UDC 681.3

THE DEVELOPMENT OF THE HARDWARE INDEPENDENT PARTICLE SYSTEM

R.V. Malcheva, S.A. Kovalev, S. Koval, Mohammad Yunis
Donetsk National Technical University
E-mail: raisa@cs.dgtu.donetsk.ua

Introduction

The aim of the work is to develop the hardware independent Particle System. It is used to simulate certain fuzzy phenomena, which are otherwise very hard to reproduce with conventional rendering techniques. Examples of such phenomena which are commonly replicated using particle systems include fire, explosions, smoke, moving water, sparks, falling leaves, clouds, fog, snow, dust, meteor tails, hair, fur, grass, or abstract visual effects like glowing trails, magic spells.

Computer graphics researchers have long placed a significant emphasis on rendering aesthetically pleasing fire in 3D. Until now, efforts have failed to realistically capture the physical characteristics of fire. The natural randomness and turbulence of fire typically leads to a rendering solution based on well-understood calculations, such as Perlin noise and Gaussian distribution-based particle systems. These visual effects all point to a common model of fire but do not demonstrate the physical properties of fire beyond a single flame source.

Intel's Smoke demo [1] combines a traditional particle system that captures the visual effects of fire with a secondary system that treats fire as a heat source. This heat emitter dictates how a fire spreads by following a fuel source, how intense the fire is at any particular position, and the proximity of the heat source to another fuel source.

The Smoke demo architecture uses individual components defined as systems that house typical game engine features, such as physics, graphics, audio, and AI. A typical game entity in Smoke is an abstract object linked to several systems. For example, a horse in Smoke's farm scene has the following systems:

- Graphics for the model and skeletal animation.

- Geometry to control position and orientation.
- Physics for collision detection.
- Audio for sound effects.

The structural logic is the same for every object in the scene, such as the meteors that rain down from the sky, which include the graphics, physics, and audio systems as well as the fire system. The fire system is responsible for the demo's physical and visual properties of fire.

The procedural fire system consists of two discrete parts: a particle emitter based on a particle system inspired by Luna [2] that includes billboard flame textures (fire particles) and a heat emitter system that models the heat property of fire (heat particles). The Smoke demo includes a water hose (Fig.1) that allows users to move around the scene and extinguish the fire caused by the falling meteors.

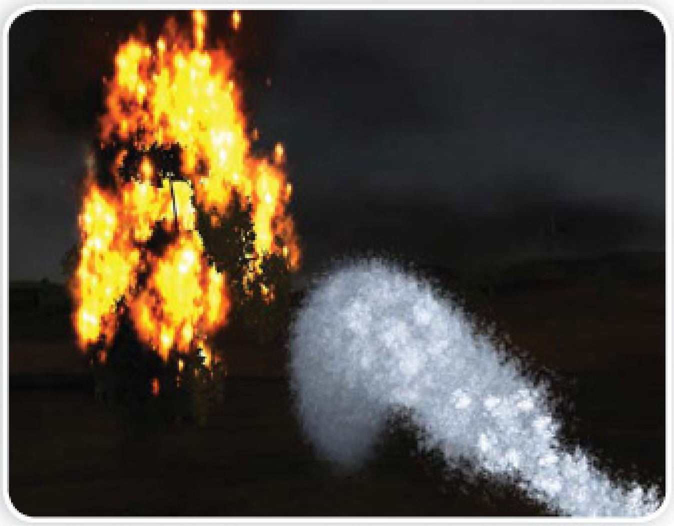

Fig.1. The fire system—water introduced as a cold emitter

As already noted, each fire object—in this case an entity bound to the geometric tree object—is iterated over every branch that is on fire. The fire object is checked against each heat emitter contained in the fire for a collision with an adjacent non-burning branch. Water is a natural extension of the fire system, and additional checks are used to extinguish a burning element in the collision checking code and to prevent the water from spreading, as the fire does, to another object.

A fuel source, such as a tree in the Smoke demo, consists of multiple branches and canopies (for each leaf cluster). However, any geometry, including meteor objects, can be a potential fuel source. Each branch and canopy of a tree can serve as a host to the fire's smart particle system. The system can use the host object's axis-aligned bounding box (AABB) to not only determine where the visual flame particles should be positioned but also to conduct collision checks in the heat emitter. Just as in a real fire, heat tends to spread upward and away from a heat source, moving up a tree from branch to branch, finally reaching the canopy. At the same time, fire can occasionally spread downward, following the path of a canopy to a branch [3].

Typical implementation

Typically a particle system's position and motion in 3D space are controlled by what is referred to as an emitter. The emitter acts as the source of the particles and its location in 3D space determines where they are generated and whence they proceed. A regular 3D mesh object, such as a cube or a plane, can be used as an emitter. The emitter has attached to it a set of particle behavior parameters. These parameters can include the spawning rate (how many particles are generated per unit of time), the particles' initial velocity vector (the direction they are emitted upon creation), particle lifetime (the length of time each individual particle exists before disappearing), particle color, and many more. It is common for all or most of these parameters to be "fuzzy" — instead of a precise numeric value, the artist specifies a central value and the degree of randomness allowable on either side of the center (i.e. the average particle's lifetime might be 50 frames ±20%). When using a mesh object as an emitter, the initial velocity vector is often set to be normal to the individual face(s) of the object, making the particles appear to "spray" directly from each face.

A typical particle system's update loop (which is performed for each frame of animation) can be separated into two distinct stages: the parameter update/simulation stage and the rendering stage.

Developing an Update method for the simulation stage

For the simulation stage the Particle Class is used. During the simulation stage, the number of new particles that must be created is calculated based on spawning rates and the interval between updates, and each of them is spawned in a specific position in 3D space based on the emitter's position and the spawning area specified. Each of the particle's parameters is initialized according to the emitter's parameters. They are: System lifetime, Emission, Particle lifetime, Direction, Spread, Start Speed, Gravity, Radial acceleration, Tangential Acceleration, Particle Size, Particle Spin, Alpha channel, Particle Color. At each update, all existing particles are checked to see if they have exceeded their lifetime, in which case they are removed from the simulation. Otherwise, the particles' position and other characteristics are advanced based on some sort of physical simulation, which can be as simple as translating their current position, or as complicated as performing physically-accurate trajectory calculations which take into account external forces.

For each particle the following operations are performed in Update method using the following formulas:

```
            vecAccel = par->vecLocation - vecLocation;
            vecAccel.Normalize();
            vecAccel2 = vecAccel;
            vecAccel *= par->fRadialAccel;
            // vecAccel2.Rotate(M_PI_2);
            // the following is faster
            ang = vecAccel2.mX;
            vecAccel2.mX = -vecAccel2.mY;
            vecAccel2.mY = ang;
            vecAccel2 *= par->fTangentialAccel;
            par->vecVelocity += (vecAccel + vecAccel2) * HGE_UPDATE_SPEED;
     if (par->fGravity != 0.0f) par->vecVelocity.mY += par->fGravity *
HGE_UPDATE_SPEED;
      par->vecLocation += par->vecVelocity;
            // updates size
            par->fSize += par->fSizeDelta * HGE_UPDATE_SPEED;
            theSprite->SetScale(FPoint(par->fSize, par->fSize));
            // updates color
            par->colColor[0] += par->colColorDelta[0] * HGE_UPDATE_SPEED;
            par->colColor[1] += par->colColorDelta[1] * HGE_UPDATE_SPEED;
            par->colColor[2] += par->colColorDelta[2] * HGE_UPDATE_SPEED;
            par->colColor[3] += par->colColorDelta[3] * HGE_UPDATE_SPEED;
            par->fSpin += par->fSpinDelta * HGE_UPDATE_SPEED;
```

It is common to perform some sort of collision detection between particles and specified 3D objects in the scene to make the particles bounce off of or otherwise interact with obstacles in the environment. Collisions between particles are rarely used, as they are computationally expensive and not really useful for most simulations. An example of polygon clipping is shown on the Fig. 2.

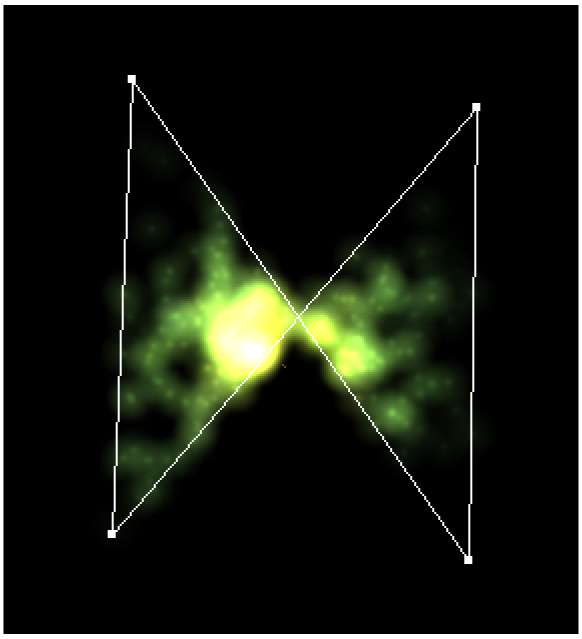

Fig. 2. An example of polygon clipping

Operation of polygon clipping is also performed:

```
    bool HGEParticleSystem::wn_PnPoly( Sexy::Point theTestPoint)
    {
           int    wn = 0;    // the winding number counter
           // loop through all edges of the polygon
           for (unsigned int i = 0; i < mPolygonClipPoints.size() - 1; i++)
           {   // edge from mPolygonClipPoints[i] to mPolygonClipPoints[i+1]
                 if (mPolygonClipPoints[i].mY <= theTestPoint.mY)
                 {
    // start y <= theTestPoint.mY
    if (mPolygonClipPoints[i+1].mY > theTestPoint.mY)   // an upward crossing
    if (isLeft( mPolygonClipPoints[i], mPolygonClipPoints[i+1], theTestPoint)
> 0)  // the TestPoint left of edge
     ++wn;            // have a valid up intersect
                 }
                 else
                 {
    // start y > theTestPoint.mY (no test needed)
    if (mPolygonClipPoints[i+1].mY <= theTestPoint.mY) // a downward crossing
    if (isLeft( mPolygonClipPoints[i], mPolygonClipPoints[i+1], theTestPoint)
< 0)  // theTestPoint right of edge
     --wn;            // have a valid down intersect
                 }
           }
           return (wn != 0);
    }
```

Specifics of a Rendering stage

This stage is unique for each hardware. After the update is complete, each particle is rendered, usually in the form of a textured billboarded quad (i.e. a quadrilateral that is always facing the viewer). However, this is not necessary; a particle may be rendered as a single pixel in small resolution/limited processing power environments. Particles can be rendered as Metaballs in off-line rendering; isosurfaces computed from particle-metaballs make quite convincing liquids. For each Particle Sprite class is used, which is directly implemented according using the API according to used hardware.

Implementation and Summary

The advantages of the developed Particle System are the following. It has lower system requirements then existing software. It gives 100 FPS on the machine without 3D acceleration.

Implementation for different hardware extends the spheres of using. Developed package consists of Editor and extensions for the engines. The Popcap Framework is used for PC, XNA framework is used for XBOX 360 [4] and NitroSystem is used for Nintendo DS [5].

1. *Ryan Shrout.* A Smoke Screen from Intel: Implementing Multi-threaded Gaming. - Intel, 2008.

2. *Luna, Frank D.* Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach. Plano, TX: Wordware Publishing, 2006.

3. An Overview of How to Accurately Model Procedurally Spreading Fire - http://software.intel.com/en-us/articles/smokegame-technology-demo-download.

4. www.mirpristavok.com.ua/index.php?cPath=33_46.

5. xnintendo.com/nintendo-ds/page/7.