템의 버퍼들로 제한될 필요가 없다. 이는 피부, 모션 흔적(motion trails), 조명 반사 카드 따위를 포함한 모든 프레임에 버텍스 버퍼를 만드는 어떤 코드에 의해서든 사용될 수 있다.

이 매우 중점적인 힙으로 인해 시각 테스트에 떨어진 어떤 다양한 물체도 프레임의 메모리를 필요로 하지 않음으로써, 우리는 오직 실제로 렌더링되는 물체의 메모리

대금만 지불하면 되었다. 만약 우리가 각 물체가 생성돼서 사라질 때까지 (실제로 그 물체를 거의 볼 수 없음에도 불구하고) 메모리를 할당해야 한다면, 우리는 이와 같은 버텍스 버퍼 할당을 통합하는 것보다 훨씬 더 많은 메모리를 사용할 것이다.

미세 동조 렌더링 실행(Fine-Tuning Rendering Performance)

<프로토타입 2>의 게임 세계에서 효과는 통제에서 벗어나기 쉽다. 폭발, 연기, 피분사, 불, 그리고 폭죽은 모두 주기적으로, 종종 모두 한 번에 사라진다. 이렇게 됐을 때, 프레임 버퍼에 섞인 많은 양의 픽셀들이 프레임 속도를 매우 느리게 만든다. 따라서 우리는 실행 문제를 확인하고 다루기 위한 엄청난 양의 효과 기술에 전념해야 한다.

우리는 이 짐을 이펙트 아티스트에게 맡기기로 했다. 그들이 효과를 만든 사람들이라서 모든 기술과 게임 플레이 요건을 알기 때문이다. 또한 의도적으로 효과 관련 프레임 속도 문제를 그들이 책임지게 했다. 그렇지 않으면, 우리는 그들이 릴리즈 전에 렌더링 팀이 최적화하길 바라면서 보기는 좋으나 엉망으로 실행되는 것을 만들 때가 많다는 걸 깨달았다.

이렇게 되면 프로젝트 막판의 귀중한 시간을 너무 많이 잡아먹었다. 그리고 심지어 실현 가능하지도 않을 때도 많았다. 이는 별로 놀랄 일이 아니다. 아티스트가 실행 제약을 이해하고, 그 안에서 일하는 업계에선 이런 것은 보편적인 관례다. 하지만 기한이 다가오고 모든 사람이 압력을 받을 때, 그냥 해놓고 나중에 고치지 뭐, 하는 생각을 많이들 한다.

우리는 실행을 수량화하는 것이 아티스트에게 더 쉬우면 그들이 무언가 잘못했을 때 처음부터 제대로 하는 게 가능성이 높다는 것을 깨달았다. 이건 우리가 <프로토타입>을 마무리할 때 GPU 시간이 효과에 쓰이는 바람에 모든 것이 거의 최저 프레임 속도를 향하는 상황에서 어렵게 배운 교훈이다. 아티스트들이 쉽게 실행 정보에 접근할 수 있게 해주면 그들은 실행 조율에서 더 적극적인 역할을 한다.

이 피드백은 전반적으로 프레임당 예산에 비교해 파티클 렌더링 비용이 얼마인가를 보여주는 간단한 비율로 시작했다. 그리고 아티스트에게 각 개별 효과에 대한 세부사항을 주는 것을 확장했다.(그림 2를 보라.) 그들은 최근 실행된 모든 효과

리스트와 각각의 메모리 사용과 GPU 로딩의 측면에서 비용이 얼마인지를 볼 수 있었다.

Effects

ID	Drawable	Store	Particles	Memory
2422	fx_hammerfist_spike_impact	Explosion	3 / 3	1.8k
2423	explosionFireBall_Fire	Explosion	2 / 2	1.7k
2426	explosionHeliDamage	Persistent	6 / 16	3.4k
2430	fxDebrisMetal001	General	6 / 14	4.7k
2428	gorePustuleBurst	Character	1 / 12	3.0k
2429	gorePustuleBurst	Character	9 / 12	3.0k
2442	gorePustuleBurst	Character	1 / 2	1.8k
2443	gorePustuleBurst	Character	8 / 12	3.0k
2445	gorePustuleBurst	Character	8 / 12	3.0k
2427	fx_embers002	Ambient	6 / 24	7.2k
2251	fx_embers002	Ambient	8 / 24	7.2k
2258	steam001	Ambient	1 / 7	2.3k
2437	biasSmoke001	Persistent	3 / 18	4.3k
2250	explosionHeliDamage	Persistent	1 / 16	3.4k
2424	explosionDebris001	Explosion	33 / 33	8.8k
2432	fxDebrisMetal001	Explosion	3 / 14	4.7k
193	biasSmoke005	RedZoneSmoke	clone	0.5k

GPU 로드에서 우리는 전반적인 예산 비율처럼 각 효과를 화면에 쓰는 데 얼만큼의 픽셀이 드는지 확인하는 간단한 폐쇄형 쿼리 카운터(occlusion query counter)를 사용했다. 이는 너무 많은 파티클이나 (시간은 투자하지만 최종 이미지에 전혀 기여하지 않는 엄청난 양의 완전히 투명한 픽셀을 초래하는) 형편없는 텍스쳐 사용으로 인한 GPU 초과 사용의 훌륭한 지표가 될 수 있다.

아티스트들은 어떤 효과가 가장 비싼지 그리고 어디에 집중해 최적화를 해야 하는지 바로 볼 수 잇다. 다른 시각화 모드들 역시, 오버드로우(overdraw)를 야기하는 묘사 렌더링이나 특정 효과 파티클의 와이어 프레임(wireframe)을 내보이는 등의 실행 문제를 조사하는 데 유용하다.

보편적인 렌더링 기술이 그러하듯이, 우리가 아티스트들의 작업에 대해 더 빠른 피드백을 줄수록, 그들이 일을 더 잘할 수 있고 게임이 전반적으로 더 좋아지는 것이다. 모두에게 좋은 셈이다!

실행 증가를 위해 이펙트 스크립팅과 LOD 사용하기

우리의 효과 스크립팅 시스템은 또한 최근 렌더링 양에 기반한 세부 효과 등급을 바꾸기 위해 만들어졌고, 효과들을 적합하게 활용하기 위한 주요 수단이 되었다. 도시의 이쪽 끝부터 다른 쪽 끝까지 어떤 효과를 볼 수 있게 함으로써, 카메라 부근의 효과에 들어가는 예산에 더 집중하고 최근 렌더링 로딩에 근거해 LOD를 바꿀 수 있어야 했다.

LOD 시스템은 주로 위에 설명한 편향과 오버라이드의 속성에 기반한다. 이펙트 아티스트는 특정 거리에서 효과의 편향과 무시 값을 바꾸는 LOD를 만들어낼 수 있다. 그러면 이 값은 카메라와 효과의 거리에 기반해 모든 LOD 사이를 채운다. 예를 들어, 거리가 멀 때 아티스트는 배출률을 낮추고 효과의 파티클 크기를 증가시키는 선택을 할 것이다. 그리고 이는 가까이서 봐야 알아차릴 수 있을 정도의 디테일은 떨어지지만, 비슷한 모습을 유지하더라도 오버드로우 양은 줄 것이다.

보간(interpolation)은 튄다거나 다른 비슷한 문제 없이(비록 여전히 특정 거리에서 (크로스페이드(cross-fade)와 함께) 완전히 다른 효과와 전환되거나 효과가 모두 함께 불구가 될 여지가 있지만) 지속적인 LOD 이행하게 한다. GPU 비용 절감이

주요 목표지만, 이 LOD들은 주로 결국 메모리와 CPU 시간 모두를 절약한다.

LOD를 고를 때 우리가 사용하는 다른 측정 기준은 이전 프레임 파티클들의 렌더링 가격이다. 이는 아티스트에게 주어진 통계가 발생하고 LOD 시스템에 의견을 주는 같은 폐쇄형 쿼리를 사용한다.

만약 이전 프레임이 비교적 비쌌다면 우리는 더 비싼 효과를 만듦으로써 최근 프레임을 더 나쁘게 만들고 싶지는 않다. 따라서 우리는 더 빨리 회복하기 위해 더 싼 LOD들을 실행한다. 아티스트는 어떤 LOD를 선택하고 어떤 실행 등급이 사용되어야 하는지에 대한 전적인 통제권을 갖는다.

파티클 때문에 프레임 속도가 상당히 떨어졌을 때, 이는 종종 하나의 비싼 효과 때문이 아니라 적당히 비싼 효과가 많이 한 번에 사용돼서 발생한다. 이 점에서 최적화는 무엇이든 어떤 다른 효과가 실행되었는지 고려해야 한다. 이를 위해 우리는 "효과 타이머"를 쓴다. 효과 타이머를 이용하여, 특정 효과가 최근에 이미 실행되었는지를 확인하고, 이에 기반해 다른 효과를 실행하도록 선택할 수 있다.

제일 좋은 예는 크고 비싼 폭발이다. 우리는 한 번에 하나의 큰 폭발이 발사되고 다른 동시 폭발이 더 작고 덜 비싸길 바란다. 미사일이 차량 사이로 발사되어 서너 개의 차들이 한 번에 폭발할 때가 그렇다. 한 대는 보기 좋은 효과를 실행하고 다른 차들이 더 작고 싼 비용으로 폭발하는 것이다. 훨씬 적은 렌더링 비용으로 비슷한 시각적 효과를 내는 셈이다.

우리의 효과 스크립팅 시스템이 주로 렌더링 최적화의 수단이지만, 이는 게임 플레이에도 유용하게 적용할 수 있다. 예를 들어, 플레이어가 멀리서 탱크를 포격해 태웠을 때 만약 플레이어가 AI의 탱크 포격에 맞아 같은 폭발이 카메라 바로 앞에서 일어난다면, 결과는 잠시 동안 플레이어를 가릴 것이고 그들의 활동 시야는 완전히 막힐 것이다. 우리는 적절하게 인상적이고 강렬한 폭발을 원하지만 말이다.

이는 전투 중간에 매우 불만스러울 수 있다. 따라서 이런 경우 우리는 파티클 수를 줄이고 불투명도를 낮추며 효과를 더 짧고 작게 만들기 위해 다른 LOD들을 사용할 수 있다. 이는 <프로토타입 2>의 플레이를 더 좋게 만들 뿐 아니라, 프레임 속도에 충격을 덜 주면서 더 싼 효과를 이용하는 것이다.

파티클 렌더링

LOD와 스크립팅 시스템을 최고로 한다 해도, <프로토타입 2>의 아수라장은 여전히 파티클 효과가 매우 비싸질 수 있다는 것을 의미한다. 그렇게 됐을 때, 우리는 (번지(Bungie)의 크리스 추(Chris Tchou)가 <GDC2011>에서 발표[3]했듯) 다중해상도 렌더링을 담기 위해 더 극단적인 전환 방식을 취한다.

더 낮은 해상도 렌더링 타깃(우리의 경우 하프 리졸루션 – 픽셀 수 25퍼센트)으로 전환하기 위한 결정도 이전 프레임의 파티클 렌더링 비용에 근거한다. 타깃이 낮을 때, 모든 파티클들이 전체 해상도 버퍼로 렌더링된다. 이는 간단한 장면이 전체 해상도의 렌더링 파티클보다 더 비싸질 수 있는, 비교적 비싼 낮은 해상도 버퍼의 업샘플(upsample)해야 하는 것을 피하기 위해서다.

일단 퍼포먼스가 특정 수준까지 느려지고 업샘플이 더 나은 상황이 되면, 우리는 나머지 효과를 전체 해상도로 놔두는 대신 특정 효과를 더 낮은 해상도 버퍼 렌더링으로 전환한다. 이 경우, 아티스트들이 어떤 효과가 전체 해상도로 남아 있어야 하는지 골라야 한다. 주로 해상도를 떨어뜨리면 곤란한 고주파수 텍스쳐(high-frequency texture)를 가진 작은 것들, 불꽃, 피, 불 같은 것들이 선택된다. 다른 효과들은 모두 더 낮은 해상도에서 렌더링하게 된다.

그럼에도 불구하고 GPU 시간을 너무 많이 잡아먹게 되면, 마지막 대안으로 아티스트의 선호와 상관없이 모든 효과 렌더링을 낮은 해상도 버퍼로 전환한다. 업샘플을 위해, 우리는 바이래터럴 필터(bilateral filter)보다 더 싸고 더 질 좋은 (<배트맨: 아캄 어사일럼(Batman: Arkham Asylum)>에 쓰인 것[4]처럼) 가장 가까운 깊이의 필터(nearest-depth filter)를 선택한다.

우리는 많은 파티클이 가능한 한 비싸지 않게 이용하는 실제 쉐이더를 유지하고자 하므로, 이는 비교적 간단하다. 우리는 이를 애드-알파 쉐이더(add-alpha shader)라고 부르는데, 이것이 같은 쉐이더를 이용해 파티클들이 (불꽃이나 불 같은 효과의) 추가(additively)나 (연기의) 알파-혼합(alpha-blended)을 렌더링할 수 있게 해주기 때문이다. 쉐이더가 추가되거나 알파-혼합되는 것은 파티클 버텍스 색상의 알파 채널에 의해 결정된다. 이를 위해 우리는 구조의 색상과 알파 채널을 미리

---

[3] 참조 링크: http://www.gdcvault.com/play/1014348/HALO-REACH-Effects

[4] 참조 링크: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/OpacityMappingSDKWhitePaper.pdf

증가시키고 특정 혼합 기능을 이용한다. 적절한 쉐이더 코드를 보려면, 이 글 마지막의 '첨부 1'을 보라.

이는 새로운 기술은 아니지만, 우리 파티클 렌더링에 가장 중요한 부분이다. 이 쉐이더(그리고 공유된 텍스쳐 지도(shared texture atlas))를 이용하는 모든 효과의 파티클들은 하나로 합쳐지고 분류되고 같은 드로우 콜(draw call)에서 끌려들 수 있다. 이는 발생 가능한 튐을 없애는 한편, 두 개의 겹쳐진 효과가 분리된 드로우 콜로 끌려가면 카메라가 움직이고, 드로운이라는 명령이 변할 때 시각적 튐이 있을 수 있다. 버텍스 알파는 또한 시간이 흐르면서 생기를 얻어, 파티클이 첨가된 생을 시작할 수 있다. 하지만 이는 최고의 굉음으로 시작해 사라져가는 짙은 안개로 끝나는 폭발에 매우 효율적인 알파 혼합으로 끝난다.

이 코드 목록에 두 개의 구조 패치가 있다는 것을 눈치 챘을 것이다. 이는 우리 구조 애니메이션의 간단한 서브프레임 보간을 위한 것이다. 이를 통해 우리는 더 적은 프레임을 사용하여 자연스럽게 생기를 주는 이미지를 만들 수 있다

픽셀 쉐이더(Pixel Shaders) 없이 파티클 밝히기

<프로토타입 2>에서 세계는 초록, 노랑, 그리고 빨강, 세 구역으로 나뉜다. 각 구역은 하루의 다른 시간대만큼 뚜렷한 스타일과 색감을 가지고 있다. 조명과 그림자가 없으면 파티클은 이상해 보이기 쉽다. 너무 평평하고, 너무 밝거나 어둡고, 가끔은 전혀 잘못된 색(그림 3의 예시를 보라)이 나오기도 한다. 우리는 조명은 필요하지만 퍼 픽셀 쉐이딩(per-pixel shading)과 이미지 기반 조명(image-based lighting)을 위해 비싼 픽셀 쉐이더 코드를 더하기는 싫다는 걸 깨달았다. 그렇게 하면 우리가 만들 수 있는 파티클들을 상당히 많이 줄여야 하기 때문이다.

그림 3A(위)는 그림 3B(아래)의 색감을 향상시킨 것이다.

해결책은 프리 패스(pre-pass)로서 중급 "파티클 조명" 버퍼에 퍼 버텍스(per-vertex)를 밝히는 것이었다. 각 파티클 버텍스에, 조명 버퍼의 픽셀에 밝기를 분담했다. 이렇게 하면 게임의 나머지와 같은 조명 코드를 쓰고 어떤 플랫폼에서

버텍스 구조 검색의 퍼포먼스 저하를 피함으로써 쉐도우 버퍼와 이미지 기반 조명 구조에서 검색하는 데 픽셀 쉐이더를 쓸 수 있었다.

그러면 이 조명 버퍼는 픽셀 쉐이더에서 더 이상 지시를 하지 않아도 파티클의 버텍스 쉐이더를 판독하고 버텍스 색상과 결합시켰다. 여기서 유일한 문제는 어떤 플랫폼, 특히 PS 3와 몇몇 초기 DX9 GPU에서 버텍스 쉐이더 구조 검색의 퍼포먼스였다.

이런 경우에 우리는 실제로 파티클 조명 버퍼를 버텍스 버퍼로 다시 고쳤고, 어떤 버텍스 흐름이든지 읽어냈다. 이는 어떻게 메모리가 보이고 접근되는지 우리가 모두 통제할 수 있는 PS3와 이를 지원하는 DX9 GPU에선 사소한 문제였다. 우리는 ATI R2VB 확장자를 사용했다(자세한 사항은 링크 참조[pdf[5]]).

종합

파티클은 <프로토타입 2>의 세계에 활기를 불어넣는 데 중요한 부분이다. 다양한 실행 관리 시스템이 가능한 자원 안에서 효과를 내기 위해 함께 작동한다. 조명과 그림자는 시각적 질을 매우 높여주며, 이를 퍼 버텍스로 함으로써, 다른 방법으로 했을 때보다 상당히 적은 비용으로 세계의 모든 파티클을 밝힐 수 있다. 또한 마지막으로, 효과 기술 개발의 가장 중요한 측면 중 하나는 아티스트들에게 그들이 일을 하는 데 필요한 도구를 주고, 우리가 우리 일을 하는 것을 도울 수 있다는 점이다. 무엇보다, 파티클은 우리 모두를 보기 좋게 만들어준다!

필자는 오리지널 래디컬 파티클 이펙트 시스템의 많은 부분을 만든 케빈 루즈(Kevin Loose)와 해롤드 웨스트런드(Harold Westlund)에게 감사를 표했다.

첨부 1: 애드-알파 쉐이더 코드(Add-Alpha Shader Code)

```
// Add-alpha pixel shader. To be used in conjunction
```

[5] 참조 링크: http://developer.amd.com/media/gpu_assets/R2VB_programming.pdf

```
// with the blend factors {One, InverseSourceAlpha}

    float4 addalphaPS(
    float4 vertexColour : COLOR0,
    float2 uvFrame0 : TEXCOORD0,
    float2 uvFrame1 : TEXCOORD1,
    float subFrameStep : TEXCOORD2 ) : COLOR

{

// Fetch both texture frames and interpolate

    float4 frame0 = tex2D( FXAtlasSampler, uvFrame0 );
    float4 frame1 = tex2D( FXAtlasSampler, uvFrame1 );
    float4 tex = lerp(frame0, frame1, subFrameStep);

// Pre-multiply the texture alpha. For alpha-blended particles,
// this achieves the same effect as a SourceAlpha blend factor

    float3 preMultipliedColour = tex.rgb * tex.a;
    float3 colourOut = vertexColour.rgb * preMultipliedColour;

// The vertex alpha controls whether the particle is alpha
// blended or additive; 0 = additive, 1 = alpha blended,
// or an intermediate value for a mix of both

    float alphaOut = vertexColour.a * tex.a;
    return float4( colourOut, alphaOut );
}
```