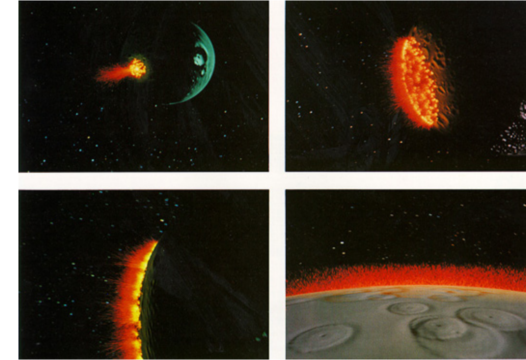
Particle Systems

305890
2009년 봄학기
5/27/2009
박경신

Star Trek II (1983)

- Particle System이란 불꽃, 연기, 물 등의 자연현상을 무수히 많은 작은 입자의 동적인 움직임으로 표현한 것
- Star Trek II (1983) "Genesis Effect"에서 처음 소개됨

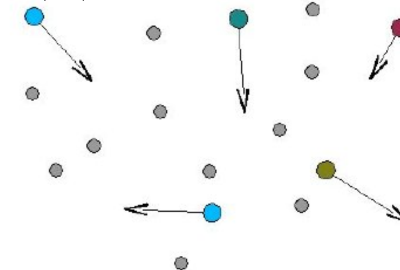


Particle Systems

- Particles and Point Sprites
- Particle System Components
  - 파티클에 부여할 수 있는 속성들을 배우고 Direct3D에서 파티클을 표현하는 방법을 배운다
- Particle Systems 예제
  - 눈 (Snow)
  - 불꽃놀이 (Firework)
  - 입자총 (Particle Gun)

Particles and Point Sprites

- Particle
  - 입자(particle)는 위치 (position), 속도 (velocity), 색 (color)를 가진 간단한 점(point)으로 표현
  - 개별적 입자들은 시스템에 추가(add)되어 움직(move)이고 중력이나 바람과 같은 물리적인 힘에 의해 영향을 받음 그리고 결국엔 죽게 됨 (life)



Particles and Point Sprites

- Particle의 표현
  - Point sprite를 사용. 포인트 기본형과는 달리 텍스처 적용과 크기 변경 가능
  - 포인트 기본형 (D3DPRIMITIVETYPE의 D3DPT_POINTLIST)이 파티클시스템에 적합하나, 포인트 형은 단일 픽셀로 래스터화되므로 파티클의 크기나 텍스처를 입힌 파티클 구현에 제한적이였음.

```
struct Particle {
    D3DXVECTOR3 _position;
    D3DCOLOR _color;
    static const DWORD FVF;
};
const DWORD Particle::FVF = D3DFVF_XYZ | D3DFVF_DIFFUSE;
```

Particles and Point Sprites

- Particle의 크기 지정
  - 각각의 particle 크기를 직접 조절할 수 있어 다양한 효과 표현 가능
  - Struct에 "float _size"를 추가하고, FVF에 D3DFVF_PSIZE를 추가함
  - 그러나 대부분의 HW가 지원하지 않으므로 권장하지 않음. 지원 여부는 D3DCAPS9의 FVFCaps 멤버의 D3DFVFCAPS_PSIZE bit을 확인하면 알 수 있음.

```
struct Particle {
    D3DXVECTOR3 _position;
    D3DCOLOR _color;
    float _size;
    static const DWORD FVF;
};
const DWORD Particle::FVF = D3DFVF_XYZ | D3DFVF_DIFFUSE |
    D3DFVF_PSIZE;
```

Point Sprite Render State

- Point Sprite와 관련된 Render State
  - D3DRS_POINTSPRITEENABLE
    - Texture coordinate계산을 제어함
    - true: 현재 지정된 전체 texture가 각 point로 매핑되도록 point primitive의 texture coordinate를 지정함
    - false: 전체 point에 대해서 vertex struct 내의 vertex texture coordinate가 사용됨

    ```
    _device->SetRenderState(D3DRS_POINTSPRITEENABLE, true);
    ```
  - D3DRS_POINTSIZE
    - Point sprite의 size를 지정함
    - Size가 camera space에서의 size인지 screen space에서의 size인지는 D3DRS_POINTSCALEENABLE에 의해 결정됨

    ```
    _device->SetRenderState(D3DRS_POINTSIZE, d3d::FtoDw(2.5f));
    DWORD d3d::FtoDw(float f) { return * ((DWORD *) &f); }
    ```

Point Sprite Render State

- Point Sprite와 관련된 Render State
  - D3DRS_POINTSCALEENABLE
    - Point primitive size의 계산을 제어함
    - true: point size를 camera space에서의 값으로 해석함. 따라서 screen space에서의 point size는 거리에 따라 조절됨. (먼 particle은 작게 나타남)
    - false: point size를 screen space에서의 값으로 해석함
    - default는 false

    ```
    _device->SetRenderState(D3DRS_POINTSCALEENABLE, true);
    ```

Point Sprite Render State

- Point Sprite와 관련된 Render State
  - D3DRS_POINTSIZE_MIN
    - Point sprite의 가능한 최소 size를 지정함
  - D3DRS_POINTSIZE_MAX
    - Point sprite의 가능한 최대 size를 지정함

```
_device->SetRenderState(D3DRS_POINTSIZE_MIN, d3d::FtoDw(0.2f));
_device-> SetRenderState(D3DRS_POINTSIZE_MAX, d3d::FtoDw(5.0f));
```

Point Sprite Render State

- Point Sprite와 관련된 Render State
  - D3DRS_POINT_SCALE_A/B/C
    - 카메라와 point sprite의 거리에 따라 point sprite의 크기가 변하는 방법을 제어함
    - D3DRS_POINTSCALEENABLE이 true일 경우만 사용됨
    - Default는 1.0f/0.0f/0.0f임. 유효한 값의 범위는 모두 >= 0.0f 임
    - Point sprite의 최종 size:

$$FinalSize = ViewportHeightSize * Size * \sqrt{\frac{1}{A + B(D) + C(D^2)}}$$

- Size는 D3DRS_POINTSIZE의 값.
- A, B, C는 D3DRS_POINT_SCALE_A/B/C의 값.
- D는 view space내의 point sprite와 카메라(원점에 위치)와의 거리

```
_device->SetRenderState(D3DRS_POINTSCALE_A, d3d::FtoDw(0.0f));
_device->SetRenderState(D3DRS_POINTSCALE_B, d3d::FtoDw(0.0f));
_device->SetRenderState(D3DRS_POINTSCALE_C, d3d::FtoDw(1.0f));
```

Particle Attribute

- Particle의 여러 속성들을 별도의 구조체로 보관하는 것이 편리
  - 렌더링 시에 속성 구조체로부터 particle 구조체로 필요한 것만 복사

```
struct Attribute {
    Attribute() {
            _lifeTime = 0.0f;
            _age = 0.0f;
            _isAlive = true;
    }
    D3DXVECTOR3 _position;      // world space에서의 particle 위치
    D3DXVECTOR3 _velocity;      // 초당 이동 단위
    D3DXVECTOR3 _acceleration;  // 초당 속도 단위
    float _lifeTime;            // particle이 소멸시까지의 소요 시간
    float _age;                 // particle의 현재 나이
    D3DXCOLOR _color;           // particle의 현재 색
    D3DXCOLOR _colorFade;       // particle 색이 시간에 따라 fade되는 방법
    bool _isAlive;              // 현재 생존 상태
};
```

Particle System Components

- Particle System 요소
  - Particle들의 모임
  - Particle들을 보여주고 관리하는 역할
  - 모든 particle에 영향을 주는 전역 특성을 관리함
- PSystem
  - 전형적인 particle system을 일반화한 base class
  - Particle 갱신과 디스플레이, 소멸, 생성 등을 관장하는 역할을 함

PSystem

```
#include "d3dUtility.h"
#include "camera.h"
#include <list>

namespace psys {
class PSystem { // 전형적인 particle system을 일반화한 base class
    protected:
        IDirect3DDevice9* _device;
        D3DXVECTOR3 _origin;              // particle이 시작되는 곳
        d3d::BoundingBox _boundingBox;   // particle이 이동할 수 있는 부피 제한하는데 사용
                                          // 이 영역을 지나면 바로 소멸됨
        float _emitRate;                  // 새 particle이 추가되는 비율 – numpart/sec
        float _size;                      // 모든 particle들의 size

        IDirect3DTexture9* _tex;
        IDirect3DVertexBuffer9* _vb;
        std::list<Attribute> _particles;  // 현재 모든 particle들의 attribute list
        int _maxParticles;                // 한 순간에서의 max allowed particles

        // 아래의 3개는 particle system을 효율적으로 rendering하는 데 사용
        DWORD _vbSize;                    // vertex buffer가 보관할 수 있는 particle 수
                    // vertex buffer에서의 particle수와 system에서의 particle수가 다를 수 있음
        DWORD _vbOffset;                  // offset in vertex buffer to lock
        DWORD _vbBatchSize;               // number of vertices to lock starting at _vbOffset
        // attribute list로 부터 dead particle들을 모두 제거
        virtual void removeDeadParticles();
```

PSystem

```
    public:
        PSystem();
        virtual ~PSystem();
        // vertex buffer & texture 생성
        virtual bool init(IDirect3DDevice9* device, char* texFilename);
        // 모든 particle 속성을 reset; dead particle의 memory를 free하지 않고 새로 spawn함
        virtual void reset();
        // 한 particle 속성을 reset
        virtual void resetParticle(Attribute* attribute) = 0;
        // 한 particle 추가
        virtual void addParticle();
        // 모든 particle을 갱신
        virtual void update(float timeDelta) = 0;
        // 렌더링에 앞서 지정해야 할 초기 render state를 지정
        virtual void preRender();
        virtual void render();
        // 특정 particle system에 지정했을 수 있는 render state를 복구
        virtual void postRender();
        bool isEmpty();   // 현재 particle system에 particle이 없으면 true
        bool isDead();    // 현재 particle system에 모든 particle이 죽었으면 true
};
```

PSystem

```
#include <cstdlib>
#include "pSystem.h"
using namespace psys;
PSystem::PSystem() {
    _device = 0;     _vb = 0;   _tex = 0;
}
PSystem::~PSystem() {
    d3d::Release<IDirect3DVertexBuffer9*> (_vb);
    d3d::Release<IDirect3DVertexBuffer9*> (_tex);
}
bool PSystem::init(IDirect3Device9* device, char* texFilename) {
    _device = device;            // device pointer
    HRESULT hr = 0;
    // dynamic으로 해서 매 frame마다 vb를 갱신해도 속도 감소가 적도록 함.
    // D3DUSAGE_POINTS로 해서 point sprite를 사용할 것임을 지정함.
    hr = device->CreateVertexBuffer(_vbSize * sizeof(Particle),
            D3DUSAGE_DYNAMIC|D3DUSAGE_POINTS|D3DUSAGE_WRITEONLY,
            Particle::FVF, D3DPOOL_DEFAULT, &vb, 0);
    if (FAILED(hr)) {
        ::MessageBox(0, "CreateVertexBuffer() – FAILED", "Psystem", 0);
        return false;
    }
    hr = D3DXCreateTextureFromFile(device, texFilename, &_tex);
    if (FAILED(hr)) {
        ::MessageBox(0, "D3DXCreateTextureFromFile() – FAILED", "Psystem", 0);
        return false;
    }
    return true;
}
```

PSystem

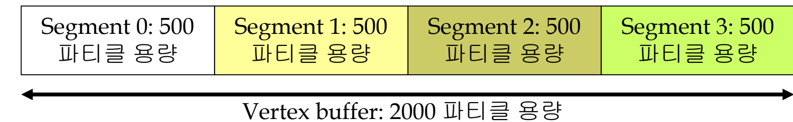
```
void PSystem::reset() {
    std::list<Attribute>::iterator i;
    for(i = _particles.begin(); i != _particles.end(); i++) {
        resetParticle(&(*i));
    }
}
void PSystem::addParticle() {
    Attribute attribute;
    resetParticle(&attribute);
    _particles.push_back(attribute);
}
bool PSystem::isEmpty() { return _particles.empty(); }
bool PSystem::isDead() {
    std::list<Attribute>::iterator i;
    for(i = _particles.begin(); i != _particles.end(); i++) {
        if (i->isAlive) return false;
    }
    return true;
}
void PSystem::removeDeadParticles() {
    std::list<Attribute>::iterator i;
    i = _particles.begin();
    while (i != _particles.end()) {
        if (i->isAlive == false) i = _particles.erase(i);
        else i++;
    }
}
```

PSystem

```
void PSystem:: preRender() {
    _device->SetRenderState(D3DRS_LIGHTING, false);
    _device->SetRenderState(D3DRS_POINTSPRITEENABLE, true);
    _device->SetRenderState(D3DRS_POINTSCALEENABLE, true);
    _device->SetRenderState(D3DRS_POINTSIZE, d3d::FtoDw(_size));
    _device->SetRenderState(D3DRS_POINTSIZE_MIN, d3d::FtoDw(0.0f));
    // control the size of the particle relative to distance
    _device->SetRenderState(D3DRS_POINTSCALE_A, d3d::FtoDw(0.0f));
    _device->SetRenderState(D3DRS_POINTSCALE_B, d3d::FtoDw(0.0f));
    _device->SetRenderState(D3DRS_POINTSCALE_C, d3d::FtoDw(1.0f));
    // use alpha from texture
    _device->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
    _device->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);
    _device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
    _device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
    _device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
}

void PSystem:: postRender() {
    _device->SetRenderState(D3DRS_LIGHTING, true);
    _device->SetRenderState(D3DRS_POINTSPRITEENABLE, false);
    _device->SetRenderState(D3DRS_POINTSCALEENABLE, false);
    _device->SetRenderState(D3DRS_ALPHABLENDENABLE, false);
}
```

PSystem

- render() - 각 프레임에 다음과 같은 작업을 수행함
  - 모든 파티클을 갱신함
  - 모든 생존 파티클이 렌더링 될 때까지:
    - 만약 버텍스 버퍼가 가득 차지 않았다면:
      - D3DLOCK_NOOVER_WRITE flag로 segment i를 잠금
      - Segment i로 500 파티클을 복사
    - 만약 버텍스 버퍼가 가득 찼다면:
      - 버텍스 버퍼의 처음부터 시작. i=0
      - D3DLOCK_DISCARD flag로 segment i를 잠금
      - Segment i로 500 파티클을 복사
    - Segment i를 렌더링
    - 다음 segment로 i++



PSystem

```
void render() {
     // GPU와 CPU의 효과적인 활용을 위해, 여러 section별로 나누고,
     // 각 section에 대해서, data filling/rendering을 반복함.
     if (!_particles.empty()) {
         preRender();
         _device->SetTexture(0, _tex);
         _device->SetFVF(Particle::FVF);
         _device->SetStreamSource(0, _vb, 0, sizeof(Particle));
         if (vbOffset >= vbSize) _vbOffset = 0;
         Particle* v = 0;
         _vb->Lock(_vbOffset * sizeof(Particle), _vbBathSize*sizeof(Particle),
              (void**) &v, _vbOffset ? D3DLOCK_NOOVERWRITE: D3DLOCK_DISCARD);
         DWORD numParticlesInBatch = 0;
         std::list<Attribute>::iterator i;
         for (i = _particles.begin(); i != _particles.end(); i++) {
             if (i->_isAlive) {
                 v->_position = i->_position;
                 v->_color = (D3DCOLOR) i->_color;
                 v++;     // next element
                 numParticlesInBatch++;     // increase batch counter
                 if (numParticlesInBatch == _vbBatchSize) {
                     _vb->Unlock();
                     _device->DrawPrimitive(D3DPT_POINTLIST, _vbOffset, _vbBatchSize);
                     _vbOffset += _vbBatchSize;
```

PSystem

```
                     if (_vbOffset >= _vbSize) _vbOffset = 0;
                       _vb->Lock(_vbOffset *sizeof(Particle),
                              _vbBatchSize*sizeof(Particle), (void**) &v,
                              _vbOffset ? D3DLOCK_NOOVERWRITE : D3DLOCK_DISCARD);
                     numParticlesInBatch =0;
                 } // if full
             } // if alive
         } // for
         _vb->Unlock();
         if (numParticlesInBatch) {
             _device->DrawPrimitive(D3DPT_POINTLIST, _vbOffset, numParticlesInBatch);
         }
         _vbOffset += _vbBatchSize; // next block
     }
}
```

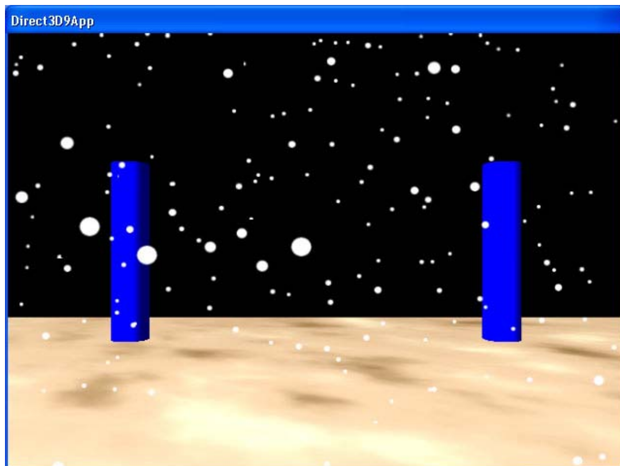
d3dUtility.h/cpp

```
// return random float in [lowBound, highBound] interval
float d3d::GetRandomFloat(float lowBound, float highBound) {
    if (lowBound >= highBound) // bad input
        return lowBound;
    // get random float in [0, 1] interval
    float f = (rand() % 10000)*0.0001f;
    // return float in [lowBound, highBound] interval
    return (f*(highBound – lowBound)) + lowBound;
}

// return a random vector in the bounds specified by min and max
float d3d::GetRandomVector(D3DXVECTOR3* out, D3DXVECTOR3* min, D3DXVECTOR3*
    max) {
    out->x = GetRandomFloat(min->x, max->x);
    out->y = GetRandomFloat(min->y, max->y);
    out->z = GetRandomFloat(min->z, max->z);
}

// rand()를 사용할 경우, 프로그램 초기화에서 srand()를 호출할 것
// seed random number generator
srand((unsigned int)time(0));
```

Psystem Extended Classes

```
#include "d3dUtility.h"
#include "camera.h"
#include <list>
namespace psys {
    class Snow : public Psystem {
    public:
        Snow (d3d::BoundingBox* boundingBox, int numParticles);
        void resetParticle(Attribute* attribute);
        void update(float timeDelta);
    };
    class Firework : public Psystem {
    public:
        Firework (D3DXVECTOR3* origin, int numParticles);
        void resetParticle(Attribute* attribute);
        void update(float timeDelta);
        void preRender();
        void postRender();
    };
    class ParticleGun : public Psystem {
    public:
        ParticleGun (Camera* camera);
        void resetParticle(Attribute* attribute);
        void update(float timeDelta);
    private:
        Camera* _camera;
    };
}
```

Example: Snow System



Snow

```
Snow::Snow(d3d::BoundingBox* boundingBox, int numParticles) {
    _boundingBox = *boundingBox;
    _size = 0.25f;
    _vbSize = 2048;
    _vbOffset = 0;
    _vbBatchSize = 512;
    for (int i=0; i<numParticles; i++)
        addParticle();
}
void Snow::resetParticle(Attribute *attribute) {
    attribute->_isAlive = true;
    d3d::GetRandomVector(&attribute->_position, &_boundingBox._min,
    &_boundingBox._max.y;
    attribute->_position.y = _boundingBox._max.y;
    // snow flakes fall downwards and slightly to the left
    attribute->_velocity.x = d3d::GetRandomFloat(0.0f, 1.0f) * -3.0f;
    attribute->_velocity.y = d3d::GetRandomFloat(0.0f, 1.0f) * -10.0f;
    attribute->_velocity.z = 0.0f;
    // white snow flake
    attribute->_color = d3d::WHITE;
}
```

Snow

```
void Snow::update(float timeDelta) {
    std::list<Attribute>::iterator i;
    for (int i=_particles.begin(); i!=_particles.end(); i++) {
        i->_position += i->_velocity * timeDelta;
        if (_boundingBox.isPointInside(i->_position) == false)
            resetParticle(&(*i));   // respawn instead of kill it
    }
}
```

Snow System Driver

```
psys::PSystem* TheSnow = 0;
Camera TheCamera(Camera::AITCRAFT);
bool Setup() {
    // seed random number generator
    srand((unsigned int)time(0));
    // Create Snow System.
    d3d::BoundingBox boundingBox;
    boundingBox._min = D3DXVECTOR3(-10.0f, -10.0f, -10.0f);
    boundingBox._max = D3DXVECTOR3( 10.0f,  10.0f,  10.0f);
    TheSnow = new psys::Snow(&boundingBox, 5000);
    TheSnow->init(Device, "snowflake.dds");
    // Create basic scene.
    d3d::DrawBasicScene(Device, 1.0f);
    // Set projection matrix.
    D3DXMATRIX proj;
    D3DXMatrixPerspectiveFovLH(&proj, D3DX_PI / 4.0f,
                        (float)Width / (float)Height, 1.0f, 5000.0f);
    Device->SetTransform(D3DTS_PROJECTION, &proj);
    return true;        }
```

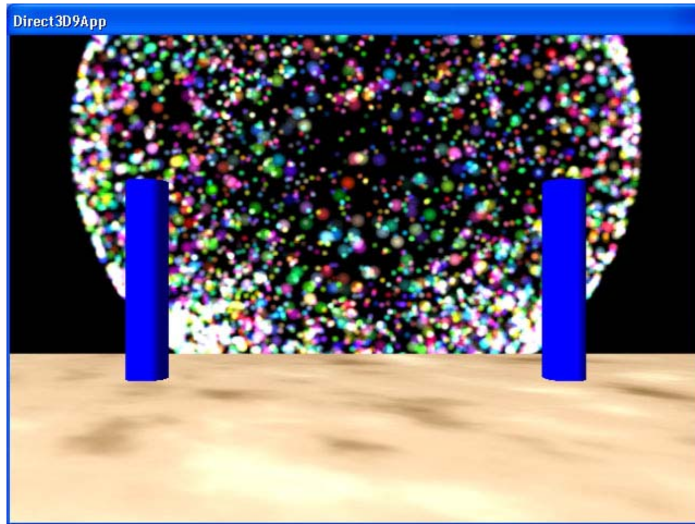
Snow System Driver

```
void Cleanup() {
    d3d::Delete<psys::PSystem*>( TheSnow );
    d3d::DrawBasicScene(0, 1.0f);
}
bool Display(float timeDelta) {
    if( Device )        {
        if( ::GetAsyncKeyState(VK_UP) & 0x8000f )
            TheCamera.walk(4.0f * timeDelta);
        if( ::GetAsyncKeyState(VK_DOWN) & 0x8000f )
            TheCamera.walk(-4.0f * timeDelta);
        // ….

        D3DXMATRIX V;
        TheCamera.getViewMatrix(&V);
        Device->SetTransform(D3DTS_VIEW, &V);

        TheSnow->update(timeDelta);
```

Snow System Driver

```
        // Draw the scene:
        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x00000000, 1.0f, 0);
        Device->BeginScene();
        D3DXMATRIX I;
        D3DXMatrixIdentity(&I);
        Device->SetTransform(D3DTS_WORLD, &I);

        d3d::DrawBasicScene(Device, 1.0f);

        // order important, render snow last.
        Device->SetTransform(D3DTS_WORLD, &I);
        TheSnow->render();

        Device->EndScene();
         Device->Present(0, 0, 0, 0);
    }
    return true;
}
```

Example: Firework System



Firework

```
Firework::Firework(D3DXVECTOR3* origin, int numParticles) {
    _origin = *origin;
    _size = 0.9f;
    _vbSize = 2048;
    _vbOffset = 0;
    _vbBatchSize = 512;
    for (int i=0; i<numParticles; i++)
        addParticle();
}
void Firework::resetParticle(Attribute *attribute) {
    attribute->_isAlive = true;
    attribute->_position = _origin;
    D3DXVECTOR3 min = D3DXVECTOR3(-1.0f, -1.0f, -1.0f);
    D3DXVECTOR3 max = D3DXVECTOR3( 1.0f,  1.0f,  1.0f);
    d3d::GetRandomVector(&attribute->_velocity, &min, &max);
    attribute->_velocity *= 100.0f;
    attribute->_color = D3DXCOLOR(d3d::GetRandomFloat(0.0f, 1.0f),
            d3d::GetRandomFloat(0.0f, 1.0f), d3d::GetRandomFloat(0.0f, 1.0f), 1.0f);
    attribute->_age = 0.0f;
    attribute->_lifeTime = 2.0f;            // lives for 2 seconds
}
```

Firework

```
void Firework::update(float timeDelta) {
    std::list<Attribute>::iterator i;
    for (int i=_particles.begin(); i!=_particles.end(); i++) {
        i->_position += i->_velocity * timeDelta;
        i->_age += timeDelta;
        if (i->_age > i->_lifeTime)
            i->_isAlive = false;    // kill
    }
}
void Firework::preRender() {
    PSystem::preRender();
    _device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
    _device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
    // read but don't write particles to z-buffer
    _device->SetRenderState(D3DRS_ZWRITEENABLE, false);
}
void Firewori::postRender() {
    PSystem::postRender();
    _device->SetRenderState(D3DRS_ZWRITEENABLE, true);
}
```

Firework System Driver

```
psys::PSystem* TheFirework = 0;
bool Setup() {
    // seed random number generator
    srand((unsigned int)time(0));
    // Create the Firework system.
    D3DXVECTOR3 origin(0.0f, 10.0f, 50.0f);
    TheFirework = new psys::Firework(&origin, 6000);
    TheFirework->init(Device, "flare.bmp");
    // Setup a basic scene.
    d3d::DrawBasicScene(Device, 1.0f);
    // Set projection matrix.
    D3DXMATRIX proj;
    D3DXMatrixPerspectiveFovLH(&proj, D3DX_PI / 4.0f, // 45 - degree
                    (float)Width / (float)Height, 1.0f, 5000.0f);
    Device->SetTransform(D3DTS_PROJECTION, &proj);

    return true;
}
```

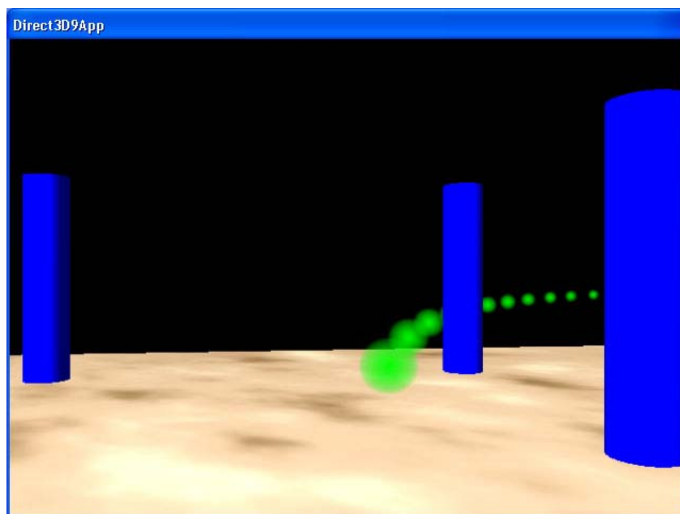
Firework System Driver

```
void Cleanup() {
    d3d::Delete<psys::PSystem*>( TheFirework );
    d3d::DrawBasicScene(0, 1.0f);
}
bool Display(float timeDelta) {
    if( Device )       {
        if( ::GetAsyncKeyState(VK_UP) & 0x8000f )
            TheCamera.walk(4.0f * timeDelta);
        if( ::GetAsyncKeyState(VK_DOWN) & 0x8000f )
            TheCamera.walk(-4.0f * timeDelta);
        // ….

        D3DXMATRIX V;
        TheCamera.getViewMatrix(&V);
        Device->SetTransform(D3DTS_VIEW, &V);

        TheFirework->update(timeDelta);
        if (TheFirework->isDead())
            TheFirework->reset();
```

Firework System Driver

```
        // Draw the scene:
        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x00000000, 1.0f, 0);
        Device->BeginScene();
        D3DXMATRIX I;
        D3DXMatrixIdentity(&I);
        Device->SetTransform(D3DTS_WORLD, &I);

        d3d::DrawBasicScene(Device, 1.0f);

        // order important, render snow last.
        Device->SetTransform(D3DTS_WORLD, &I);
        TheFirework->render();

        Device->EndScene();
        Device->Present(0, 0, 0, 0);
    }
    return true;
}
```

Example: Particle Gun System



Particle Gun

```
ParticleGun::ParticleGun(Camera* camera) {
    _camera = *camera;
    _size = 0.8f;
    _vbSize = 2048;
    _vbOffset = 0;
    _vbBatchSize = 512;
}
void ParticleGun::resetParticle(Attribute *attribute) {
    attribute->_isAlive = true;
    D3DXVECTOR3 cameraPos;
    _camera->GetPosition(&cameraPos);
    D3DXVECTOR3 cameraDir;
    attribute->_position = cameraPos;
    attribute->_position.y -= 1.0f;         // slightly below camera
    attribute->_velocity = cameraDir * 100.0f;
    attribute->_color = D3DXCOLOR(0.0f, 1.0f, 0.0f, 1.0f);
    attribute->_age = 0.0f;
    attribute->_lifeTime = 1.0f;            // lives for 1 seconds
}
```

Particle Gun

```
void ParticleGun::update(float timeDelta) {
    std::list<Attribute>::iterator i;
    for (int i=_particles.begin(); i!=_particles.end(); i++) {
        i->_position += i->_velocity * timeDelta;
        i->_age += timeDelta;
        if (i->_age > i->_lifeTime)
            i->_isAlive = false;    // kill
    }
    removeDeadParticles();
}
```

Particle Gun System Driver

```
psys::PSystem* TheGun = 0;
bool Setup() {
    // seed random number generator
    srand((unsigned int)time(0));
    // Create the Firework system.
    D3DXVECTOR3 origin(0.0f, 10.0f, 50.0f);
    TheGun = new psys::ParticleGun(&TheCamera);
    TheGun->init(Device, "flare_alpha.dds");
    // Setup a basic scene.
    d3d::DrawBasicScene(Device, 1.0f);
    // Set projection matrix.
    D3DXMATRIX proj;
    D3DXMatrixPerspectiveFovLH(&proj, D3DX_PI / 4.0f, // 45 - degree
                    (float)Width / (float)Height, 1.0f, 5000.0f);
    Device->SetTransform(D3DTS_PROJECTION, &proj);

    return true;
}
```

Particle Gun System Driver

```
void Cleanup() {
    d3d::Delete<psys::PSystem*>( TheGun );
    d3d::DrawBasicScene(0, 1.0f);
}
bool Display(float timeDelta) {
    if( Device )        {
        if( ::GetAsyncKeyState(VK_UP) & 0x8000f )
            TheCamera.walk(4.0f * timeDelta);
        if( ::GetAsyncKeyState(VK_DOWN) & 0x8000f )
            TheCamera.walk(-4.0f * timeDelta);
        // ….

        D3DXMATRIX V;
        TheCamera.getViewMatrix(&V);
        Device->SetTransform(D3DTS_VIEW, &V);

        TheGun->update(timeDelta);
```

Particle Gun System Driver

```
        // Draw the scene:
        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x00000000, 1.0f, 0);
        Device->BeginScene();
        D3DXMATRIX I;
        D3DXMatrixIdentity(&I);
        Device->SetTransform(D3DTS_WORLD, &I);

        d3d::DrawBasicScene(Device, 1.0f);

        // order important, render snow last.
        Device->SetTransform(D3DTS_WORLD, &I);
        TheGun->render();

        Device->EndScene();
        Device->Present(0, 0, 0, 0);
    }
    return true;
}
```