3D Game Programming 31
- Particle Effect

afewhee@gmail.com

- Particle 기초
- Coordinate System (좌표 시스템)
- Particle 심화

- Newton 방정식: 위치, 속도, 가속도, 힘의 정의
  - 위치 $p(x,y,z)=\vec{p}$
  - 속도 $\vec{v}=\lim_{t\to t_0}\frac{\vec{p}(t)-\vec{p}(t_0)}{t-t_0}=\frac{\Delta\vec{x}}{\Delta t}(\Delta t\to 0)=\frac{d\vec{x}}{dt}$
  - 가속도 $\vec{a}=\lim_{t\to t_0}\frac{\vec{v}(t)-\vec{v}(t_0)}{t-t_0}=\frac{\Delta\vec{v}}{\Delta t}(\Delta t\to 0)=\frac{d\vec{v}}{dt}$
  - 힘 $\vec{F}=m*\vec{a}$
- 실 세계
  - 속도 $\vec{v}\approx\frac{\Delta\vec{x}}{\Delta t}$ 또는 $\vec{v}\sim\frac{\Delta\vec{x}}{\Delta t}$
  - 가속도 $\vec{a}\approx\frac{\Delta\vec{v}}{\Delta t}$ 또는 $\vec{a}\sim\frac{\Delta\vec{v}}{\Delta t}$

≈ 매우 비슷

'~' : 비슷

- Newton 방정식
  - 속도 구하기
    - 이상적인 방정식

$$d\vec{v} = \vec{a} * dt$$

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

실 세계

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

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

  - 위치 구하기
    - 이상적인 방정식

$$d\vec{p} = \vec{v} * dt$$

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

실 세계

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

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

● 시간에 대한 가속도의 변화가 일정한 Newton 방정식 풀이

$$d\vec{v} = \vec{a} * dt$$

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

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

$$d\vec{p} = \vec{v} * dt$$

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

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

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

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

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

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

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

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

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

- 시간에 대한 가속도의 변화가 일정한 Newton 방정식 풀이

뉴턴 방정식에 대한 최종 위치 프로그램 순서

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

pseudo-code

```
// 가속도 갱신
update a(x,y,z);

// 속도 갱신
v(x,y,z) += a(x,y,z) * t;

// 위치 갱신
p(x,y,z) += v(x,y,z) * t;
```

a(x,y,z) = Σa_i ➔ 가속도의 합
= 공기저항, 마찰력, 횡력, 항력, 등
운동에 영향을 주는 모든 가속도를
더한 값

시간 t = 장면의 평균 프레임(Frame) 시간

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

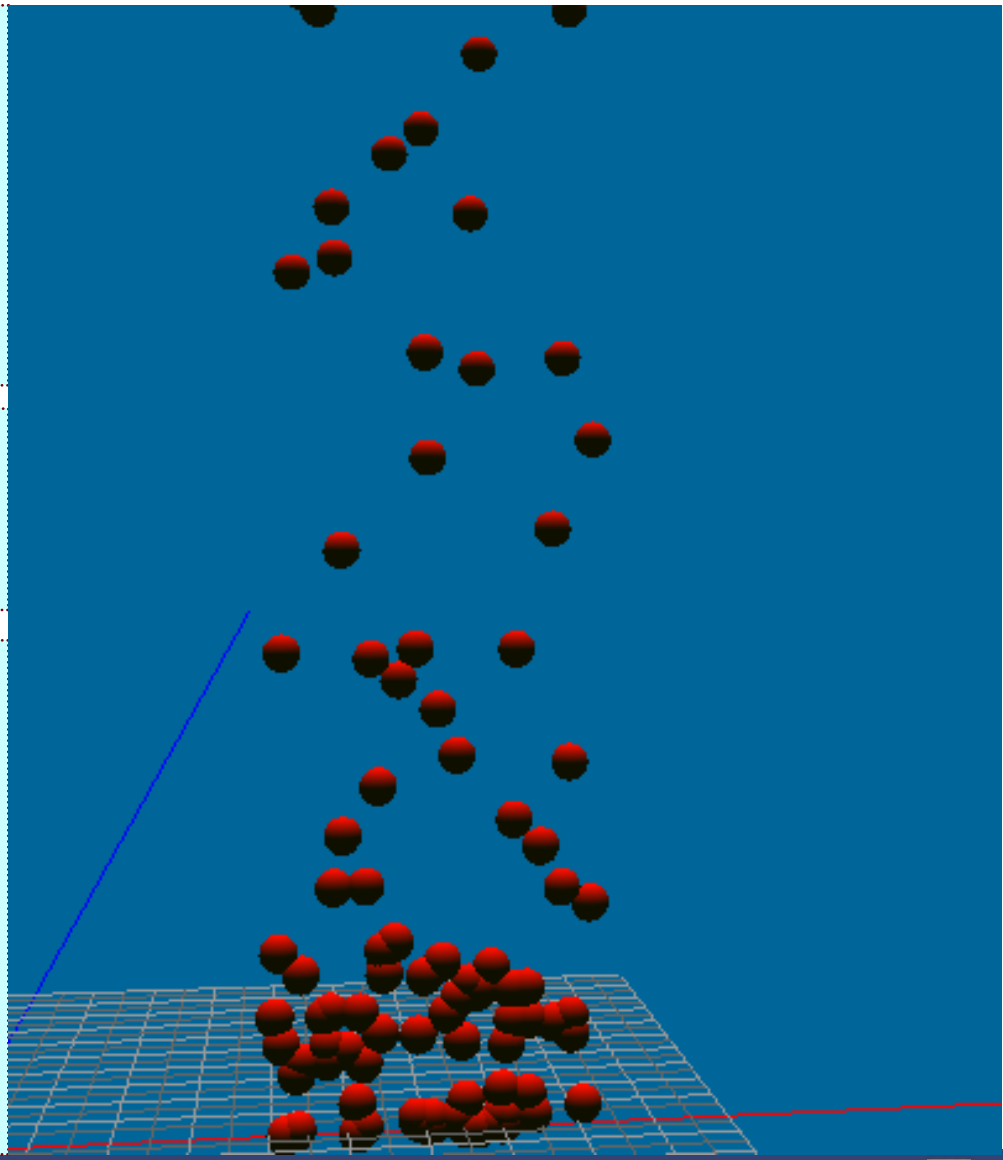
$$\Delta\vec{p} = \vec{v} * \Delta t$$

Newton 방정식 프로그램 방법

```
// 운동을 결정하는 변수 설정
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
```

```
// 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
```

```
// 매 프레임 마다 가속도, 속도, 위치 등을 순서대로 갱신
FLOAT    ftime = m_fTimeAvg * 0.1f;

// 1. 가속도 갱신
m_CrnA   = m_CrnA;

// 2. 현재 속도 갱신
m_CrnV +=m_CrnA * ftime;

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

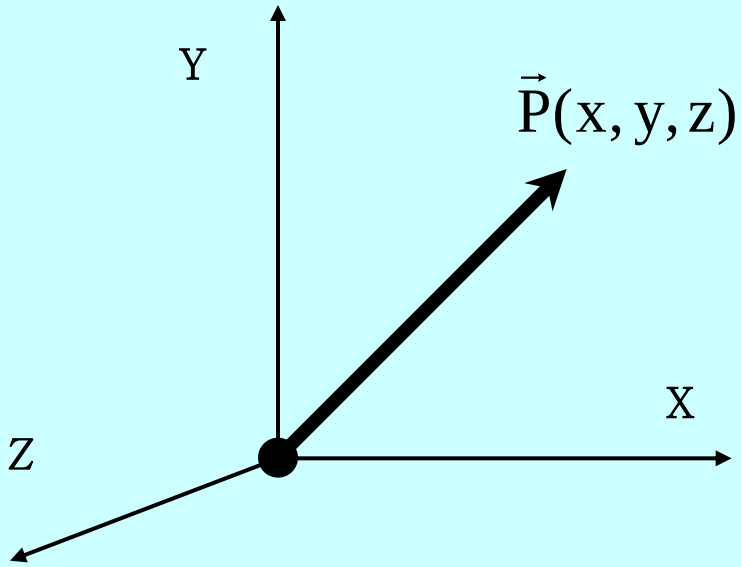


- Coordinate System (좌표계) 종류
  - 데카르트 좌표계
  - 원통 좌표계
  - 극 좌표계, 구 좌표계

●데카르트 좌표계 (Cartesian Coordinate)

- ◆흔히 우리가 알고 있는 x, y, z로 표현되는 좌표계
- ◆가장 일반적으로 사용하며 대칭성이 없는 경우에 많이 사용

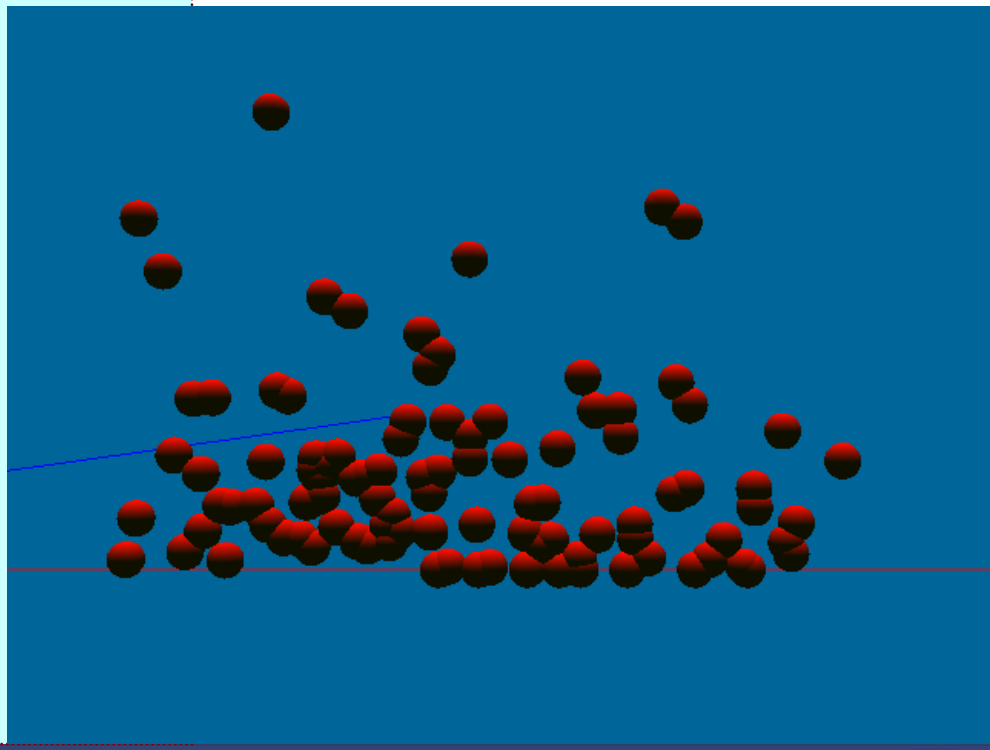
<Cartesian Coordinate>




```
//데카르트 좌표계를 이용한 파티클 설정 예

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


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


...
```

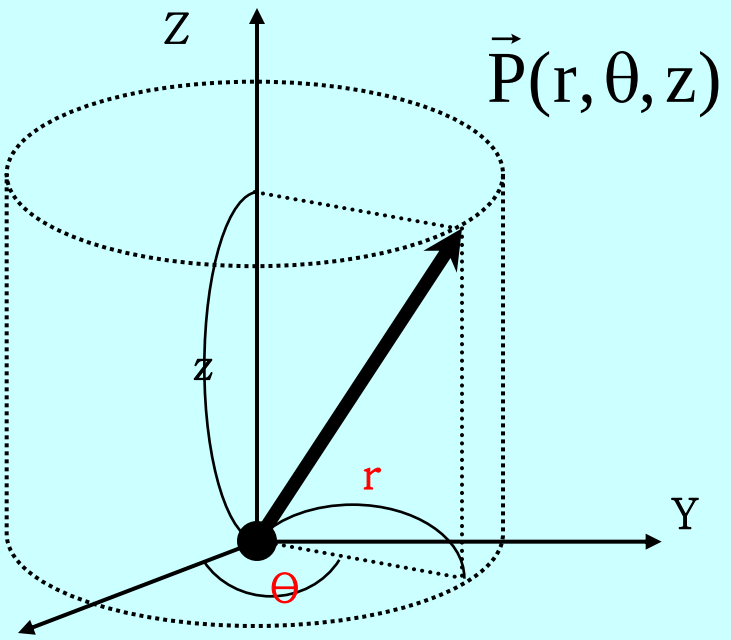
●데카르트 좌표계 예 - 파티클

```
INT CMcParticle::FrameMove()
...
   FLOAT   ftime = m_fTimeAvg * 0.1f;
   for(int i=0; i<m_PrtN; ++i)
   {
      CMcParticle::Tpart*  pPrt = &m_PrtD[i];

      // 공기저항을 구한다.
      D3DXVECTOR3  vcAirR = pPrt->m_CrnV;                  // 공기저항의 방향 벡터
      FLOAT       fLenV  = D3DXVec3LengthSq(&vcAirR);  // 속도의 제곱 크기 구함
      // 공기저항의 방향 벡터를 구한다.
      D3DXVec3Normalize(&vcAirR,  &vcAirR);
      // 이동 속도와 반대로 설정
      vcAirR  *= -1.F;
      // 속력제곱 * 공기 저항 계수를 곱함.
      vcAirR  *= fLenV * pPrt->m_fDamp;
      // 1. 가속도에 공기저항을 더한다.
      pPrt->m_CrnA = pPrt->m_IntA + vcAirR;
      // 2. 현재 속도 갱신
      pPrt->m_CrnV += pPrt->m_CrnA * ftime;
      // 3. 현재 위치 갱신
      pPrt->m_CrnP += pPrt->m_CrnV * ftime;
      // 4. 경계값 설정
      if(pPrt->m_CrnP.y<0.f)
      {
         pPrt->m_CrnV.y *= -1.f;            // 진행의 방향을 바꾸고
         pPrt->m_CrnV.y *= pPrt->m_fElst;    // 탄성 계수를 곱한다.

         pPrt->m_CrnP.y  = 0.f;             // y의 위치를 다시 정한다.
      }
   }
...
```



- 원통 좌표계(Cylinder coordinate)
  - 데카르트 좌표계의 2개의 좌표축을 r과 θ로 표현
  - 직선에 대칭인 경우 사용
  - z=0 이면 극 좌표계
  - EX) Healing

<Cylinder Coordinate>



점 p(r, θ, z)에 대한 데카르트 좌표계 표현

$$x = r * \cos\theta$$
$$y = r * \sin\theta$$
$$z = z$$

(x, y, z)에 대한 원통 좌표계 표현

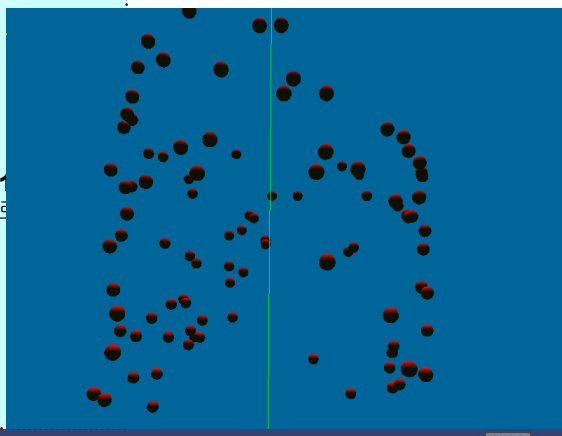
$$r = \sqrt{x^2 + y^2},\ \theta = \tan^{-1}\left(\frac{y}{x}\right),\ z = z$$

r:[0, ∞], θ:[0, 2π], z:[-∞, +∞]

● 원통 좌표계 예

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


      // 공기저항을 구한다.
      D3DXVECTOR3 vcAirR = pPrt->m_CrnV;                  // 공기저
      FLOAT       fLenV  = D3DXVec3LengthSq(&vcAirR);     // 속

      // 공기저항의 방향 벡터를 구한다.
      D3DXVec3Normalize(&vcAirR,  &vcAirR);

      // 이동 속도와 반대로 설정
      vcAirR  *= -1.F;

      // 속력제곱 * 공기 저항 계수를 곱함.
      vcAirR  *= fLenV * pPrt->m_fDamp;

      // 1. 가속도에 공기저항을 더한다.
      pPrt->m_CrnA = pPrt->m_IntA + vcAirR;

      // 2. 현재 속도 갱신
      pPrt->m_CrnV += pPrt->m_CrnA  * ftime;

      // 3. 현재 위치 갱신
      pPrt->m_CrnP += pPrt->m_CrnV * ftime;

      // 4. 경계값 설정
      if(pPrt->m_CrnP.y<0.f)
      {
         pPrt->m_CrnV.y  *= -1.f;          // 진행의 방향을 바꾸고
         pPrt->m_CrnV.y  *= pPrt->m_fElst;    // 탄성 계수를 곱한다.

         pPrt->m_CrnP.y = 0.f;             // y의 위치를 다시 정한다.
      }
   }
```

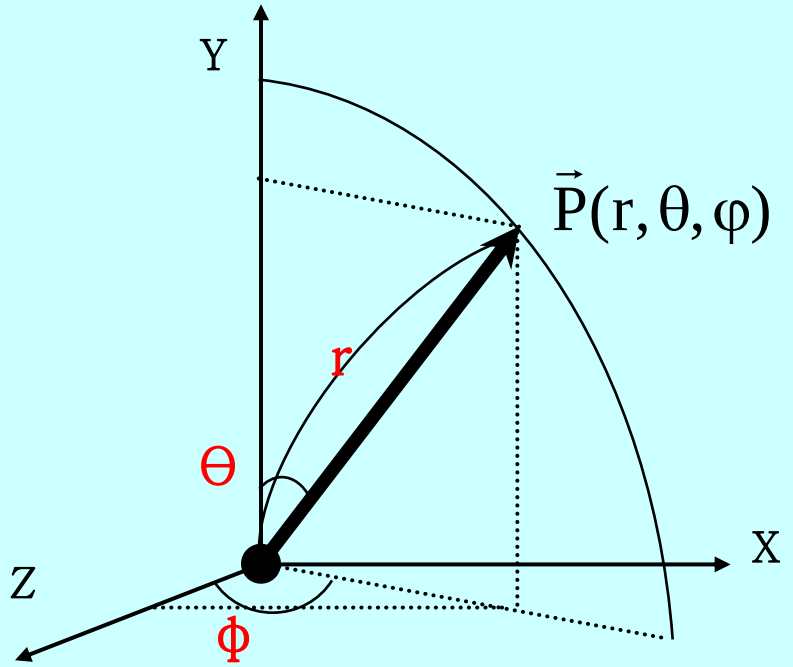
```
void CMcParticle::SetAni(BOOL  bAni)
{
   FLOAT   fAngle;
   FLOAT   fSpdR;      // x, z 방향 속력
   FLOAT   fSpdY;      // y 방향 속력

   for(int i=0; i<m_PrtN; ++i)
   {
      CMcParticle::Tpart*  pPrt = &m_PrtD[i];
      // 초기 가속도
      pPrt->m_IntA = D3DXVECTOR3(0, 0.F,0);
      //초기 속도와 위치를 설정하기 위한 변수
      fAngle = float(rand()%360);
      fSpdR = 100.f + rand()%101;
      fSpdR *=0.001f;

      fSpdY = 20.f + rand()%101;
      fSpdY *=0.02f;
      // 라디안으로 변경
      fAngle = D3DXToRadian(fAngle);
      // 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);
      // 탄성 계수 설정
      pPrt->m_fElst= (50 + rand()%51)*0.01f;
      // 공기저항 계수
      pPrt->m_fDamp= (100 + rand()%101)*0.0000
      // 초기 위치, 속도, 가속도를 현재의 값들의 초기 값으
      pPrt->m_CrnP = pPrt->m_IntP;
      pPrt->m_CrnV = pPrt->m_IntV;
      pPrt->m_CrnA = pPrt->m_IntA;
   }
}
```



●구 좌표계(Spherical coordinate)

- ◆점 p의 좌표를 p(r, θ, φ) 표현한 좌표계
- ◆점(Point)에 대칭이 있는 경우에 유리
- ◆φ가 없으면 극 좌표계(Polar Coordinate)와 동일

< Spherical coordinate >



점 p(r, θ, φ)에 대한 데카르트 좌표계 표현

$$z = r * \sin\theta * \cos\varphi$$
$$x = r * \sin\theta * \sin\varphi$$
$$y = r * \cos\theta$$

(x, y, z)에 대한 구 좌표계 표현

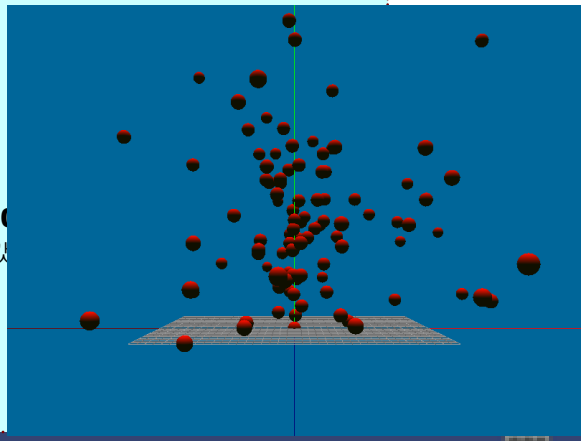
$$r = \sqrt{x^2 + y^2 + z^2},\ \theta = \tan^{-1}\left(\frac{\sqrt{z^2 + x^2}}{y}\right),\ \varphi = \tan^{-1}\frac{x}{z}$$

r:[0, ∞], θ:[0, π], φ:[0, 2π]

● 구 좌표계 예

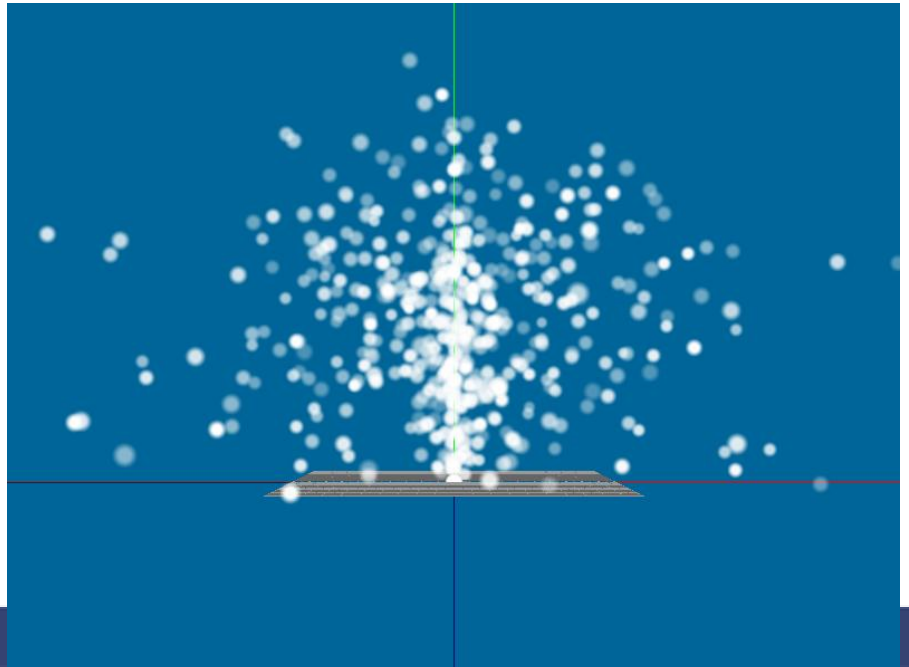
```
INT CMcParticle::FrameMove()
...
   for(int i=0; i<m_PrtN; ++i)
   {
      CMcParticle::Tpart*  pPrt = &m_PrtD[i];

      // 공기저항을 구한다.
      D3DXVECTOR3  vcAirR = pPrt->m_CrnV;                //
      FLOAT       fLenV  = D3DXVec3LengthSq(&vcAirR);

      // 공기저항의 방향 벡터를 구한다.
      D3DXVec3Normalize(&vcAirR,  &vcAirR);

      // 이동 속도와 반대로 설정
      vcAirR  *= -1.F;

      // 속력제곱 * 공기 저항 계수를 곱함.
      vcAirR  *= fLenV * pPrt->m_fDamp;

      // 1. 가속도에 공기저항을 더한다.
      pPrt->m_CrnA = pPrt->m_IntA + vcAirR;

      // 2. 현재 속도 갱신
      pPrt->m_CrnV += pPrt->m_CrnA * ftime;

      // 3. 현재 위치 갱신
      pPrt->m_CrnP += pPrt->m_CrnV * ftime;

      // 4. 경계값 설정
      if(pPrt->m_CrnP.y<0.f)
      {
         SetPart(i);
      }
   }
...
```

```
void CMcParticle::SetPart(int  nIdx)
{
   CMcParticle::Tpart*  pPrt = &m_PrtD[nIdx];
   FLOAT   fTheta;     // 각도 θ
   FLOAT   fPhi;       // 각도 φ
   FLOAT   fSpdR;      // 속도 크기
   // 초기 가속도
   pPrt->m_IntA = D3DXVECTOR3(0,  -0.2F,0);
   //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);
   // 초기 위치
   pPrt->m_IntP.x = 0.f;
   pPrt->m_IntP.y = 0.f;
   pPrt->m_IntP.z = 0.f;
   // 탄성 계수 설정
   pPrt->m_fElst=  (50 + rand()%51)*0.01f;
   // 공기저항 계수
   pPrt->m_fDamp= (100 + rand()%101)*0.00
   // 초기 위치, 속도, 가속도를 현재의 값들의 초기 값
   pPrt->m_CrnP = pPrt->m_IntP;
   pPrt->m_CrnV = pPrt->m_IntV;
   pPrt->m_CrnA = pPrt->m_IntA;
}
```



- Point Sprite
  - 한 정점에 하나의 텍스처 적용
  - 사각형 렌더링에 필요한 정점의 사용량을 1/4 정도로 줄임
  - 한 파티클과 한 정점이 1:1로 대응 → 프로그램 구현이 용이

  - 버텍스 버퍼를 통해서만 렌더링 가능



- Point Sprite 프로그램 방법
  - 1. UV가 없는 정점 구조체 선언

```
struct VtxD
{
    D3DXVECTOR3 p;
    DWORD       d;

    VtxD() : p(0,0,0), d(0xFFFFFFFF){}
    VtxD(FLOAT X,FLOAT Y,FLOAT Z,DWORD D=0xFFFFFFFF): p(X,Y,Z), d(D){}
    enum { FVF =(D3DFVF_XYZ|D3DFVF_DIFFUSE) };
};
```

  - 2. 버텍스 버퍼 생성m_pDev->CreateVertexBuffer(...,D3DUSAGE_POINTS,...);

  - 3.렌더링 옵션 설정

```
// Point Sprite 활성화
m_pDev->SetRenderState(D3DRS_POINTSPRITEENABLE, TRUE);
m_pDev->SetRenderState(D3DRS_POINTSCALEENABLE, TRUE);
// Point Size 조정
m_pDev->SetRenderState(D3DRS_POINTSIZE, FtoDW(50.f));
m_pDev->SetRenderState(D3DRS_POINTSIZE_MIN, FtoDW(1.0f));
// Point Scale 조정
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));
```

Billboard 효과

Transform 원리를 이용한 방법

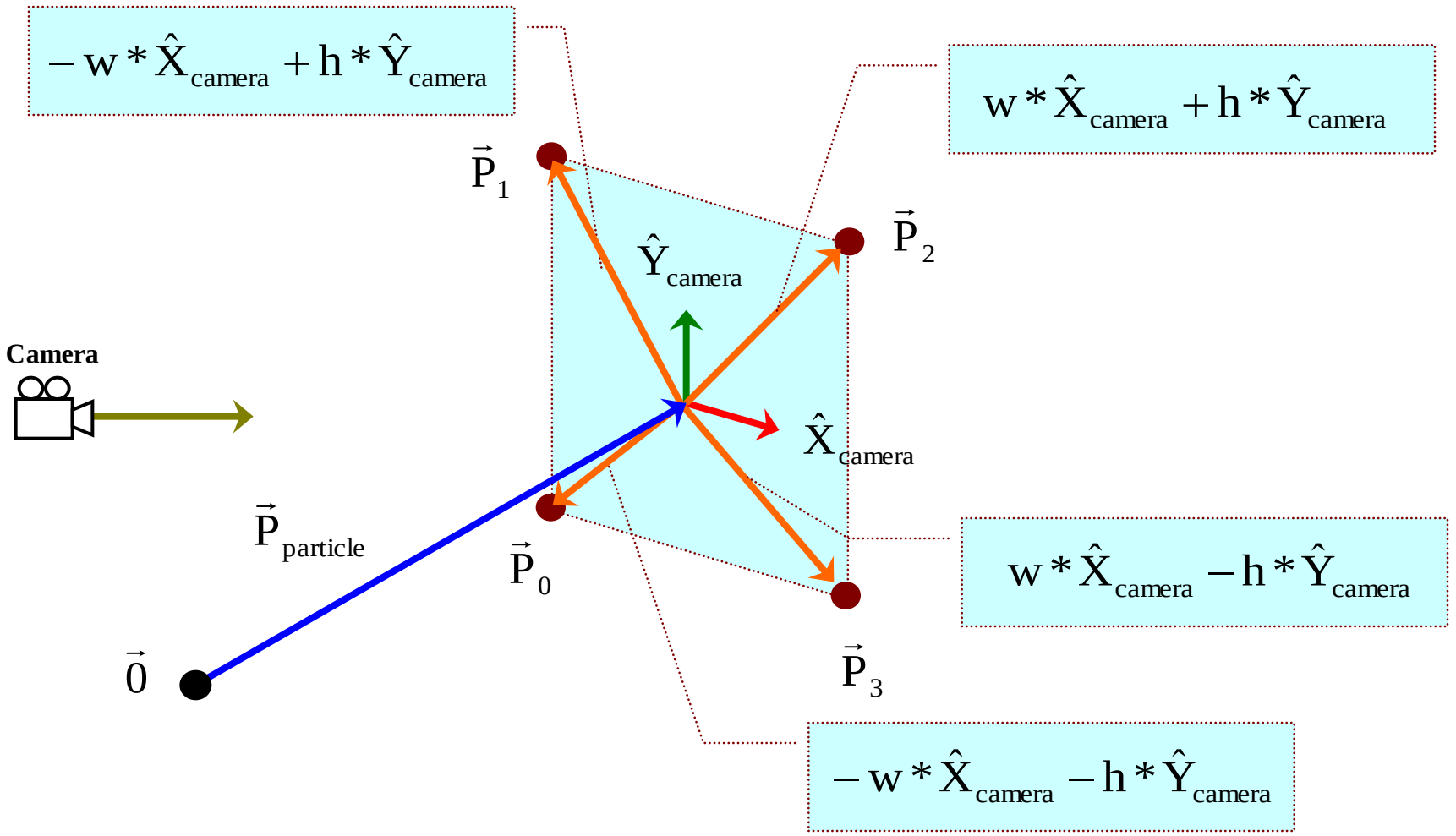
- Direct3D 렌더링 머신의 뷰잉 변환 행렬의 초기 값은 단위 행렬 → 카메라의 위치는 0, 시선 방향은 z축
- 정점은 렌더링 파이프라인의 뷰잉 변환을 거치므로 뷰 행렬의 역행렬을 가지고 빌보드 행렬을 만들어 이를 정점에 미리 적용하면 뷰잉 변환을 통과 한 것처럼 처리가 됨 → 사각형의 정점을 구성할 때 x, y만 구성하고 z=0으로 하면 항상 카메라의 z축에 수직인 평면을 만들 수 있음

카메라의 축을 이용한 방법

- 카메라의 x축, y축에 평행한 사각형을 그리면 항상 카메라에 수직인 평면을 그림
- 직관적으로 이해가 쉽다.
- Particle을 회전시키거나 운동하는 데 편리

- Billboard 효과
  - Graphic 파이프라인에서의 정정 변환
    - 월드 변환 → 뷰 변환 → 투영 변환
      → 정규 변환 좌표 = 모델 좌표(x,y,z) * 월드 행렬 * 뷰 행렬 * 투영 행렬
  - 만약 월드 변환의 행렬이 뷰 변환 행렬의 역 행렬이라면 정점은 투영 변환만 적용
    - 정규 변환 좌표 = 모델 좌표(x,y,z) * 뷰 행렬$^{-1}$ * 뷰 행렬 * 투영 행렬
      = 모델 좌표(x,y,z) * (뷰 행렬$^{-1}$ * 뷰 행렬) * 투영 행렬
      = 모델 좌표(x,y,z) * (단위 행렬) * 투영 행렬
      = 모델 좌표(x,y,z) * 투영 행렬
  - 렌더링 머신의 뷰잉 변환 행렬 초기 값은 단위 행렬 이고 이것은 카메라의 위치가 (0,0,0) 시선 방향은 z축이다. 이에 맞추어 정점의 구성을 z=0으로 하고 x와 y를 구성하면 항상 카메라의 z축과 수직인 평면을 구성할 수 있음
  - 빌보드 행렬: 빌보드 효과를 위한 행렬 뷰 행렬의 역 행렬에 41=0, 42=0, 43=0 적용해서 구함
    → 이동이 없으므로 카메라에 대한 반대 방향으로의 회전 행렬
  - 구현 방법 → 카메라 빌보드 참조

Billboard 효과 - 카메라 축을 이용한 빌보드



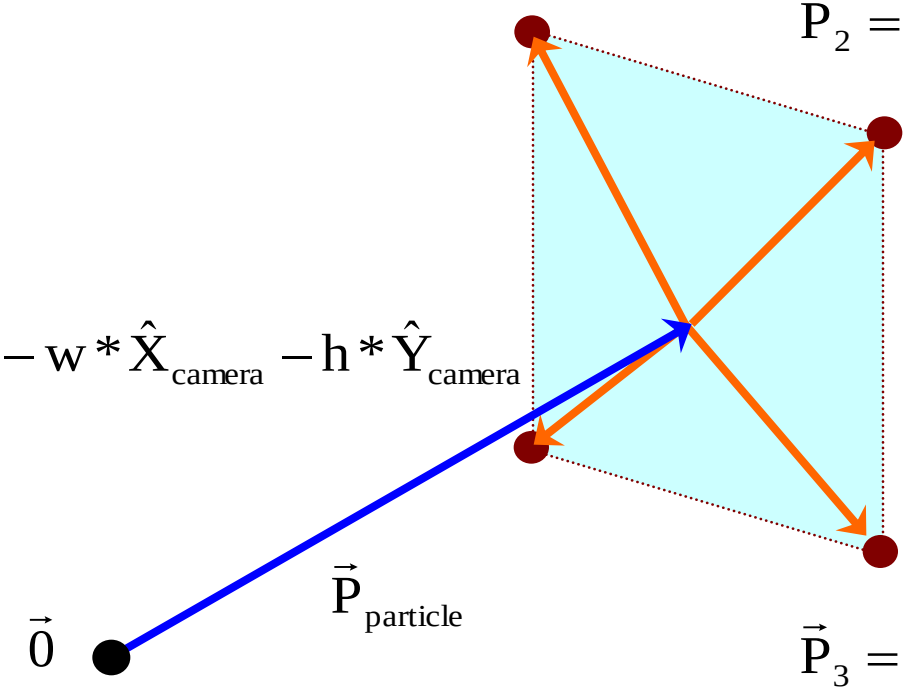
Billboard 효과 - 카메라 축을 이용한 빌보드

$$\vec{P}_1 = \vec{P}_{particle} - w * \hat{X}_{camera} + h * \hat{Y}_{camera}$$

$$\vec{P}_2 = \vec{P}_{particle} + w * \hat{X}_{camera} + h * \hat{Y}_{camera}$$

$$\vec{P}_0 = \vec{P}_{particle} - w * \hat{X}_{camera} - h * \hat{Y}_{camera}$$

$$\vec{P}_3 = \vec{P}_{particle} + w * \hat{X}_{camera} - h * \hat{Y}_{camera}$$



- Billboard 효과 - 카메라 축을 이용한 빌보드 구현

```
D3DXVECTOR3 vcP = pPrt->m_CrnP;
…
FLOAT       fW = pPrt->m_PrsW;
FLOAT       fH = pPrt->m_PrsH;
FLOAT       fD = min(fW, fH);

(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+0)->u     = 0;(pVtx+0)->v     = 0;

(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+1)->u     = 1;(pVtx+1)->v     = 0;

(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+2)->u     = 0;(pVtx+2)->v     = 1;

(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;
(pVtx+3)->u     = 1;(pVtx+3)->v     = 1;
```

- 파티클 정렬
  - 파티클을 대부분 반투명 색상을 가지고 있으므로 알파 블렌딩을 위해서 정렬을 통해서 카메라의 시선 벡터 방향으로 멀리 있는 파티클 순으로 렌더링
  - 거리 결정은 카메라의 Z 축 벡터와 내적으로 결정

Ex)

```
// 뷰 행렬
D3DXMATRIX mtView;
m_pDev->GetTransform(D3DTS_VIEW, &mtView);
// 카메라 Z축
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);
```

- 파티클 생명
  - 자원의 효율을 높이기 위해 생명 시간을 주어 일정 시간이 지나면 다시 재생

- 방법
  - 시간으로 설정
  - 색상의 알파 값으로 설정 ➔ 알파가 0이면 다시 재생
  - 시간, 색상의 혼합으로 주로 사용

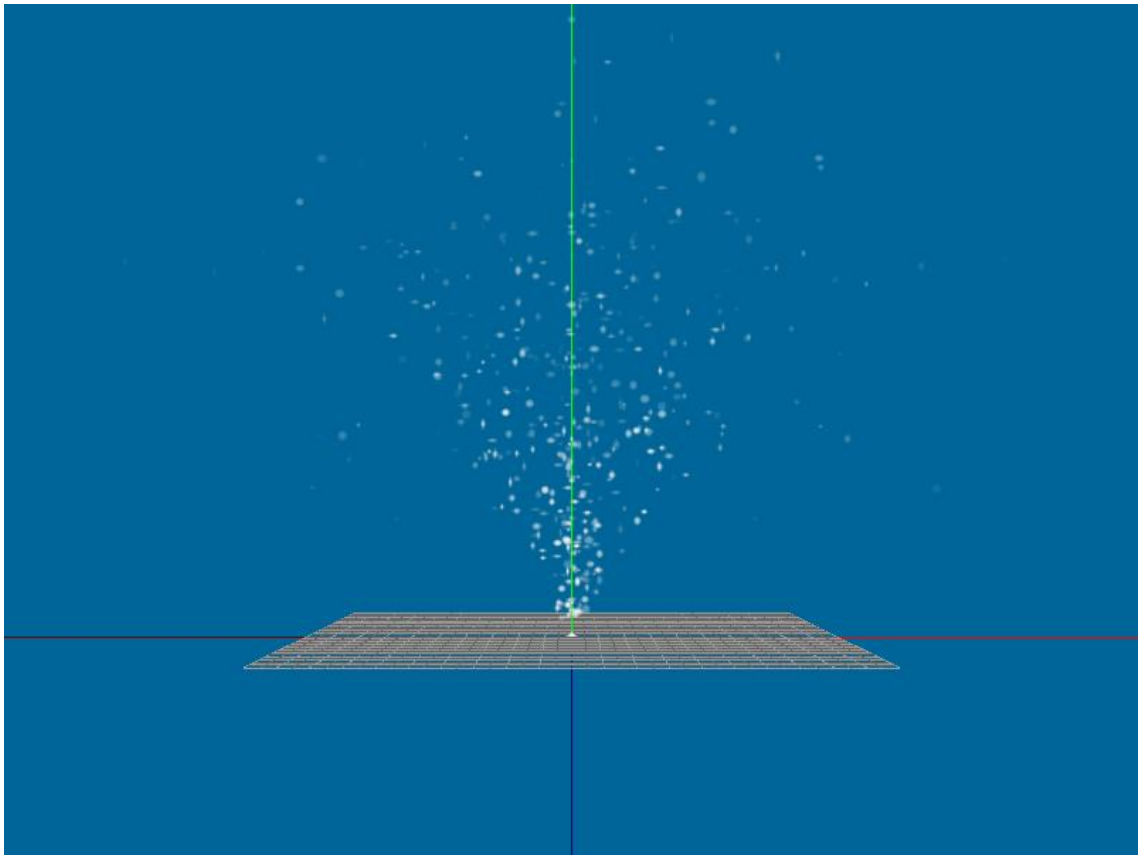
- 생명 시간이 들어간 파티클 구조체

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

- 파티클 갱신 순서
  - 1. 운동을 갱신 ➔ 영역을 벗어나면 죽은 상태로 설정
  - 2. 파티클 생명을 갱신 ➔ 시간이 만료되면 죽은 상태로 설정
  - 3. 생명이 다한 파티클을 재생

- 프로그램 구현

```
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);
}
```



- 파티클 운동의 복합 효과와 파티클 툴

```
// 공기저항을 구한다.
D3DXVECTOR3     vcAirR = pPrt->CrnV;                    // 공기저항
        의 방향 벡터
FLOAT           fLenV  = D3DXVec3LengthSq(&vcAirR);     // 속도의 제
        곱 (Vx*Vx + Vy*Vy + Vz*Vz)크기 구함

// 공기저항의 방향 벡터를 구한다.
D3DXVec3Normalize(&vcAirR, &vcAirR);

// 이동 속도와 반대로 설정
vcAirR *= -1.F;

// 속력제곱 * 공기 저항 계수를 곱함.
vcAirR *= fLenV * pPrt->fDamp;

// 바람에 의한 perturbation을 더한다.
vcWind.x = m_vcWind.x * (10 + rand()%11)/20.f * (1+sinf(D3DXToRadian(pPrt->CrnR.x)));
vcWind.y = m_vcWind.y * (10 + rand()%11)/20.f * (1+sinf(D3DXToRadian(pPrt->CrnR.y)));
vcWind.z = m_vcWind.z * (10 + rand()%11)/20.f * (1+sinf(D3DXToRadian(pPrt->CrnR.z)));

// 회전 요소를 첨가 한다.
vcWind.x += 1.8F* sinf(D3DXToRadian(pPrt->CrnR.x));
vcWind.y += 1.8F* sinf(D3DXToRadian(pPrt->CrnR.y));
vcWind.z += 1.8F* sinf(D3DXToRadian(pPrt->CrnR.z));

vcWind *=.8F;

// 1. 가속도에 공기저항을 더한다.
pPrt->CrnA = pPrt->IntA + vcAirR;

// 2. 현재 속도 갱신
pPrt->CrnV += pPrt->CrnA * ftime;

// 3. 현재 위치 갱신
pPrt->CrnP += pPrt->CrnV * ftime;
pPrt->CrnP += vcWind * ftime;

// 회전
pPrt->CrnR +=pPrt->CrnRv * ftime;

// 4. 죽은 파티클을 재생한다.
if(pPrt->CrnP.y<0.f)
{
    this->SetPart(i);
}
```