제 19 회 한국정보처리학회 춘계학술발표대회 논문집 제 10 권 제 1 호 (2003. 5)

강화된 인터페이스를 가진 파티클 에디터

방철*, 박종구*,안현식**
*성균관대학교 정보통신공학부, **국민대학교 전자정보통신공학부
bcheul@shinbiro.com

Particle System Editor with Strengthened Interface

Cheul Bang*, Jong-Koo Park*, Hyun-Sik Ahn**
*School of Info. & Comm. Eng., SungKyunKwan University
**School of Elec. Eng., Kookmin University

요 약

실제 게임에서는 파티클 시스템을 사용하여 대부분의 특수효과들(비, 불, 안개, 폭발, 분수 등...)을 만들어낸다. 파티클 효과를 간편하게 게임안에 삽입하기 위한 전용 에디터를 사용하기도 하는데, 이것을 좀더 편리하게 만들어 보고자 하는데 본 논문의 초점이 있다. 본 논문에서는 강화된 인터페이스를 가진 파티클 시스템을 만들어 보다 편리하게 효과를 생성하고 삽입하고자 한다. 기본적으로 매우 직관적인 효과들을 편리하게 표현해보고자 하는 의도에서 만들어진 것으로, 직관적으로 보이는 곡선을 그려 파티클의 이동경로를 미리 지정해주고 몇 가지 요소들을 제거함으로써 완전한 기존의 파티클 시스템과는 약간의 차별성을 두었다.

KeyWords : particle system, interface

1. 서론

파티클 시스템은 여러 개의 개별적인 조각(즉 파티클)들의 집합이다. 각 파티클은 개별적인 특성들을(속도, 색상, 수명 등등) 가지며, 다른 파티클과는 독립적인 방식으로 움직인다. 주어진 한 파티클 시스템의 파티클들은 일반적으로 공통의 특성들을 공유하므로, 개별 파티클이 독립적으로 움직인다고 해도 파티클들 전체는 하나의 공통적인 효과를 만들어 내게 된다. 파티클 시스템을 이용한 특수 효과의 대표적인 예는 폭죽의 불티들이 흩날리는 모습인데, 이 때 각 불티가 바로 파티클이다. 텍스쳐나 기타 속성들을 적절히 사용하면 불이나 연기, 폭발, 액체(물이나 피 등)의 분무, 눈발, 비행기의 연기 흔적 같은 효과들을 구현할 수 있다. 각각의 효과마다 개별적 파티클 시스템을 구현하는 것보다는 범용적이고 일반화된 파티클 시스템을 만들고 그것을 조정함으로써 개별적인 파티클 시스템을 구현하는 방식이 더 바람직하다.

파티클 시스템은 자기 완결적인 성질을 가지므로 객체 지향적 설계를 통해서 구현하는 것이 적합하다. C++ 메서드로 파티클 시스템을 구현할 때 가능한 접근 방식은 두 가지이다. 하나는 파티클 시스템 기반 메서드를 만들고 개별 효과를 위한 파티클 시스템들을 그 기반 메서드의 자식으로 하는 것이고, 또 하나는 하나의 범용 파티클 시스템 메서드를 만들고 그 메서드의 여러 속성들을 변경, 설정함으로써 개별적인 효과들을 구현하는 것이다. 둘 중 어떤 것을 선택하는가는 개인적인 취향 문제이나, 약간 더 유연하며 훨씬 깔끔한 전자의 방식이 선호된다.

기본적으로 가장 기본 요소인 파티클들에 대해서 생각해보면, 파티클을 구현하는 메서드가 필요로 하는 속성들은 위치, 속도, 수명, 크기, 무게, 표현, 색상, 소유자 등이 있다..

이와 같은 파티클 시스템을 여러 가지 요소들의 속성과 수치(value)를 조정함으로써 간편하게 하나의 효과를 만들어 낼 수 있게 만들어 놓은 것을 파티클 시스템 에디터(particle system editor)라 한다. 개발자들은

자신만의 파티클 시스템 에디터(이하 파티클 에디터라 칭함)를 간혹 만들어 사용하기도 하는데 보통 사용하는 파티클 에디터는 파티클 시스템에서 필요로 하는 변수(factor)들을 수치상으로 입력해서 사용하는 구성을 가지고 있다. 하지만 이러한 방식들은 매우 직관적이지 못한 약점을 지니고 있다.

이러한 점을 고려하여 본 논문에서는 강화된 인터페이스를 지닌, 다시 말하면 직관적인 형태를 가진 파티클 에디터를 구현해보고자 한다.

2. 기존의 파티클 시스템과 파티클 시스템 에디터

인터페이스를 강화한 파티클 에디터를 구현하기 전에 기존의 파티클 요소(factor)에 충실한 에디터를 만들어보았다.

파티클 시스템의 기본 요소인 파티클들에 대해서 생각해보면, 파티클을 구현하는 메서드가 필요로 하는 속성들은 위치, 속도, 가속도, 수명, 크기, 무게, 표현, 색상, 소유자 등이 있다.

이렇게 구현된 기본 파티클들을 제어하는 것이 파티클 시스템의 주된 기능이다. 파티클들은 개별적인 속성들을 가지고 개별적인 방식으로 움직이지만, 모든 파티클들이 공통으로 공유하는 속성들도 존재한다. 파티클 시스템이 관리해야 하는 것들은 파티클 목록, 위치, 방출 비율, 외부 힘들, 기본 파티클 속성들과 범위들, 현재, 혼합, 표현 등이 있다. 파티클 시스템이 이러한 여러가지 일들을 수행하기 위해서는 초기화, 갱신, 랜더링, 이동, 상태변경, 외부 힘의 설정/얻기와 같은 메서드들이 필요하다. 물론 이 외에도 필요한 메서드를 추가 할 수 있다.

이와 같은 여러 설계상의 내용들을 바탕으로 파티클 시스템을 구현할 수 있다. 여기에서 제시하는 방향은 최소한의 기능성을 담은 기반 클래스들을 작성한 다음, 그 기반 클래스들을 상속하는 구체적인 파티클 시스템의 구현 클래스들을 작성하는 것이다. 다음의 기반 클래스들을 그 자체로 직접 쓰이기 위한 것이 아니므로, 이 클래스들로부터 특정 효과에 맞는 새 클래스를 파생시켜야 한다. 결국 개별 파티클 대표 클래스와 입자 시스템의 기반 클래스로 크게 나뉘게 된다.

지금까지 기존의 파티클 시스템을 에디트(Edit)하기 위한 요소들을 설명하였다. 다음 절에서는 본 논문에서 제안하는 파티클 시스템 에디터는 어떤 구성을 가지는지 살펴보자.

3. 강화된 인터페이스를 가진 파티클 시스템 에디터

파티클 시스템 에디터를 구성하는 개략적인 모든 것은 위 절서 모두 설명하였다. 지금부터는 인터페이스를 개선하기 위해 무엇을 하였는지에 대하여 설명하도록 하겠다.

직관적인 인터페이스를 만들기 위하여 기존의 파티클의 속도나 가속도, 무게, 무게 변화량 등은 무시하였으며 방향, 속도, 가속도 등의 요소들은 다른 것으로 대체 하였다.

표 1. 기존 시스템과 본 논문의 시스템간의 Factor 비교표

파티클 시스템을 구성하는 Factor	개선된 파티클 시스템에서의 Factor
파티클의 현위치	유지
파티클의 이전위치	유지
방향	접선 벡터의 방향
속도, 가속도	직접적인 값을 입력받음
파티클의 에너지(수명)	유지
파티클의 크기	유지
시간에 다른 크기 변화량	유지
파티클의 무게	무시
시간에 따른 무게 변화량	무시
파티클의 색상	유지
파티클의 생상 변화량	유지

표 1 을 보면, 변화된 요소들은 방향, 속도, 가속도, 무게, 무게 변화량 임을 알 수 있다.

기본적인 인터페이스는 우선 곡선을 그리는 것에서 시작한다. Nurbs 를 이용하여 몇 개의 컨트롤 포인틀를 사용하여 곡선을 그리고 그려진 곡선은 파티클의 근사한 이동경로를 뜻한다. 그려진 곡선에서 일정한 간격을 가지고 접선 벡터를 구해내고 구해진 접선 벡터의 방향은 파티클의 진행방향이 된다. 그리고 속도와 가속도 부분은 좀더 실제적인 효과를 내기 위하여 제거 할 수 없는 부분인데 곡선자체에서는 속도와 가속도의 정보를 얻어오는 것이 불가능하므로 모든 컨트롤 포인트 구간에 따로 값을 입력할 수 있도록 하였다.

파티클의 무게와 시간에 따른 무게 변화량은 중력에 의해 받는 영향을 고려한 인자들이므로 본 시스템에서는 아무런 소용이 없다. 그러므로 이와 관련된 부분은 모두 제거를 하였다.

강화된 인터페이스를 가진 에디터에서 표 1 의 factor 들을 적절히 설정해서 파티클 시스템을 적절히 작동시키기 위해서는 다음과 같은 메서드들이 필요하다.

초기화 : 파티클 시스템 자체를 초기화 하는 메서드도 당연히 필요하다. 초기화 메서드에서는 앞에서 설명한 여러 속성들을 설정하고, 필요하다면 텍스쳐도 준비한다. 초기화에 쓰일 값들을 인자들로 받아들이게 할 수도 있다.

갱신 : 갱신을 위한 메서드는 이전의 갱신으로부터 흐른 시간 변화량을 인자로 받고 그 값에 기반해서 살아있는 모든 파티클들의 상태를 변경하거나 새로운 파티클들을 방출한다.

랜더링 : 파티클 시스템은 결국 그래픽적인 효과를 위한 것이므로 당연히 렌더링을 위한 메서드가 필요하다. 이 메서드에서는 각 파티클의 위치나 크기, 색상 등에 기반해서 모든 파티클들을 그리게 된다.

이동 : 파티클 시스템의 위치는 초기화 시점에서 결정되지만, 필요에 따라서는 그 위치를 옮겨야 할 수도 있다. 이를 위한 것이 이동 메서드다. 파티클

시스템의 현재 위치를 돌려주는 메서드로 추가하는 것이 좋다.

상태 변경 : 시스템의 상태를 변경하기 위한 메서드도 필요하다. 또한 현재 상태를 알려주는 메서드도 필요하다.

이제 본 시스템을 구성하는 클래스를 살펴보자.

개별 파티클을 대표하는 클래스 - 이 클래스는 외부에서 직접 접근할 수 있는 public 데이터 멤버들만 가질 것이므로 class 대신 struct 로 만들어도 아무 문제 없다.

```
struct particle_t
{
vector3_t m_pos; // 파티클의 현재 위치
vector3_t m_prevPos; // 파티클의 이전 위치
vector3_t m_direction; // 방향

float m_energy; // 파티클의 에너지-파티클의 수명을
결정한다.

float m_size; // 파티클의 크기
float m_sizeDelta; // 시간에 따른 크기 변화량

float m_color[4]; // 파티클의 현재 색상
float m_colorDelta[4]; // 시간에 따른 색상 변화량
};
```

그림 1. 개별 파티클을 대표하는 구조체

앞에서 이야기한 특성들이 모두 포함된 것은 아니다. 그 이유는 이것이 기반 클래스이기 때문이다. 특정 효과에 필요한 추가적인 특성들은 이 클래스로부터 파생된 자식 클래스들에 집어넣으면 된다. 모든 특성들을 기반 클래스에 집어넣으면 일이 편해지겠지만, 쓰이지 않는 특성들이 생길 수도 있기 때문에 메모리가 낭비될 가능성이 많다. 하나의 파티클 시스템은 상당히 많은 수의 파티클 객체들을 생성하기 때문에 메모리 낭비가 생각보다 매우 커질 수 있음을 주의해야 한다.

이 클래스에는 메서드가 하나도 없다. 어차피 파티클의 초기화는 파티클 시스템이 알아서 치리하기 때문이다. 앞에서 파티클의 개신 메서드에 대해 이야기하긴 했지만 그림 1 에서는 파티클의 갱신도 파티클 시스템이 담당한다. 갱신 메서드가 파티클 클래스에 있다면 매 프레임마다 수많은 메서드들이 호출되어야 하므로 속도에 나쁜 영향을 주게 된다.

파티클 시스템의 기반 클래스

```
class CParticleSystem
{
public:
CParticleSystem(int maxParticles, vector3_t origin);

// 가상함수들
virtual void Update(float elapsedTime) = 0;
virtual void Render() = 0;
virtual int Emit(int numParticles);
virtual void InitializeSystem();
virtual void KillSystem();

protected:
virtual void InitializeParticle(int index) = 0;
particle_t *m_particleList; // 입자들의 목록
int m_maxParticles; // 입자의 최대 갯수
int m_numParticles; // 활동 중인 입자들의 갯수
vector3_t m_origin; // 입자 시스템의 위치(중심)

float m_accumulatedTime; // 마지막 입자가 방출된 후 흐른
시간을 계산하는데 쓰인다.

vector3_t m_force; // 입자 시스템에 적용되는 외부
힘(중력, 바람 등 □ 사용자지정값).
};

//Particles.cpp
/************************************************
CParticleSystem::Constructor

입자 시스템의 기본값들을 설정한다.
************************************************/
CParticleSystem::CParticleSystem(int maxParticles,
vector3_t origin)
{
m_maxParticles = maxParticles;
m_origin = origin;
m_particleList = NULL;
} // CParticleSystem::Constructor 의 끝

/************************************************
CParticleSystem::Emit()

인자로 지정된 개수의 새 입자들을 생성하고 기본값들과
약간의 난수들을 이용해서 그 입자들을 초기화 한다. 난수는
오직 초기화에서만 쓰인다.
************************************************/
int CParticleSystem::Emit(int numParticles)
{
// numParticles 개의 새 입자들을 생성(최대 한도 안에서)
while (numParticles && (m_numParticles < m_maxParticles))
{
// 현재 입자를 초기화하고 활동 중인 입자 개수를 증가.
InitializeParticle(m_numParticles++);
--numParticles;
}
return numParticles;
} // CParticleSystem::Emit 의 끝

/************************************************
CParticleSystem::InitializeSystem()
 최대 개수만큼의 입자들에 대한 메모리를 할당
************************************************/
void CParticleSystem::InitializeSystem()
{
// 이미 메모리가 할당되어 있으면 해제한다.
if (m_particleList)
{
delete[] m_particleList;
m_particleList = NULL;};
```

```
// 최대 개수의 입자들을 할당
m_particleList = new particle_t[m_maxParticles];

// 활동 중인 입자 개수와 누적된 시간을 초기화 한다.
m_numParticles = 0;
m_accumulatedTime = 0.0f;
} // CParticleSystem::InitializeSystem의 끝


/***********************************************
CParticleSystem::KillSystem()
```

그림 2. 파티클 시스템의 기반 클래스

그림 2 는 하나의 추상 기반 클래스이므로 이 클래스를 직접 사용하는 것은 불가능하다. 이 클래스는 실제 파티클 시스템의 클래스를 파생시키기 위한 기반 클래스일 뿐이다. 이 클래스는 또한 파티클 시스템을 초기화하거나 정지시키는 기능성을 제공한다. 구체적인 파티클 시스템은 앞에서 설명한 입자 시스템 기반 클래스들을 상속해서 만든다.

4. 구현 결과

본 논문에서 제시한 파티클 시스템 에디터를 사용하여 구현한 결과는 그림 3 과 같다.

그림 3 을 살펴보면 곡선을 이용해 파티클의 이동경로를 미리 정해주었음에도 불구하고 매우 다양한 경로로 파티클이 이동하고 있음을 알 수 있다. 이 부분에 관한 처리는 매우 많은 곡선을 그림으로서 해결될 수 있지만 너무나 비 효율적이므로 곡선의 접선벡터에 일정 범위 안에서 랜덤함수를 적용하여 흔들리게 함으로써 해결하였다.

그림 3. 구현 결과

실제적으로 중력이나 속도, 가속도, 무게, 무게 변화량 등에 들어가는 비용이 매우 크고 이 부분에서 계산량이 대부분임을 감안하고 보았을 때, 비약적인 속도의 증가가 이루어 지지는 않았으나, 약 20~30% 정도의 속도 향상이 있었고, 간단한 곡선들로 표현할 수 있는 효과의 경우에는 매우 근접한 결과를 낼 수 있었다.

5. 결론 및 향후 연구 과제

인터페이스를 강화하기 위해서 버려야 할 요소가 너무 많았다는 점에서 크게 좋은 결과는 기대하지 않았지만 기존의 파티클 시스템과 비교했을 때 많이 어색하지 않은 결과를 도출 할 수 있었다. 몇 가지 요소를 버리거나 대체함으로써 기존의 파티클 시스템과 비교하여 장점과 단점이 발생하였다.

장점은 직관적인 진행방향을 그리고 시작한다는 점에서 효과의 전체적인 모습을 미리 예상할 수 있다는 점과 많은 비용이 소요되는 곳의 요소를 제거함으로써 속도의 향상이 있었다는 것이다.

단점으로는 본 에디터를 사용하여 만들어진 효과를 데이터로 변환하여 바로 게임엔진 안에 삽입할 수 있어야 하는데 몇 가지 요소의 제거와 변형으로 인하여 그냥 삽입이 불가능 하다는 점이다. 이를 해결하기 위해서는 본 에디터의 결과물을 기존의 파티클 에디터와 같은 형식으로 만드는 방법이 있고, 게임엔진 안에 별도의 파티클 구동 모듈을 삽입하여 본 에디터의 결과물을 바로 사용하도록 하는 방법이 있을 수 있다. 하지만 전자의 경우 변환에 필요한 정보들이 매우 부족하여 불가능하다 생각되고, 후자의 방법을 택함이 마땅하다. 그리고 인터페이스의 한계상 직관적인 효과가 아니라면 표현하기 힘들다는 단점이 있다. 예로서 폭발 효과 같은 경우에는 거의 만들어 내기 힘들다.

본 논문에서 시도한 방법은 리얼리티 부분과 표현 효과의 다양성 부분에서 부족함이 많이 있으나 속도 개선의 문제나 자연현상에 충실한 기존의 파티클 시스템보다는 게임 안에서 이용되는 효과라는 점에서 좀더 코스트를 줄이는 방법의 기초가 될 수 있다는 점에 의미를 둘 수 있으며 이용자가 약간의 노력을 곁들이고 단점을 보완한다면 효과적이고 직관적으로 에디트 할 수 있는 환경을 만들어 줄 것이라 생각한다.

참고문헌

[1] Kevin Haskins, Dave Astle, Andre LaMothe 「OPENGL GAME PROGRAMMING」 류광 역

[2] http://dip2k.coco.st/

[3] http://www.gamedev.net/reference/articles/article677.asp

[4] http://www.gamedev.net/opengl/volfog.html

[5] http://www.nar-associates.com/nurbs/nurbs.html

[6] http://nexe.gamedev.net/News/News.asp

[7] http://nehe.gamedev.net/

[8] Theroy and Applications , Eric J. Stollnitz , Tony D. DeRose , David H. Salesin 「Wavelets for Computer Graphics」