운동과 파티클 효과(Particle Effect)

3D 게임 프로그래밍에서 그래픽이나 기획의 도움 없이 재미있게 혼자서 프로그래밍을 할 수 있는 분야가 파티클 효과(Particle Effect: 파티클 이펙트)입니다. 게임에서 파편이 터지거나 캐릭터의 레벨 증가와 같은 장면에서 파티클이 사용되는데 파티클 효과는 자체의 효과보다 다른 이펙트와 잘 어울려야 표현에 대한 상승효과를 만들어 낼 수 있고, 그렇지 않으면 잘 만들어 놓고 오히려 눈에 거슬릴 수 있습니다. 따라서 하나의 파티클을 만들었으면 반드시 게임 장면에서 테스트를 거쳐야 합니다.
파티클 효과의 구성을 억지로 나눈다면 순수 파티클로 구성된 효과, 파티클의 복합으로 구성된 효과, 파티클의 운동에 2D 텍스처 애니메이션을 혼합한 효과 등으로 나누어 볼 수 있으며, 게임에서는 주로 파티클의 혼합과 2D 텍스처 애니메이션을 결합한 형태가 가장 많이 사용됩니다.
파티클 효과를 섞어서 사용하면 또한 파티클의 사용을 관리해주는 파티클 매니저가 필요합니다. 파티클을 많이 사용하면 할수록 멋진 장면을 연출할 수 있지만 컴퓨터의 자원은 한정적이므로 이에 대한 관리는 필 수 이며, 대부분 파티클 매니저 클래스를 두어 관리를 합니다.

파티클 제작은 뉴턴의 운동방정식에서 출발을 합니다. 먼저 간단한 형태의 뉴턴 방정식을 살펴보도록 합시다.

1. 뉴턴 방정식

뉴턴이 정의한 운동의 법칙, 제1법칙 관성의 법칙, 제2법칙 가속도의 법칙, 제3법칙 작용 반작용의 법칙은 초등학교 때부터 누누이 들어서 잘 알고 있다고 판단하고 자세한 설명은 생략하겠습니다. 그런데 뉴턴의 제1법칙은 제2법칙으로도 충분히 설명이 가능하므로 실제로 뉴턴의 법칙은 2와 3법칙, 두 개로 정의할 수 있습니다.

제2법칙을 기준으로 위치, 속도, 가속도에 대한 정의를 한다면 물체의 위치(x, y, z)를 벡터로 $p(x,y,z)=\vec{p}$ 라 하고 시간을 t라 가정한다면 속도는 다음과 같이 정의합니다.

$$\vec{v}=\lim_{t\to t_0}\frac{\vec{p}(t)-\vec{p}(t_0)}{t-t_0}=\frac{\Delta\vec{x}}{\Delta t}(\Delta t\to 0)=\frac{d\vec{x}}{dt}$$

즉, 속도는 위치의 변화를 시간의 변화로 나눈 값이라 할 수 있습니다.
여기서 Δt는 위치와 대응하는 시간의 차이로 Delta(델타) t 로 읽습니다. 가속도는 시간의 변화에 따른 속도의 변화로 정의합니다.

$$\vec{a} = \lim_{t \to t_0} \frac{\vec{v}(t) - \vec{v}(t_0)}{t - t_0} = \frac{\Delta \vec{v}}{\Delta t}(\Delta t \to 0) = \frac{d\vec{v}}{dt}$$

힘은 질량 * 가속도로 정의합니다.

$$\vec{F} = m * \vec{a}$$

dt는 현실에서 측정이 되지 않으므로 Δt를 이용하며 근사(approximation)을 통해서 속도와 가속도를 정의 합니다.

$$\vec{v} \approx \frac{\Delta \vec{x}}{\Delta t} \qquad \text{또는} \qquad \vec{v} \sim \frac{\Delta \vec{x}}{\Delta t}$$
$$\vec{a} \approx \frac{\Delta \vec{v}}{\Delta t} \qquad\qquad \vec{a} \sim \frac{\Delta \vec{v}}{\Delta t}$$

여기서 '≈' 은 '매우 비슷하다' 라는 뜻이고, '~' 은 '비슷하다' 는 의미입니다.
만약 가속도를 알고 있다면 적분을 이용해서 속도를 구할 수 있습니다.

$$d\vec{v} = \vec{a} * dt$$
$$\vec{v} - \vec{v}_0 = \int_{t=t_0}^{t} \vec{a} * dt$$

이것은 이상적인 모습이지 현실에서 구하기가 어렵습니다. 그래서 아주 짧은 시간에는 가속도가 일정하다고 보고 근사식을 이용합니다.

$$\Delta \vec{v} = \vec{a} * \Delta t$$
$$\vec{v} - \vec{v}_0 = \vec{a} * (t - t_0)$$
$$\therefore \vec{v} = \vec{a} * (t - t_0) + \vec{v}_0$$

공식에 가까워 지도록 하기 위해 t와 t0의 시간 차이를 가능한 작게 만들도록 노력을 합니다. 이상적인 위치를 구하기 위해서는 적분을 사용해야 합니다.

$$d\vec{p} = \vec{v} * dt$$
$$\vec{p} - \vec{p}_0 = \int_{t=t_0}^{t} \vec{v} * dt$$

가속도에서 속도를 계산 할 때와 마찬가지로 근사식을 가지고 위치를 구합니다.

$$\Delta\vec{p} = \vec{v} * \Delta t \therefore$$
$$\vec{p} - \vec{p}_0 = \vec{v} * (t - t_0)$$
$$\therefore \vec{p} = \vec{v} * (t - t_0) + \vec{p}_0$$

주어진 시간 t와 t0사이에서 a(t), v0, p0가 주어진다면 가속도에서 속도를 구하고 속도에서 최종 위치를 구할 수 있습니다.

$$d\vec{v} = \vec{a} * dt$$
$$\vec{v} - \vec{v}_0 = \int_{t=t_0}^{t} \vec{a} * dt$$
$$\vec{v} = \int_{t=t_0}^{t} \vec{a} * dt \ + \ \vec{v}_0$$
$$d\vec{p} = \vec{v} * dt$$
$$\vec{p} - \vec{p}_0 = \int_{t=t_0}^{t} \vec{v} * dt$$
$$\vec{p} = \int_{t=t_0}^{t} \vec{v} * dt \ + \ \vec{p}_0$$
$$\vec{p} = \int_{t=t_0}^{t} \left(\int_{t=t_0}^{t} \vec{a} * dt \ + \ \vec{v}_0 \right) * dt \ + \ \vec{p}_0$$

괄호를 풀어서 가속도는 이중 적분, 속도는 적분 형태로 변경합니다.

$$\vec{p} = \int_{t=t_0}^{t} \left(\int_{t=t_0}^{t} \vec{a} * dt \ + \ \vec{v}_0 \right) * dt \ + \ \vec{p}_0$$
$$\vec{p} = \int_{t=t_0}^{t} \int_{t=t_0}^{t} \vec{a} * dt * dt \ + \ \int_{t=t_0}^{t} \vec{v}_0 * dt \ + \ \vec{p}_0$$

v0는 주어진 시작속도이고 상수 이므로 최종 위치는 가속도에 대해서 이중 적분, 시간과 속도의 곱, 그리고 시작 위치로 결정 됩니다.

$$\vec{p} = \int_{t=t_0}^{t} \int_{t=t_0}^{t} \vec{a} * dt * dt \; + \; \vec{v}_0 * \int_{t=t_0}^{t} dt \; + \; \vec{p}_0$$

$$\therefore \vec{p} = \int_{t=t_0}^{t} \int_{t=t_0}^{t} \vec{a} * dt * dt \; + \; \vec{v}_0 * (t - t_0) \; + \; \vec{p}_0$$

만약 가속도가 일정하다면 이중 적분에서 가속도가 상수가 되므로 수식은 좀 더 간단해집니다.

$$\vec{p} = \vec{a} * \int_{t=t_0}^{t} \int_{t=t_0}^{t} dt * dt \; + \; \vec{v}_0 * (t - t_0) \; + \; \vec{p}_0$$

$$\vec{p} = \vec{a} * \left(\frac{1}{2} (t - t_0)^2 \right) \; + \; \vec{v}_0 * (t - t_0) \; + \; \vec{p}_0$$

$$\therefore \vec{p} = \frac{1}{2} * \vec{a} * (t - t_0)^2 \; + \; \vec{v}_0 * (t - t_0) \; + \; \vec{p}_0$$

여기에 시작 시간을 t0=0 으로 정한다면 최종 위치는 좀 더 간결하게 구할 수 있습니다.

$$\vec{p} = \frac{1}{2} * \vec{a} * t^2 \; + \; \vec{v}_0 * t \; + \; \vec{p}_0$$

이 공식을 이용해서 시작 위치 p0, 시작 속도 v0, 가속도 a가 주어지면 시간에 따른 위치 p를 구할 수 있습니다. 게임에서 1/2 값은 가속도에 포함 시켜서 다음과 같은 식을 사용하기도 합니다.

$$\vec{p} = \vec{a}' * t^2 \; + \; \vec{v}_0 * t \; + \; \vec{p}_0$$

이 공식을 이용하기 보다는 다음과 같은 순서를 통해서 최종 위치를 구하는 것이 좋다고 봅니다.

1. 시간에 대한 가속도를 갱신(Update) 한다.
2. 가속도에 대한 속도를 갱신한다.
3. 속도에 대한 위치를 갱신한다.

이들은 다음과 같이 벡터로 표현됩니다.

$$\Delta \vec{v} = \vec{a} * \Delta t$$
$$\Delta \vec{p} = \vec{v} * \Delta t$$

로 되고 이것을 다시 의사 코드(pseudo-code: 프로그램으로 작성되기 위한 논리적인 가상코드)로 작성한다면 다음과 같이 정리될 수 있습니다.

```
update a(x,y,z);
v(x,y,z) += a(x,y,z) * t;
p(x,y,z) += v(x,y,z) * t;
```

여기서 update a(x,y,z)는 가속도의 집합으로 공기저항, 마찰력, 횡력, 항력, 등 파티클의 운동에 영향을 주는 모든 가속도를 더한 값입니다. 또한 시간 t는 장면의 평균 프레임(Frame)으로 정합니다.
이러한 가정이 현실의 장면과 유사한지 코드로 구현해 봅시다. 먼저 파티클의 운동을 결정하는 변수들을 설정합니다.

```
D3DXVECTOR3     m_IntP;         // 초기 위치
D3DXVECTOR3     m_IntV;         // 초기 속도
D3DXVECTOR3     m_IntA;         // 초기 가속도

D3DXVECTOR3     m_CrnP;         // 현재 위치
D3DXVECTOR3     m_CrnV;         // 현재 속도
D3DXVECTOR3     m_CrnA;         // 현재 가속도
```

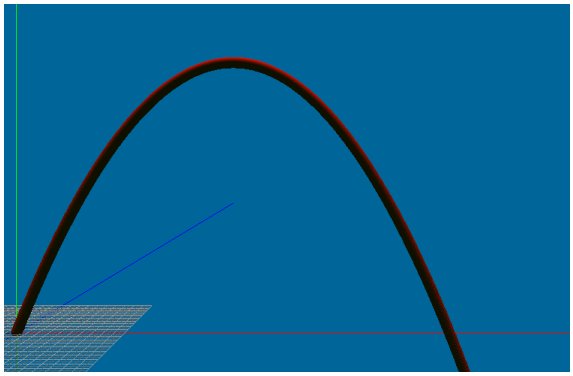
다음으로 애니메이션이 활성화 되면 초기에 주어진 값들을 현재의 값들에 설정합니다.

```
// 초기 위치, 속도, 가속도를 현재의 값들의 초기 값으로 설정
m_CrnP = m_IntP;                // 초기 위치
m_CrnV = m_IntV;                // 초기 속도
m_CrnA = m_IntA;                // 초기 가속도
```

매번 프레임을 돌면서 가속도, 속도, 위치 등을 순서대로 갱신합니다.

```
INT CMcParticle::FrameMove()
...
FLOAT   ftime = m_fTimeAvg * 0.1f;      // 평균 시간(delta time) 설정
...
m_CrnA  = m_CrnA;                       // 1. 가속도 갱신
m_CrnV +=m_CrnA * ftime;                // 2. 현재 속도 갱신
```

```
m_CrnP +=m_CrnV * ftime;                    // 3. 현재 위치 갱신
```



<공기 저항, 충돌이 없는 입자의 운동: prt11_newton.zip>

입자가 바닥(y = 0)과 충돌(입자의 위치<=0)하고 탄성계수에 의한 감속을 구현해 봅시다. 법선 벡터 N과 원점에서 최단거리 -d로 구성된 평면과 입자가 충돌할 때 완전 탄성 충돌의 경우 운동 방향 V벡터에 대한 반사 벡터 V'와 평면을 뚫고 지나간 P의 위치에 대한 반사 위치 P'는 다음 수식으로 계산이 됩니다.

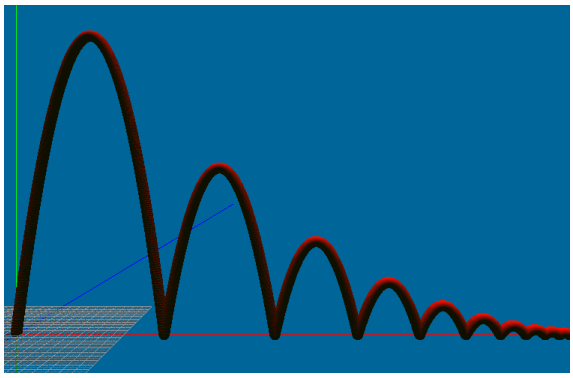
$$\vec{V}' = \vec{V} - 2(\vec{V} \cdot \hat{N})\hat{N}$$
$$\vec{P}' = \vec{P} - 2(\vec{P} \cdot \hat{N} + d)\hat{N}$$

바닥은 (0,1,0) 법선과 d= 0인 평면이므로 이 수식에 대입해서 풀면 다음과 같습니다.

```
V'  = (V.x, V.y, V.z) - 2 * V.y * (0,1,0)      = (V.x, -V.y, V.z)
P'  = (P.x, P.y, P.z) - 2 * (P.y +0) * (0,1,0) = (P.x, -P.y, V.z)
```

즉, 바닥에 충돌하게 되면 위치와 속도의 y 성분을 반대 방향으로 설정하면 되고, 탄성 계수는 속도에 곱하는 과정을 구현하면 됩니다.

```
INT CMcParticle::FrameMove()
...
if(m_CrnP.y<0.f)                    // 4. 경계 값 설정
{
        m_CrnV.y *= -1.0f;          // 속도의 진행의 방향을 바꾼다.
        m_CrnV.y *= m_fElst;        // 속도에 탄성 계수를 곱한다.
        m_CrnP.y *= -1.0f;          // y의 위치를 반대 방향으로 정한다.
}
```



<탄성 계수가 적용된 입자의 바닥 충돌: prt12_newton_bound.zip>

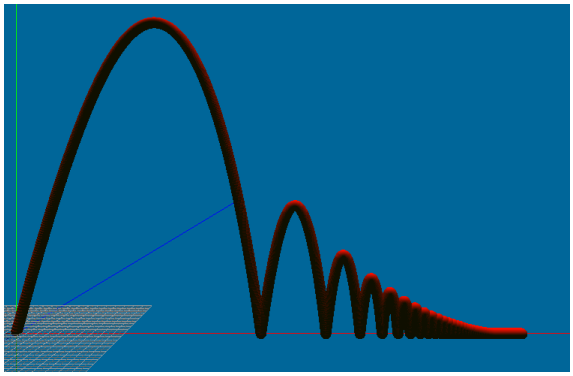
간단한 충돌을 해봤으니 공기 저항도 구현해 봅시다. 공기 저항은 일반적으로 속력(속도의 크기)의 제곱과 저항을 받는 면적에 비례한다고 합니다.

공기 저항 $=\frac{A}{m}|\vec{v}|^2$, 또는 공기 저항 $=\frac{A}{m}|\vec{v}|$

여기서는 공기저항이 속력의 제곱에 비례하는 부분을 적용해 보도록 하겠습니다.
위의 공식에서 공기저항의 방향은 물체의 이동방향과 반대입니다. 물체의 이동 방향은 속도의 방향과 일치하므로 속도를 복사해서 단위벡터로 만들고, 여기에 '-1'을 곱하게 되면 공기저항의 방향으로 만들어 집니다. 그 다음, 속도 크기의 제곱을 곱하고, 공기저항 상수를 곱하면 구하고자 하는 공기저항이 됩니다.
여기서는 면적과 질량은 각각 1로 설정하고 다음과 같이 공기저항을 구해서 가속도를 갱신합니다.

```
INT CMcParticle::FrameMove()
...
// 공기저항 방향 벡터
D3DXVECTOR3      vcAirR = m_CrnV;

// 속도 제곱 = Vx*Vx + Vy*Vy + Vz*Vz
FLOAT    fLenV  = D3DXVec3LengthSq(&vcAirR);

// 공기저항 방향 벡터 정규화
D3DXVec3Normalize(&vcAirR, &vcAirR);

// 이동 속도와 반대로 설정
vcAirR  *= -1.0F;
```

```
// 최종 공기 저항 = 속력제곱 * 공기 저항 계수
vcAirR  *= fLenV * m_fDamp;

// 초기 가속도에 공기저항을 더한다.
m_CrnA  = m_IntA + vcAirR;
...
```



prt13_newton_acc.zip

2. 좌표계(Coordinate System)

좌표계의 선택은 파티클의 운동에 대한 계산을 편리하게 만들 수 있습니다. 예를 들어 x, y로 구성된 2차원 좌표계에서 (0,0)을 중심으로 반경(반지름)이 r인 점들을 일정한 간격으로 배치한다고 합시다.

x*x + y*y = r*r 이므로 x가 정해지면 y = sqrt(r * r - x * x)으로 구할 수 있습니다.

이것을 화면에 출력하면 x와 y는 각각의 축에 일정한 간격을 유지할 뿐 원래의 목적인 원호(arc)에서 점들의 간격이 일정하지 않게 분포하는 것을 알 수 있습니다.

이것을 다음과 같은 삼각함수를 이용하면 원하는 결과를 쉽게 얻을 수 있습니다.

```
Θ  = 3.141592654f * Angle/180.f;
x= r * cos(Θ)
y= r * sin(Θ)
```

x와 y를 직접 구하지 않고 r과 Θ를 먼저 설정하고 삼각 함수 등을 사용해서 x, y, z등을 결정하

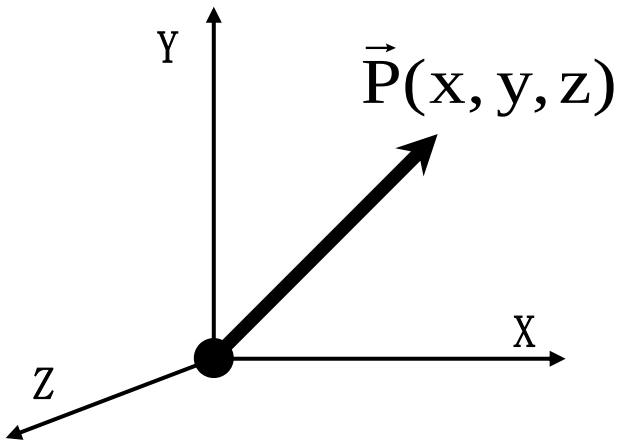
는 경우가 게임 프로그램에서 종종 있습니다. 첨언한다면 우리가 중학교 때부터 무심코 사용했던 앞의 공식은 2차원의 경우 극 좌표(Polar Coordinate)계라 불리는 좌표계를 사용해왔던 것입니다. 극 좌표계는 2차원에서 점에 대칭일 경우에 유리합니다.
이 장에서는 여러 가지 좌표계에서 게임에서 유용하게 사용할 수 있는 좌표계는 직각 좌표계, 원통 좌표계, 구 좌표계 3가지에 대한 입자 운동을 살펴보겠습니다.

2.1 데카르트 좌표계

직각 좌표계는 데카르트 좌표계(Cartesian coordinate)라 하며 3차원의 경우 x, y, z 로 구성된 좌표계입니다. 이 좌표계의 특징은 어떠한 부분에서도 대칭이 없는 경우에 적당합니다. 따라서 게임 프로그램에서 가장 보편적으로 사용되는 좌표계입니다.

다음 그림은 오른손 좌표계를 중심으로 데카르트 좌표계를 표현한 그림입니다. 좌표계는 왼손 좌표계, 오른손 좌표계로 크게 분류되는데, 분류 방법은 x와 y를 오른손으로 감아 쥐었을 때 오른손의 엄지 방향과 z의 방향이 일치하면 오른손 좌표계가 되고 그렇지 않으면 왼손 좌표계가 됩니다. 참고로 수학, 물리학 등 자연과학과 공학, CAD 등에서는 전부 오른손 좌표계를 사용합니다. 게임에서는 OpenGL, Max, Maya 등이 오른손 좌표계를 사용하고, 왼손 좌표계를 사용하는 경우는 DirectX와 게임 엔진 등에서 지원되는 경우가 종종 있습니다.

일반적으로 위치를 x, y, z 라고 쓰면 이것은 데카르트 좌표계를 사용하고 있음을 암시합니다.



<데카르트 좌표계>

앞서 구현한 prt13_newton_acc.zip 파티클 예제도 데카르트 좌표계를 사용했습니다. 이 예제에 좀 더 많은 입자들을 넣기 위해서 각각의 입자에 대한 구조체를 정의가 필요합니다.

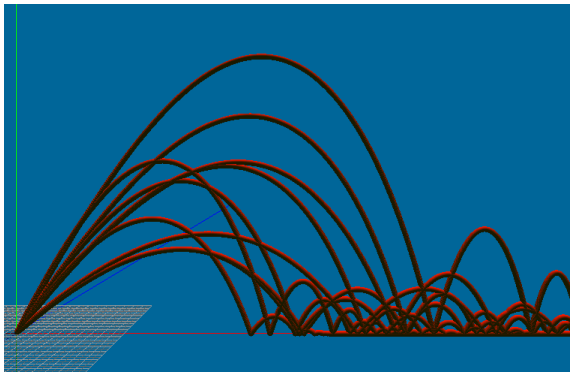
```
struct Tpart
```

```
{
        D3DXVECTOR3     m_IntP;         // 초기위치
        D3DXVECTOR3     m_IntV;         // 초기속도
        D3DXVECTOR3     m_IntA;         // 초기기가속도
        D3DXVECTOR3     m_CrnP;         // 현재위치
        D3DXVECTOR3     m_CrnV;         // 현재속도
        D3DXVECTOR3     m_CrnA;         // 현재가속도
        FLOAT           m_fElst;        // 탄성(Elastic) 계수
        FLOAT           m_fDamp;        // 공기저항(Air Registance) 계수
};
```

이 구조체로 구성된 여러 개의 파티클을 초기화 하는 함수를 다음 같이 만들 수 있습니다. 이 예제는 데카르트 좌표계의 응용으로써 x, y, z에 대한 각각의 속도와 탄성 계수, 공기저항 등을 Random 함수를 이용해 설정했습니다.
그리고 애니메이션을 설정하는 함수 안의 초기 속도, 위치, 탄성 계수, 공기저항 계수를 설정하는 부분을 살펴보면 각각 x, y, z에 대해서 직접 설정하고 있음을 볼 수 있습니다.

```
void CMcParticle::SetAni(BOOL bAni)
...
for(int i=0; i<m_PrtN; ++i)
{
        CMcParticle::Tpart* pPrt = &m_PrtD[i];

        // 초기가속도
        pPrt->m_IntA = D3DXVECTOR3(0, -.1F,0);

        // 초기속도
        pPrt->m_IntV.x = (50 + rand()%51)*0.1f;
        pPrt->m_IntV.y = (50 + rand()%101)*0.1f;
        pPrt->m_IntV.z = 0.f;

        // 초기위치
        pPrt->m_IntP = D3DXVECTOR3(0, 0.F, 0);

        // 탄성계수설정
        pPrt->m_fElst= (50 + rand()%51)*0.01f;
```

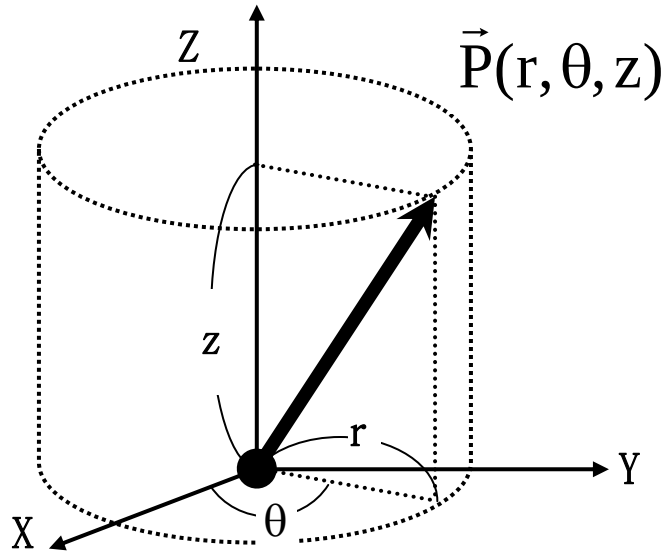
```
        // 공기저항계수
        pPrt->m_fDamp= (100 + rand()%101)*0.00001F;

        // 초기위치, 속도, 가속도를 현재의 값들의 초기값으로 설정
        pPrt->m_CrnP = pPrt->m_IntP;
        pPrt->m_CrnV = pPrt->m_IntV;
        pPrt->m_CrnA = pPrt->m_IntA;
}
...
```



<데카르트 좌표계 입자 운동: prt21_coord_cartesian.zip>

2.2 원통 좌표계(Cylinder coordinate)

원통 좌표계는 그림과 같이 데카르트 좌표계에서 2개의 좌표축을 r과 Θ로 표현하는 좌표계입니다. 이 좌표계의 특징은 직선에 대해서 대칭인 경우에 유리합니다. 예를 들어 게임에서 주인공의 위치에서 주위에 일정한 반경으로 힐링(Healing)을 표현하는 파티클 효과를 만들 경우 이 좌표계를 사용하면 편리합니다.



<원통 좌표계>

점 p(r, Θ, z)를 데카르트 좌표계로 표현하면 다음과 같이 됩니다.

x = r * cosΘ, y = r * sinΘ, z = z

반대로 x, y, z를 r, Θ, z로 전환화면 다음과 같이 됩니다.

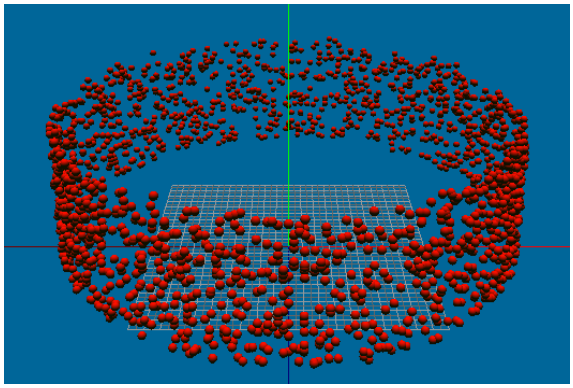
$$r = \sqrt{x^2 + y^2},\ \theta = \tan^{-1}\left(\frac{y}{x}\right),\ z = z$$

위 식에서 각각의 범위는 r:[0, ∞], Θ[0, 2π], z:[-∞, +∞] 입니다.

원통 좌표계를 이용한 파티클 효과의 예는 같은데 앞서 구현한 prt21_coord_cartesian.zip 예제의 애니메이션을 설정하는 함수 안에서 초기 속도와 위치를 이 좌표계를 이용해서 설정하고 있음을 볼 수 있습니다. 주의 해야 할 것은 이론과 코드의 구현에서 Z와 Y의 축이 바뀌어 있는데 이것은 수학과 과학에서는 Z축이 대부분 위를 향하기 때문입니다.

```
// 초기 속도
pPrt->m_IntV.x = fSpdR * cosf(fAngle);
pPrt->m_IntV.y = fSpdY;
pPrt->m_IntV.z = fSpdR * sinf(fAngle);

// 초기 위치
pPrt->m_IntP.x = 200.f * cosf(fAngle);
```

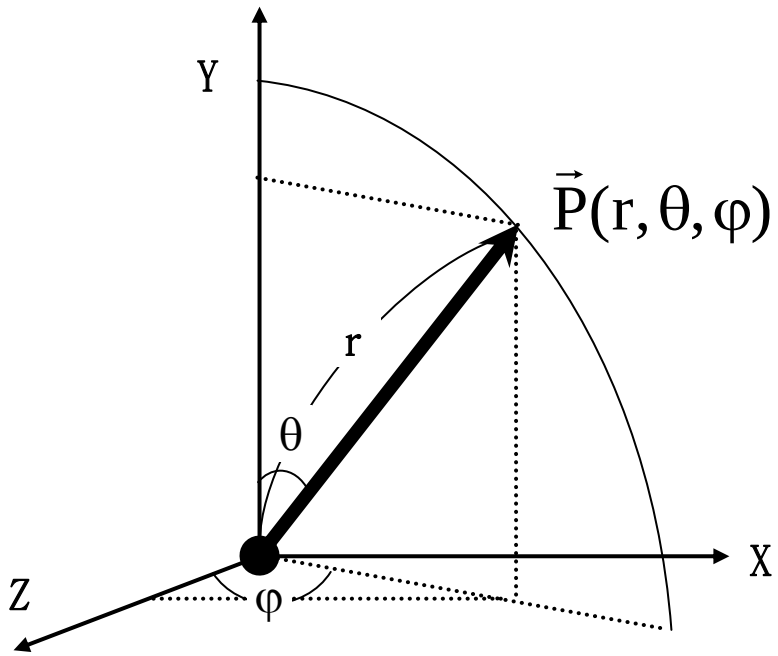
```
pPrt->m_IntP.y = 0.f;
pPrt->m_IntP.z = 200.f * sinf(fAngle);
...
```



prt22_coord_cylinder.zip

2.3 구 좌표계(Spherical coordinate)

구 좌표계는 다음 그림과 같이 점 p의 좌표를 p(r, Θ, ϕ) 표현한 좌표계입니다. 이 좌표계는 점(Point)에 대칭이 있는 경우에 유리합니다. 예를 들어 폭발과 같은 파티클 효과를 연출할 때 폭발의 중심에서 임의로 균일하게 연출하기 위해서 좌표계를 사용하면 편리합니다.



<구 좌표계>

구 좌표계로 표현한 p(r, Θ, ϕ)를 데카르트 좌표계로 표현하면 다음과 같이 됩니다.

z = r * sinΘ * cosϕ, x = r * sinΘ * sinϕ, y = r*cosΘ

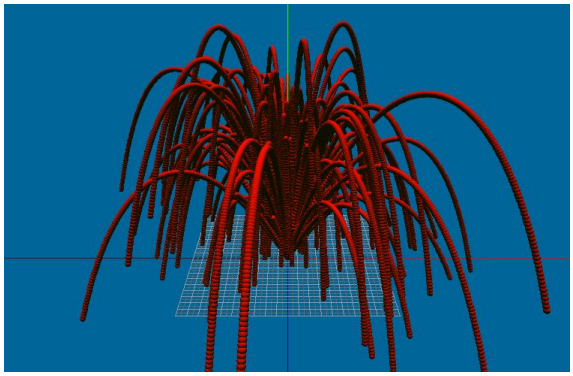
반대로 x, y, z를 r, Θ, ϕ로 전환화면 다음과 같이 됩니다.

$$r = \sqrt{x^2 + y^2 + z^2},\ \theta = \tan^{-1}\left(\frac{\sqrt{z^2 + x^2}}{y}\right),\ \varphi = \tan^{-1}\frac{x}{z}$$

위 식에서 각각의 범위는 r:[0, ∞], Θ[0, π], ϕ:[0, 2π] 입니다.

구 좌표계를 이용한 파티클 효과의 예는 폭발, 분수 등에서 볼 수 있는데 다음의 예는 구 좌표계를 이용한 분수를 표현한 예제입니다. 애니메이션을 설정하는 함수 안에서 초기 속도와 위치를 이 좌표계를 이용해서 설정하고 있음을 볼 수 있습니다.

```
//초기 속도와 위치를 설정하기 위한 변수
fTheta  = float(rand()%61);
fTheta  -=30.f;

fPhi    = float(rand()%360);

fSpdR = 100.f + rand()%101;
fSpdR *=0.1f;

// 라디안으로 변경
fTheta  = D3DXToRadian(fTheta);
fPhi    = D3DXToRadian(fPhi);

// 초기 속도
pPrt->m_IntV.x = fSpdR * sinf(fTheta) * sinf(fPhi);
pPrt->m_IntV.y = fSpdR * cosf(fTheta);
pPrt->m_IntV.z = fSpdR * sinf(fTheta) * cosf(fPhi);
...
```

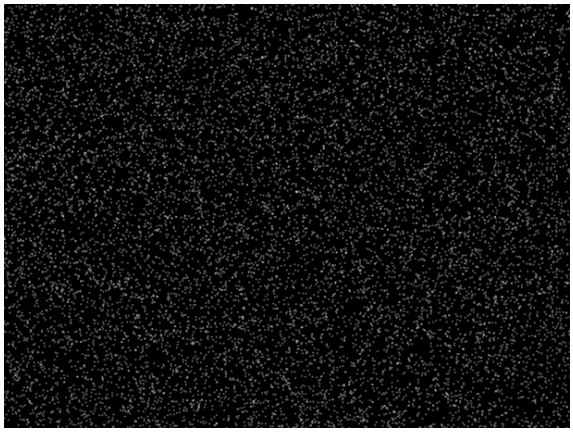


prt23_coord_spherical.zip

3. 파티클 기타

3.1 Point Sprite

파티클의 운동과 좌표계를 살펴보았습니다. 이제는 이것을 배경으로 효과적인 연출을 연습할 차례입니다. 먼저 포인트 스프라이트를 살펴보겠습니다. 포인트 스프라이트는 파티클 효과와 같은 영역에서 주로 사용되는데 파티클을 표현하는 정점에 텍스처를 사용하려면 하나 이상의 삼각형이 필요하므로 최소한 3개 이상의 좌표가 필요합니다. 하지만 포인트 스프라이트는 하나의 정점에 텍스처를 붙일 수 있어서 정점의 사용량을 1/4 정도로 줄일 수 있고, 하나의 파티클과 하나의 정점이 1:1로 대응 되어 프로그램으로 구현하기가 무척 수월해집니다.



<포인트 스프라이트를 사용한 노이즈 효과: prt31_point_sprite.zip>

포인트 스프라이트는 정점 버퍼만을 이용하기 때문에 CreateVertexBuffer() 함수로 정점 버퍼를 파티클의 수만큼 만듭니다.

```
LPDIRECT3DVERTEXBUFFER9 m_pVB;
...
```

```
m_pDev->CreateVertexBuffer(m_PrtN* sizeof(CMcParticle::VtxD)
                , D3DUSAGE_POINTS
                , CMcParticle::VtxD::FVF
                , D3DPOOL_MANAGED
                , &m_pVB, 0);
```

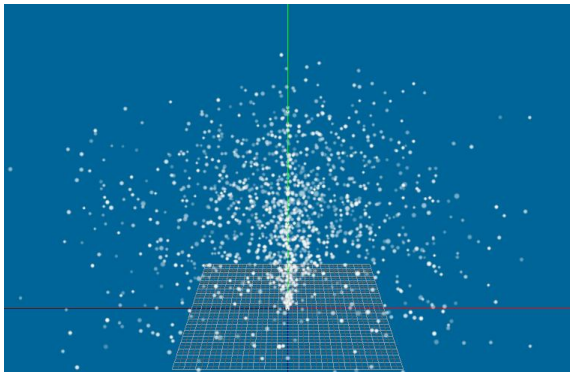
정점 버퍼를 만들었으면 파티클의 운동을 갱신한 데이터를 이용해서 정점 버퍼를 갱신합니다.

```
CMcParticle::VtxD*      pVtx;
m_pVB->Lock(0,0,(void**)&pVtx, 0);

for(i=0; i<m_PrtN; ++i)
{
        CMcParticle::Tpart* pPrt = &m_PrtD[i];

        pVtx[i].p = pPrt->m_CrnP;
        pVtx[i].d = pPrt->m_dColor;

m_pVB->Unlock();
```

갱신한 정점 버퍼를 화면에 출력합니다.

```
m_pDev->SetRenderState(D3DRS_POINTSPRITEENABLE, TRUE);
m_pDev->SetRenderState(D3DRS_POINTSCALEENABLE, TRUE);
m_pDev->SetRenderState(D3DRS_POINTSIZE, FtoDW(20.f));
m_pDev->SetRenderState(D3DRS_POINTSIZE_MIN, FtoDW(5.0f));

m_pDev->SetRenderState(D3DRS_POINTSCALE_A, FtoDW(1.0f));
m_pDev->SetRenderState(D3DRS_POINTSCALE_B, FtoDW(2.0f));
m_pDev->SetRenderState(D3DRS_POINTSCALE_C, FtoDW(3.0f));

m_pDev->SetTexture(0, m_pTx);
m_pDev->SetStreamSource(0, m_pVB, 0, sizeof(CMcParticle::VtxD));
m_pDev->SetFVF(CMcParticle::VtxD::FVF);

m_pDev->DrawPrimitive(D3DPT_POINTLIST, 0, m_PrtN);
...
```

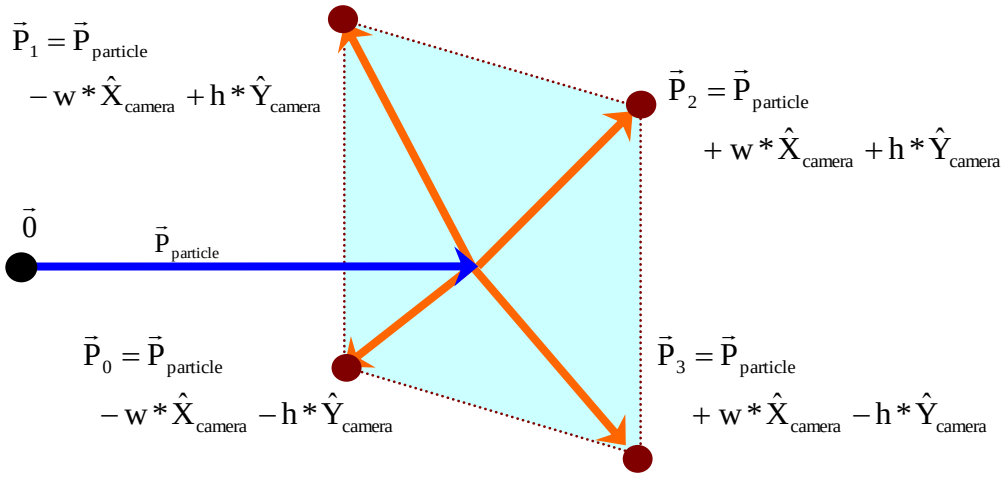
정점의 색상을 파티클의 색상으로 맞추었는데 이렇게 하기 위해서는 파티클에 색상을 포함해야 합니다. 파티클의 색상은 앞으로 파티클의 생명과 관련이 있으므로 반드시 파티클 구조체에 포함시켜야 합니다.



prt31_point_sprite.zip

3.2 빌보드(Bill Board)

포인트 스프라이트는 정점을 줄여 기억공간을 절약하는 이점이 있지만 정점 버퍼를 사용하므로 비디오 메모리를 소모한다고 볼 수 있습니다. 또한 파티클에 텍스처 애니메이션을 설정할 수 없어 좀 더 다양한 표현에 제약이 있어 게임에서는 주로 빌보드를 이용해서 파티클을 표현합니다.



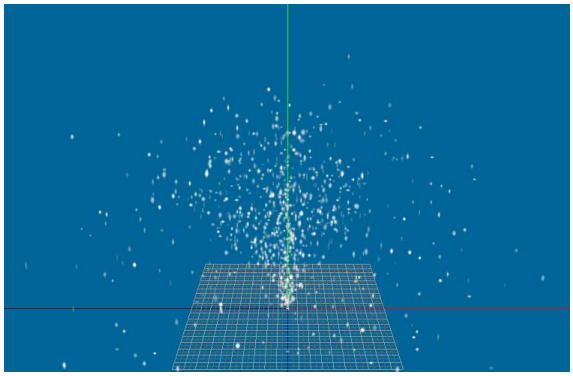
<카메라의 축을 사용한 빌보드 좌표>

빌보드는 항상 카메라의 파티클의 구조체에 빌보드를 위해 파티클의 구조체에 빌보드의 폭과 높이를 위한 표현 요소를 추가 시키고 정점 데이터를 갱신하는 부분에 빌보드를 설정합니다.

```
// 입자의 운동을 Billboard에 연결
```

```
D3DXMATRIX mtView;
m_pDev->GetTransform(D3DTS_VIEW, &mtView);
D3DXVECTOR3 vcCamX(mtView._11, mtView._21, mtView._31);
D3DXVECTOR3 vcCamY(mtView._21, mtView._22, mtView._32);

for(i=0; i<m_PrtN; ++i)
{
        CMcParticle::Tpart* pPrt = &m_PrtD[i];
        D3DXVECTOR3     vcP     = pPrt->m_CrnP;
        …
        FLOAT           fW = pPrt->m_PrsW;
        FLOAT           fH = pPrt->m_PrsH;
        FLOAT           fD = min(fW, fH);
        CMcParticle::VtxDUV1* pVtx = &m_pVtx[i*6 + 0];

        (pVtx+0)->p.x   = vcP.x - (vcCamX.x - vcCamY.x) * fW;
        (pVtx+0)->p.y   = vcP.y - (vcCamX.y - vcCamY.y) * fH;
        (pVtx+0)->p.z   = vcP.z - (vcCamX.z - vcCamY.z) * fD;
        …

        (pVtx+1)->p.x   = vcP.x + (vcCamX.x + vcCamY.x) * fW;
        (pVtx+1)->p.y   = vcP.y + (vcCamX.y + vcCamY.y) * fH;
        (pVtx+1)->p.z   = vcP.z + (vcCamX.z + vcCamY.z) * fD;
        …

        (pVtx+2)->p.x   = vcP.x - (vcCamX.x + vcCamY.x) * fW;
        (pVtx+2)->p.y   = vcP.y - (vcCamX.y + vcCamY.y) * fH;
        (pVtx+2)->p.z   = vcP.z - (vcCamX.z + vcCamY.z) * fD;
        …
        (pVtx+3)->p.x   = vcP.x + (vcCamX.x - vcCamY.x) * fW;
        (pVtx+3)->p.y   = vcP.y + (vcCamX.y - vcCamY.y) * fH;
        (pVtx+3)->p.z   = vcP.z + (vcCamX.z - vcCamY.z) * fD;
…
```

이외에 파티클을 초기화 하는 부분에서 빌보드를 위한 폭과 높이를 Random 하게 설정하고, 정점의 색상을 파티클의 색상과 연결시켜야 하는데 이러한 내용은 다음 예제에 포함되어 있습니다.



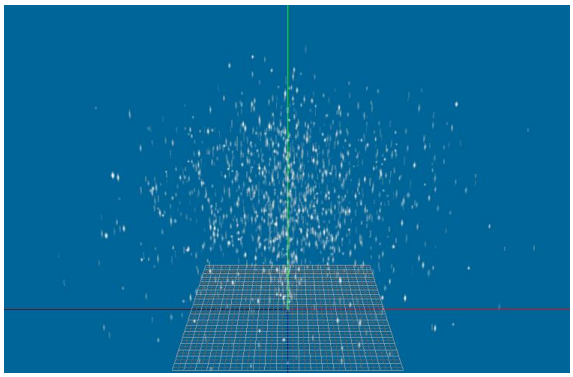
prt32_billboard.zip

3.3 정렬(Sorting)

파티클에 알파블렌딩이 있다면 파티클을 카메라의 z축으로 정렬을 해야 렌더링이 제대로 됩니다. C의 qsort 함수를 이용해서 파티클을 정렬해 봅시다. 정렬을 위해서 카메라의 z축과 파티클의 위치를 내적한 값을 저장할 수 있도록 파티클의 구조체를 수정해야 합니다. 내적을 구하고 정렬하는 부분은 다음과 같습니다.

```
// 카메라의 정보
D3DXMATRIX mtView;
m_pDev->GetTransform(D3DTS_VIEW, &mtView);

D3DXVECTOR3 vcCamX(mtView._11, mtView._21, mtView._31);
D3DXVECTOR3 vcCamY(mtView._12, mtView._22, mtView._32);
D3DXVECTOR3 vcCamZ(mtView._13, mtView._23, mtView._33);

for(i=0; i<m_PrtN; ++i)
{
        CMcParticle::Tpart* pPrt = &m_PrtD[i];
        D3DXVECTOR3     vcP     = pPrt->m_CrnP;

        // 카메라의 Z축과 파티클의 위치와 내적
        pPrt->m_PrsZ    = D3DXVec3Dot(&vcP, &vcCamZ);
}

// Sorting
qsort (m_PrtD
        , m_PrtN
```

```
        , sizeof(CMcParticle::Tpart)
        , (int(*) (const void *, const void *)) CMcParticle::SortFnc);
...
```



prt33_sort.zip

3.4 생명

파티클의 표현 요소가 어느 정도 완성 되었다면 파티클에 생명 시간을 결정할 차례입니다. 파티클에 생명시간을 주는 것은 파티클을 많이 사용하면 할수록 효과는 좋아지지만 컴퓨터의 자원은 제한되어 있습니다. 영역을 설정하는 것 이외에 파티클에 생명 시간을 주어 일정 시간이 지나면 다시 재생할 수 있도록 해야 합니다.

파티클의 전체 생명을 색상으로 설정할 수 있고, 타임으로 설정할 수 있는데 생명시간이 남아 있어도 알파 값이 0이면 다시 재생하는 혼합된 방법을 사용하기 때문에 파티클 구조체를 수정해서 생명을 설정할 수 있도록 합니다.

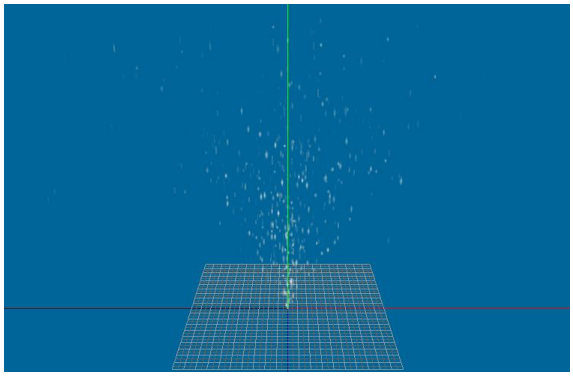
```
struct Tpart
{
        ...
        // 입자의 생명 요소
        BOOL            m_bLive;        // Active (Yes/No)
        FLOAT           m_fLife;        // Particle fLife
        FLOAT           m_fFade;        // Fade Speed
        DWORD           m_dColor;       // Color
        ...
};
```

파티클에 생명시간이 있다면 파티클을 갱신하는 순서는 다음과 같습니다.

1. 운동을 갱신한다. -> 영역을 벗어나면 죽은 상태로 설정한다.
2. 파티클 생명을 갱신한다. -> 시간이 만료되면 죽은 상태로 설정한다.
3. 죽은 파티클을 재생한다.

```
INT CMcParticle::FrameMove()
...
// 1. 운동을 갱신한다.
FLOAT   ftime = m_fTimeAvg * 0.1f;
for(i=0; i<m_PrtN; ++i)
{
        ...
        // 경계값 설정. 벗어나면 죽은 상태로 설정.
        if(pPrt->m_CrnP.y<0.f)
        {
                pPrt->m_bLive   = FALSE;
        }
}

// 2. 파티클의 생명을 갱신한다.
for(i=0; i<m_PrtN; ++i)
{
        ...
        pPrt->m_fLife -=pPrt->m_fFade*ftime;

        if(pPrt->m_fLife<=0.f)
        {
                pPrt->m_bLive   = FALSE;
                continue;
        }
        ...
}

// 3. 죽은 파티클을 재생한다.
for(i=0; i<m_PrtN; ++i)
{
        ...
        if(TRUE == pPrt->m_bLive)
```

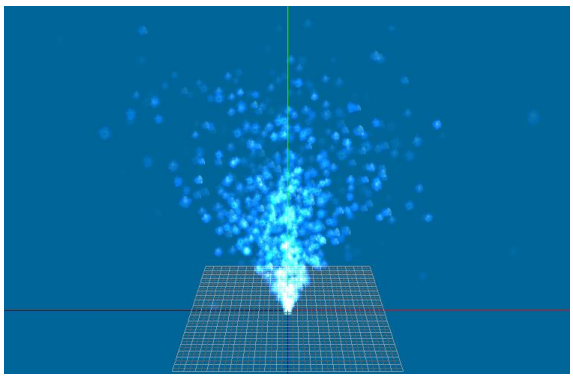
```
                continue;
        this->SetPart(i);
}
...
```



prt34_life.zip

3.5. 파티클 + 텍스처 애니메이션

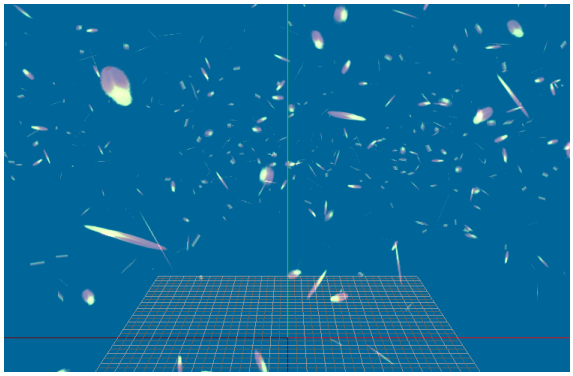
한 가지 파티클을 이용하는 것보다 여러 종류의 파티클을 이용하는 것이 좋습니다. 이것을 굳이 이름을 붙인다면 복합 효과라 할 수 있는데 복합 효과의 구성은 파티클과 파티클, 파티클과 각각의 파티클에 텍스처 애니메이션을 적용, 파티클과 운동이 없는 2D 텍스처 애니메이션 결합 등이 있습니다. 또한 파티클에 텍스처를 설정할 경우 여러 종류의 파티클 텍스처를 하나로 모아 놓은 파티클 맵을 이용하면 좀 더 다양한 연출을 할 수 있습니다.



<파티클 + 텍스처 애니메이션: prt35_tex_ani.zip>

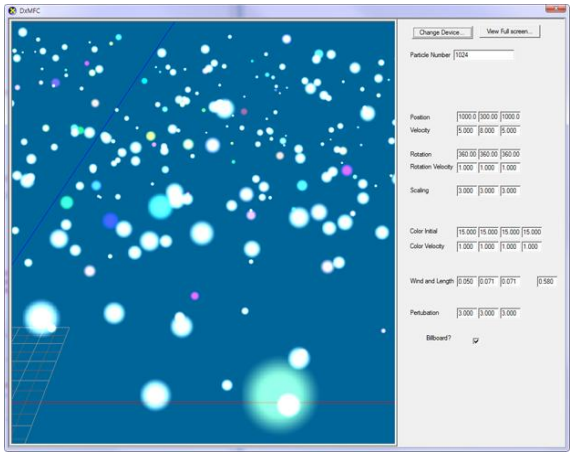
3.6 기타 예제

운동을 복잡하게 구성하면 좀 더 현실감 있는 파티클을 구현할 수 있습니다.



<prt36_petal.zip>

이를 위해서 파티클 툴(Particle tool)이 필요합니다. 파티클 툴은 보통 파티클의 초기 값, 운동 방정식, 경계 값으로 구분해서 제작합니다. 따라서 파티클의 자료 구조는 가장 중요한 내용이라 할 수 있습니다. 또한 경계 값의 설정으로 파티클은 순전히 연출이므로 비슷한 출력이라면 좀 더 적은 수로 표현 될 수 있도록 충돌 및 생명에 대한 경계 값 설정 방법이 매우 중요하다고 할 수 있습니다.



<간단한 파티클 툴(MFC): prt37_tool.zip>