UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Programación para videojuegos II

Unidad 4
Sistemas de partículas

Docente
Pablo Abratte

CONTENIDO

1. Sistemas de partículas

El término *sistema de partículas* se refiere a una técnica de gráficos por computadora que busca simular fenómenos difusos y estocásticos presentes en la realidad y que son difíciles de reproducir con técnicas convencionales. Ejemplos de dichos fenómenos incluyen fuego, humo, explosiones, aguas en movimiento y efectos visuales abstractos como destellos o hechizos, entre otras cosas.

Más específicamente, un sistema de partículas consiste en la representación gráfica de un fenómeno u objeto mediante un conjunto de masas puntuales o partículas, cuyo movimiento está sometido a fuerzas, restricciones y factores aleatorios.

Un sistema de partículas permite simular fenómenos u objetos difusos presentes en la naturaleza, mediante un conjunto de masas puntuales o partículas cuyo movimiento está sometido a fuerzas, restricciones y factores aleatorios.

A pesar de que algunos de los primeros videojuegos de computadora, como *Spacewar!* de 1960, ya utilizaban nubes de píxeles con movimiento aleatorio para simular explosiones (Fig 1), el término sistema de partículas fue acuñado por primera vez en 1983 por William T. Reeves en la publicación *Particle Systems, A Technique for Modeling a Class of Fuzzy Objects,* en la que describe los principios básicos del movimiento y de la representación de los mismos y cómo fueron utilizados por *LucasFilm Ltd.* para realizar los efectos especiales en la secuencia final de la película *Star Trek II: The Wrath of Khan.* Dichos principios continúan vigentes al día de hoy sin haber sufrido muchas alteraciones.

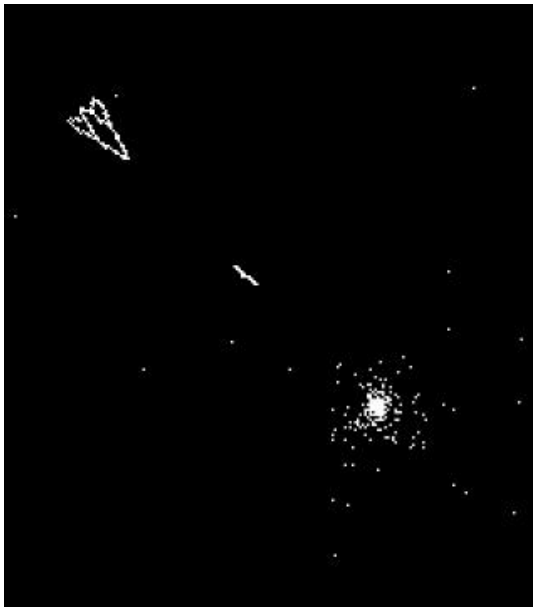

Figura 1. El videojuego *Spacewar!* de 1960 utilizaba nubes de puntos con movimiento aleatorio para representar explosiones.

Como se mencionó anteriormente, una partícula consiste en una masa puntual que posee propiedades que definen su comportamiento y representación. Estos atributos incluyen, entre otros, la posición y velocidad de la partícula, su tamaño y su color. El número de todos los parámetros posibles y sus variantes es incontable y depende del objetivo buscado.

En el código siguiente se presenta una estructura para representar una partícula genérica con sus parámetros más importantes:

```
struct Particle{
    float life;
    float x, y;
    float velx, vely;
    float angle;
    float alpha;
    float color_r, color_g, color_b;
```

```
    float size;
};
```

Generalmente, la posición y el movimiento de un sistema de partículas son controlados por un emisor (o *emitter* en inglés). Éste actúa como origen de las partículas y su ubicación en el espacio determina dónde son generadas las mismas.

Los emisores pueden adoptar distintas formas, siendo los más comunes los emisores puntuales. Otros tipos incluyen líneas, discos e inclusive objetos tridimensionales como cubos o planos.

Emisor o *emitter*: actúa como el originador de las partículas y posee atributos que definen el comportamiento de las mismas.

El emisor tiene asociado un conjunto de parámetros que definen el comportamiento de las partículas que genera. Éstos indican, entre otras cosas, la velocidad inicial de las partículas, su color y su tiempo de vida. Comúnmente, los valores de dichos parámetros se generan a partir de cierta variación aleatoria en lugar de valores numéricos precisos, para dar al sistema una apariencia más orgánica, propia de los fenómenos naturales. Por ejemplo, la vida media de las partículas podría ser de 5 segundos con una variación de ±2 segundos.

El ciclo de actualización de un sistema de partículas (que es realizado por cada cuadro de animación) puede ser separado en dos etapas, como puede observarse en la *Figura 2:* la etapa de simulación o actualización y la de renderizado.

A continuación, discutiremos algunos aspectos de cada una de las dos etapas.

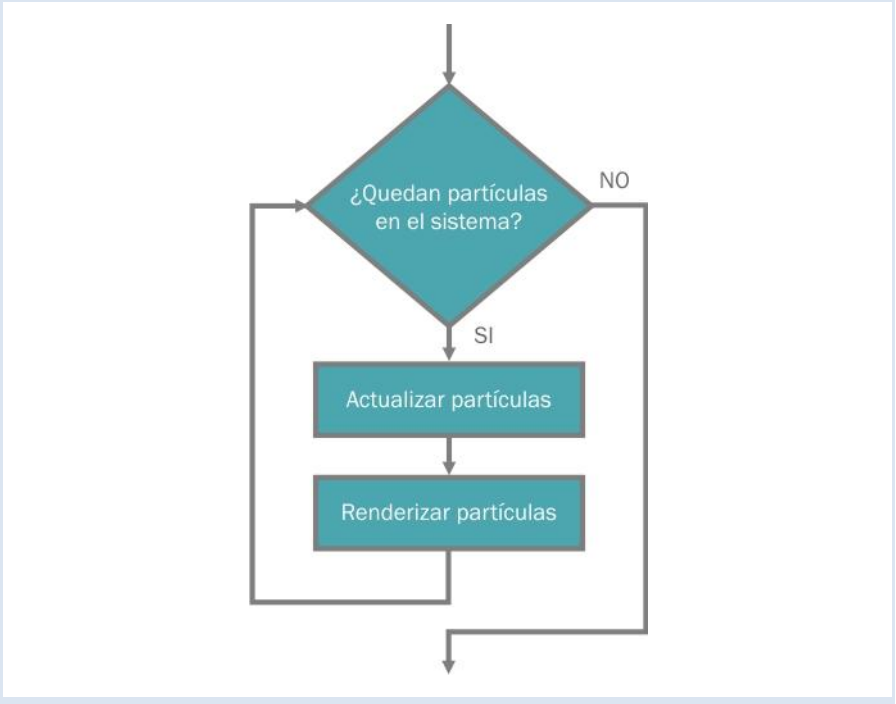

Figura 2. Ciclo de actualización de un sistema de partículas.

1.1. Simulación

En cada actualización se comprueba que las partículas existentes no hayan excedido su tiempo de vida, en cuyo caso son eliminadas de la simulación. También se crean nuevas partículas en base a la tasa de emisión del emisor y el tiempo transcurrido desde la última actualización. Cada uno de los parámetros de las nuevas partículas es inicializado de acuerdo a los parámetros del emisor.

También se actualizan el tiempo de vida de cada partícula, su posición y otros valores, según una serie de reglas que definen la simulación, como por ejemplo aplicar velocidad al movimiento, gravedad, fricción, etc.

Para realizar la simulación, puede optarse por una simulación con preservación o sin preservación de estado. Ambos métodos tienen sus ventajas y desventajas y resultan más apropiadas según los requerimientos del efecto que se busca lograr. A continuación, presentaremos ambas formas.

Simulación sin preservación de estado (stateless)

La *simulación sin preservación de estado* requiere que las propiedades de cada partícula sean computadas desde su nacimiento hasta su muerte por una función analítica (en inglés: *closed form function*), la cual se define mediante un conjunto de valores iniciales y el tiempo transcurrido desde el inicio de la simulación. Por ejemplo, la expresión:

$$p(t) = x_0 + v_0 t + \frac{1}{2} g t^2$$

define la posición vertical de una partícula para toda su trayectoria, a partir de su posición y velocidad inicial, la fuerza de gravedad y el tiempo transcurrido. La simulación mediante este tipo de funciones se denomina sin preservación de estado, ya que no se almacena información sobre ninguno de los estados anteriores de la partícula.

Este tipo de simulación es muy precisa y permite calcular de antemano las posiciones futuras de la partícula. Por otro lado, la mayoría de las veces es muy difícil o imposible encontrar una función analítica para describir el comportamiento deseado. Además, la utilización de este tipo de funciones hace imposible implementar colisiones de la partícula o reacciones a ambientes dinámicos.

Simulación con preservación de estado (state-preserving)

La *simulación con preservación de estado* implica utilizar integradores numéricos iterativos para actualizar los parámetros de las partículas en base a información sobre sus estados anteriores. Estos esquemas permiten aproximar de forma numérica la solución de una ecuación diferencial que expresa el movimiento o cambio en alguna de las magnitudes de la partícula. Si bien existen numerosos esquemas numéricos iterativos para aproximar las ecuaciones deseadas, aquí sólo presentaremos dos: el esquema de Euler hacia adelante (Forward Euler) y el de Verlet.

Un esquema de integración numérica es un algoritmo que permite hallar el valor aproximado de una integral o de la solución de una ecuación diferencial.

El *método de Euler hacia adelante* consiste en actualizar la velocidad en base a la aceleración y el paso de tiempo, y luego la posición en función de la nueva velocidad. Tomando como ejemplo el caso anterior, que expresaba la posición vertical de una partícula, dicho valor podría aproximarse haciendo lo siguiente:

$$v = \bar{v} + g \cdot \Delta t$$

$$p = \bar{p} + v \cdot \Delta t$$

El esquema permite actualizar la posición y velocidad de la partícula en base al tiempo transcurrido desde la última actualización (Δt), y su velocidad y posición en el instante de tiempo anterior ($\bar{v}$ y $\bar{p}$). Como puede observarse, para utilizar este método es necesario almacenar tanto la posición como la velocidad de la partícula en el instante anterior.

Este esquema resulta generalmente más rápido que utilizar una función analítica (en el caso de que pueda encontrarse una) y provee un resultado aproximado, como puede observarse en la *Figura 3.* Estos errores de aproximación disminuyen cuanto más pequeño es el paso de tiempo utilizado. Dado que, en general, al usar sistemas de partículas se hace hincapié en lograr un efecto visual agradable, la precisión en la simulación del movimiento de las partículas no es un factor demasiado importante.

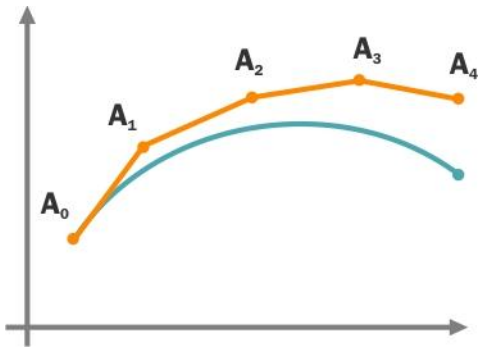

Figura 3. Gráfica de una función (azul) y su aproximación mediante el método de Euler a intervalos de tiempo regulares (anaranjado).

Modificando la velocidad al aplicar este esquema, es posible simular colisiones de la partícula con objetos del entorno. Por su simplicidad, el método de Euler hacia adelante es uno de los más utilizados cuando se necesita velocidad y la precisión no es un factor determinante, aunque generalmente produce buenos resultados al usar pasos de tiempo cortos. Este esquema es el que se emplea en las unidades anteriores para actualizar la posición del personaje.

El *método de Verlet,* por otro lado, actualiza el estado del sistema a partir de las dos posiciones anteriores, es decir:

$$p = 2\overline{p} - \overline{\overline{p}} + g \cdot \Delta t^2$$

En este caso, $\overline{p}$ es la posición anterior y $\overline{\overline{p}}$ la anterior a la anterior. Este esquema no guarda la velocidad de la partícula de manera explícita, pero es necesario almacenar las posiciones de la partícula en dos instantes anteriores. Esto hace complicado realizar cambios en la velocidad para simular colisiones.

Este tipo de esquemas no sólo permiten aproximar ecuaciones referidas al movimiento, sino también amortiguación, atracción y otras. Las ecuaciones de cambio de diversas propiedades de la partícula, como su color o transparencia, son generalmente bastante simples y no requieren de ecuaciones o esquemas complejos.

Para ahondar en conocimientos sobre diversos esquemas de integración y las diferencias entre ellos, puede tomarse como base el artículo http://en.wikipedia.org/wiki/Numerical_ordinary_differential_equations.

1.2 Renderizado

Las partículas pueden ser renderizadas utilizando puntos (GL_POINTS) o sprites. Esta última forma de representación resulta más costosa, pero produce resultados más agradables, ya que pueden utilizarse texturas –como la que se muestra en la *Figura 4*– para dar a la partícula distintas formas o una terminación más suave.

Figura 4. Textura utilizada para representar una partícula.

Al utilizar OpenGL, estos sprites se implementan con las primitivas GL_QUADS o GL_TRIANGLE_SPRIP utilizando texturas.

Otro factor importante a tener en cuenta es que si las texturas utilizadas para representar partículas poseen transparencias parciales, dichas partículas deberán ser ordenadas y dibujadas en orden para que las transparencias se visualicen correctamente.

Dado que el ordenamiento es computacionalmente costoso, generalmente es preferible evitarlo y utilizar un modo de blending que sea conmutativo, como los modos aditivo o multiplicativo.

En el modo aditivo, los valores de rojo, verde y azul de cada píxel son sumados con los valores respectivos del píxel superpuesto (y clampeados al valor máximo). Se dice que en este modo, los colores agregan luz al superponerse.

En el modo multiplicativo, por otro lado, los valores de rojo, verde y azul de los píxeles son multiplicados entre sí, y un color le quita luz a otro al superponerse. En la *Figura 5* puede observarse, a la derecha, que tres cuadrados con colores rojo, verde y azul se superponen con blending aditivo sobre un fondo cuadriculado blanco y negro, mientras que a la izquierda se ilustra la misma situación con blending multiplicativo y tres cuadrados de color cian, magenta y amarillo.

Dado que tanto la suma como la multiplicación son operaciones conmutativas, no tiene importancia cuál de los píxeles está por delante y cuál por detrás, prescindiendo de cualquier tipo de orden en los píxeles que deben ser dibujados.

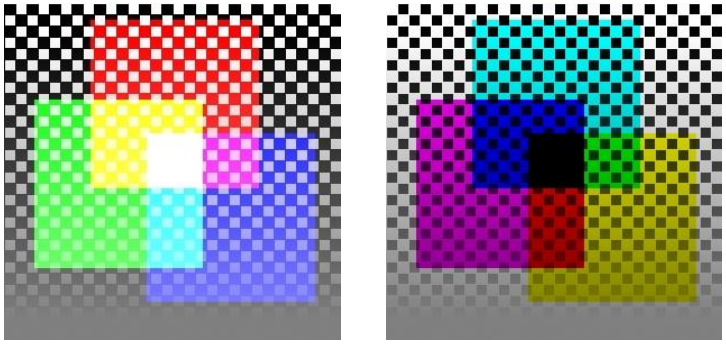

Figura 5. Cuadrados de color rojo, verde y azul superponiéndose con blening aditivo (izq). Cuadrados de color cian, magenta y amarillo superponiéndose con blending multiplicativo (der),

2. Implementación de un sistema de partículas básico

La implementación de un sistema de partículas no presenta demasiadas dificultades a grandes rasgos. Sin embargo, cuando el mismo debe ser simulados y renderizado en tiempo real, la eficiencia y optimización de recursos se vuelven factores críticos, ya que una operación tan simple como una multiplicación para actualizar la posición de una partícula se convierte en miles de operaciones cuando debe moverse un sistema con un gran número de partículas.

Existen dos limitaciones principales en el renderizado de sistemas de partículas que constituyen los cuellos de botella en las implementaciones de sistemas de partículas: la tasa de dibujado o *fillrate* y el ancho de banda o velocidad de transferencia del bus entre la CPU y la GPU.

El *fillrate* es la cantidad de píxeles por cuadro que la GPU es capaz de dibujar. En el caso de sistemas con partículas de gran tamaño existe un gran solapamiento, por lo que cada píxel es dibujado varias veces. Como el realismo de un sistema de partículas

generalmente se incrementa al utilizar partículas de menor tamaño, esta limitación pierde importancia.

Por otro lado, el bus entre la CPU y la GPU se comparte con otras tareas de renderizado, limitando a los sistemas de partículas a una cantidad aproximada de 10.000 partículas por cuadro en una aplicación típica. Para minimizar la transferencia de información desde la CPU a la GPU y superar este límite puede realizarse tanto la simulación como el renderizado del sistema de partículas en la GPU. Pero no trataremos este tema aquí.

A continuación, presentaremos la implementación de un sistema de partículas básico y para propósito general, sin ningún tipo de optimización especial. El diseño a grandes rasgos de las clases puede observarse en la *Figura 6*.

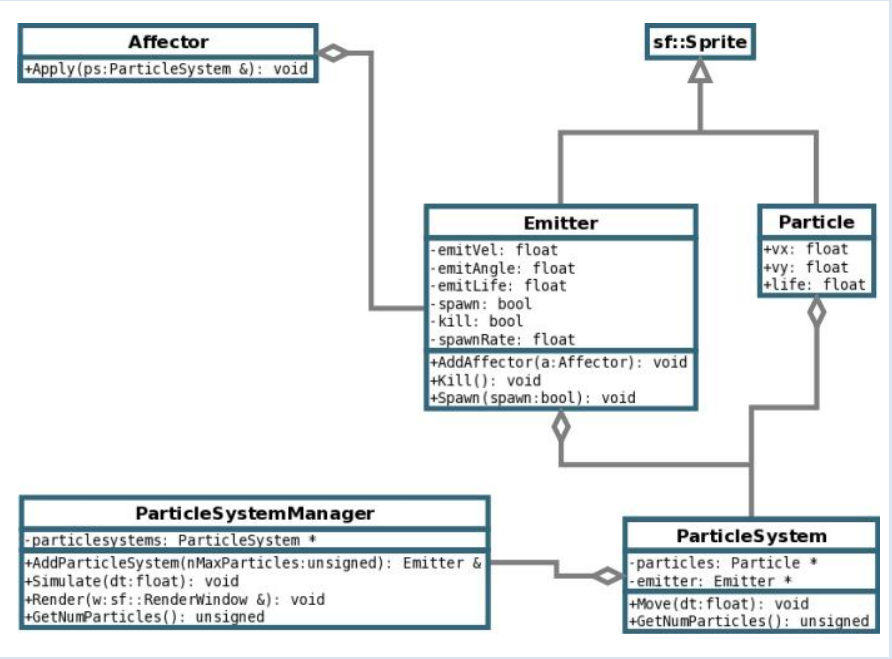

Figura 6. Diagrama de clases de la implementación de sistemas de partículas.

En el diagrama puede observarse que existe una clase *Particle* que hereda de la clase *sf::Sprite,* ya que esta última posee la mayoría de los atributos y comportamientos deseados para la partícula. La clase *Particle* incluye, además, atributos para almacenar la velocidad de la partícula y la vida que le resta a la misma.

La clase *Emitter,* que modela un emisor, también hereda de la clase *sf::Sprite.* Esto será útil para inicializar atributos de las partículas (como su tamaño, imagen, transparencia, etc.), a partir de las propiedades del emisor. Los atributos *spawn* y *spawnRate* indican, respectivamente, si el emisor está creando o no partículas y la cantidad de partículas emitidas por segundo. La clase incluye, además, un arreglo de *Affectors,* los cuales son funciones que modifican el comportamiento de las partículas emitidas y permiten un diseño modular de los sistemas de partículas, especificando qué fuerzas o cambios se desea que actúen sobre las partículas.

La clase *ParticleSystem* se encarga de asociar a un emisor con un conjunto de partículas y posee métodos para mover el sistema de partículas y contar cuantas partículas vivas hay en el sistema en un momento dado.

La clase *ParticleSystemManager* se encargará de la gestión de todos los sistemas de partículas; se ocupará de crear, mover y destruir los sistemas de partículas. Será la única clase con la que el usuario podrá interactuar para incluir un nuevo sistema de partículas en el juego.

El método *AddParticleSystem()* recibe la cantidad máxima de partículas que podrán existir simultáneamente en el sistema y se encarga de reservar memoria para las

mismas, así como de crear un emisor para el sistema, que será devuelto al usuario para que este último modifique el comportamiento del sistema.

Ahora veremos con más detalle la implementación de las clases mostradas en la *Figura 5*.

En el siguiente código podrán observar la declaración de la clase *Emitter.* No expondremos aquí la implementación de sus métodos, ya que son bastante sencillos. Sin embargo, cabe destacar que el constructor de dicha clase fue declarado bajo la etiqueta *private.* Esto no se trata de un error; fue hecho intencionalmente para que el usuario no pueda crear objetos de dicha clase, sino que sea la clase amiga *ParticleSystemManager* la encargada de crear y destruir dichos objetos. La amistad con la clase *ParticleSystem* es para que esta última pueda acceder a los atributos del emisor cuando sea necesario inicializar nuevas partículas:

```
#ifndef EMITTER_H
#define EMITTER_H
#include <SFML/Graphics.hpp>
#include "Affectors.h"
#include <vector>
using namespace std;

class Emitter: public sf::Sprite{
    private:
    // medias de velocidad y angulo de emision y vida
    float emitVel, emitAngle, emitLife;
    // desvios de velocidad y angulo de emision y vida
    float deltaVel, deltaAngle, deltaLife;
    // tasa de creacion de particulas
    float spawnRate;
    // si debe crear nuevas particulas o destruirse
    bool spawn, kill;

    // los afectadores del sistema de particulas
    vector<Affector *> affectors;

    // constructor, es utilizado por
    // la clase ParticleSystemManager
    Emitter(float emitVel=5, float emitAngle=M_PI,
            float emitLife=3, float deltaVel=0,
            float deltaAngle=M_PI, float deltaLife=1,
            float spawnRate=10);

    public:
    // agrega un affector al emisor
    void AddAffector(Affector &e);
    // permite controlar parametros de la emision
    void SetEmmitVel(float vel, float deltaVel=0);
    void SetEmmitAngle(float angle, float deltaAngle=0);
    void SetEmmitLife(float life, float deltaLife=0);
    void SetSpawnRate(float newSpawnRate);

    // indica si el emisor debe o no crear
    // nuevas particulas
    void Spawn(bool spawn);

    // senala que el emisor no seguira
    // emitiendo particulas y puede
    // liberarse
    void Kill();

    friend class ParticleSystem;
    friend class ParticleSystemManager;
};

#endif
```

El código mostrado a continuación corresponde a la declaración de la clase *ParticleSystem.* Como se mencionó anteriormente, esta clase asocia un emisor con un conjunto de partículas y tiene la responsabilidad de crear y realizar el movimiento de

las mismas de acuerdo a las características del emisor. Asimismo, es posible observar que la clase *Particula* está definida dentro de esta clase para lograr un diseño más organizado:

```
class ParticleSystem {
    public:
    // definicion de la clase particula
    struct Particle: public sf::Sprite{
        float life;
        float vx, vy;
    };

    // numero maximo de particulas y
    // cantidad de particulas vivas
    unsigned nMaxParticles, nParticles;
    // puntero a las particulas
    Particle *particles;
    // el emisor del sistema
    Emitter *emitter;

    ParticleSystem(Emitter &e, unsigned nMaxParticles=100);
    ~ParticleSystem();
    // mueve el sistema de particulas
    void Move(float &dt);
    // devuelve la cantidad de particulas vivas
    unsigned GetNumParticles(){return nParticles;};
};
```

De esta clase analizaremos su constructor y la función *Move(),* encargada de realizar el movimiento de las partículas.

El código del constructor puede observarse debajo. El mismo se encarga de reservar memoria para las partículas e inicializa su vida en -1 para que, al realizar la simulación, sean reinicializadas si el emisor está activado;

```
// construye el sistema de particulas asociandole un emisor
// y con la cantidad maxima de particulas dada
ParticleSystem::ParticleSystem(Emitter &e, unsigned nMaxParticles){
    this->emitter=&e;
    nParticles=0;

    // inicializa las particulas
    this->nMaxParticles=nMaxParticles;
    particles=new Particle[nMaxParticles];
    for(unsigned i=0; i<nMaxParticles; i++){
        // inicializa la vida de las particulas en -1 para que sean
        // reiniciadas al mover
        particles[i].life=-1;
    }
}
```

A continuación puede observarse la función *Move(),* encargada de realizar el movimiento de las partículas.

La función recorre primero los afectores asociados al emisor y los aplica a las partículas del sistema. La variable estática *particlesToSpawn* contiene la cantidad de partículas que el emisor puede crear y es incrementada en base al tiempo transcurrido y la tasa de emisión de partículas del emisor. Luego, se recorre cada partícula del sistema: en caso de que la partícula ya haya cumplido su ciclo de vida, si el emisor está activado y pueden crearse nuevas partículas según la tasa de emisión, se reinicializa la partícula en base a las propiedades del emisor. Si la partícula aún está viva, se decrementa su vida y se aplica movimiento a la misma:

```
// devuelve un numero aleatorio con distribucion uniforme entre
// [media-desv, media+desv]
#define RAND_MEDIA_DESV(media, desv) media-
desv+(rand()/float(RAND_MAX))*2*desv
```

```
void ParticleSystem::Move(float &dt){
    // aplica a las particulas los efectos asociados al emisor
    vector<Affector *>::iterator e=emitter->affectors.begin();
    while(e!=emitter->affectors.end()){
        (*e)->Apply(*this);
        e++;
    }

    // la cantidad de particulas que podemos lanzar
    float static particlesToSpawn=0;

    // actualizamos la cantidad de particulas que podemos crear
    // en base al tiempo transcurrido y al spawnRate del emisor
    if(emitter->spawn)
        particlesToSpawn+=emitter->spawnRate*dt;

    // movemos las particulas
    for(unsigned i=0; i<nMaxParticles; i++){
        Particle &p=particles[i];

        // si esta muerta
        if(p.life<=0){
            // si podemos, creamos otra particula que tome su lugar
            if(emitter->spawn && particlesToSpawn>0){
                float vel, angle;
                // inicializa propiedades de la particula segun el emisor
                p.SetImage(*emitter->GetImage());
                p.SetBlendMode(emitter->GetBlendMode());
                p.SetColor(emitter->GetColor());
                p.SetPosition(emitter->GetPosition().x,
                              emitter->GetPosition().y);
                p.SetCenter( emitter->GetImage()->GetWidth()/2,
                             emitter->GetImage()->GetHeight()/2);

                // inicializa los parametros aleatorios de la particula
                // divide la velocidad en sus componentes segun el angulo
                p.life=RAND_MEDIA_DESV( emitter->emitLife,
                                        emitter->deltaLife);
                vel=RAND_MEDIA_DESV(emitter->emitVel, emitter->deltaVel);
                angle=RAND_MEDIA_DESV( emitter->emitAngle,
                                       emitter->deltaAngle);
                p.vx=vel*cos(angle);
                p.vy=vel*sin(angle);

                // aumentamos el nro de particulas del sistema
                // y restamos la particula creada al radio
                // de particulas que podemos crear
                nParticles++;
                particlesToSpawn-=1;
            }
        }else{
            // decrementamos el tiempo de vida de la particula
            p.life-=dt;
            if(p.life<0) nParticles--;
            // mueve la particula (con la funcion Move() de sprite)
            else p.Move(p.vx*dt, p.vy*dt);
        }
    }
}
```

El código que se muestra debajo corresponde a la declaración de la clase *ParticleSystemManager*. La misma posee una lista con todos los sistemas de partículas, así como de los emisores, y es la responsable de crearlos y destruirlos.

Anteriormente mencionamos que el usuario no podía crear emisores y sistemas de partículas sino a partir de esta clase. Sin embargo, como es posible observar, que el único constructor de esta clase se encuentra bajo la etiqueta *private*, impidiendo que el usuario cree instancias de la clase:

```
class ParticleSystemManager{
    private:
    // los sistemas de particulas
    list<ParticleSystem *> particlesystems;
    // el manejador global de sistemas de particulas
    static ParticleSystemManager *globalManager;

    ParticleSystemManager();
    ~ParticleSystemManager();

    public:
    // devuelve el manejador global
    static ParticleSystemManager &GetManager();
    // agrega un sistemas de particulas
    Emitter &AddParticleSystem(unsigned nMaxParticles=1000);
    // mueve todos los sistemas de particulas
    void Simulate(float dt);
    // dibuja todos los sistemas de particulas
    void Render(sf::RenderWindow &w);
    // devuelve la cantidad de particulas entre
    // todos los sistemas
    unsigned GetNumParticles();
};
```

Asimismo podemos ver que la clase posee como variable estática un puntero a un objeto de la misma clase y una función *GetManager()* que también es estática. A continuación, presentamos el código de dicha función:

```
// devuelve el manager global de particulas
ParticleSystemManager &ParticleSystemManager::GetManager(){
    if(!globalManager)
        globalManager=new ParticleSystemManager();
    return *globalManager;
}
```

La función se encarga de inicializar el objeto *globalManager* y entregárselo al usuario. Como mencionamos anteriormente, el usuario no puede crear objetos de tipo *ParticleSystemManager.* No obstante, puede obtener una instancia a partir de esta función estática y la clase se encargará de que sólo exista una única instancia de la misma.

Dado que un solo objeto puede manejar todos los sistemas de partículas del juego, no tiene sentido que existan varios manejadores de sistemas de partículas, ya que sólo aumentaría la complejidad del código. Este tipo de clases que poseen un solo objeto se denominan *clases singleton.*

Presentaremos a continuación la implementación de las funciones más importantes de la clase. El código de *AddParticlesSystem()* se muestra debajo. La función recibe la cantidad máxima de partículas en el sistema deseado y se encarga de crearlo, junto con un emisor cuya referencia es devuelta al usuario para que éste pueda modificar la posición y el comportamiento del sistema:

```
// crea un emisor y un nuevo sistema de particulas
// y nos devuelve una referencia a al emisor
Emitter &ParticleSystemManager::AddParticleSystem(unsigned maxParticles){
    Emitter *e=new Emitter;
    particlesystems.insert(    particlesystems.end(),
                               new ParticleSystem(*e, maxParticles));
    return *e;
}
```

El método *Simulate()* de la clase se encarga de mover todos los sistemas de partículas, quitando de la lista los sistemas que ya no tengan partículas y cuyo emisor haya sido marcado con la bandera *kill.*

A continuación es posible observar el código de dicha función:

```
// mueve las particulas de todos los sistemas
void ParticleSystemManager::Simulate(float dt){
    list<ParticleSystem *>::iterator ps=particlesystems.begin();
    // mueve todos los sistemas
    while(ps!=particlesystems.end()){
        (*(ps))->Move(dt);
        if((*(ps))->nParticles==0 && (*(ps))->emitter->kill){
            ps=particlesystems.erase(ps);
        }else ps++;
    }
}
```

Luego, la función *Render()* que mostramos debajo recorre las partículas de cada sistema y dibuja en la ventana *w* las que están vivas. Como mencionamos anteriormente, las partículas son representadas mediante sprites de SFML:

```
// dibuja las partículas de todos los sistemas en la ventana w
void ParticleSystemManager::Render(sf::RenderWindow &w){
    list<ParticleSystem *>::iterator ps=particlesystems.begin();
    while(ps!=particlesystems.end()){
        ParticleSystem::Particle *p=(*ps)->particles;
        for(unsigned i=0; i<(*ps)->nMaxParticles; i++, p++){
            if(p->life>=0){
                w.Draw(*p);
            }
        }
        ps++;
    }
}
```

Por último, los afectadores de las partículas deben heredar de la clase abstracta *Affector,* que presentamos a continuación, e implementar el método *Apply()* que recibe el sistema de partículas y debe recorrerlo, modificando las propiedades de cada partícula:

```
class Affector {
    public:
    virtual void Apply(ParticleSystem &ps)=0;
};
```

Como ejemplo, mostramos la implementación de un afectador denominado *Gravity,* el cual modifica las velocidades de las partículas para aplicarles gravedad mediante el método de Euler:

```
class Gravity:public Affector{
private:
    float gx, gy;
public:
    Gravity(float gx, float gy);
    virtual void Apply(ParticleSystem &ps);
};

Gravity::Gravity(float gx, float gy){
    this->gx=gx;
    this->gy=gy;
}

void Gravity::Apply(ParticleSystem &ps){
    unsigned cont=0;
    ParticleSystem::Particle *p=ps.particles;
    while(cont<ps.nMaxParticles){
        if(p->life>=0){
            p->vx+=gx;
            p->vy+=gy;
```

```
        }
        cont++; p++;
    }
}
```

En el siguiente código se muestra un ejemplo de utilización de las clases implementadas. Se crea un sistema de partículas y se modifican sus parámetros de emisión para simular una fuente de fuego. Asimismo, se añaden afectores para que las partículas se eleven y se desvanezcan a medida que pasa el tiempo. El modo de blending utilizado es aditivo. En la *Figura 7* se muestra la imagen utilizada para las partículas y una captura del efecto logrado.

```
#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
#include "ParticleSystemManager.h"
#include <iostream>
#include <cstdio>
using namespace std;
using namespace sf;

int main(int argc, char *argv[]) {
    RenderWindow w(VideoMode(800,600),"Ejemplo Sistema de Particulas");

    // carga la imagen de la paricula
    sf::Image i;
    i.LoadFromFile("particle.png");

    // obtiene el manejador de sistemas de particulas
    ParticleSystemManager &mg=ParticleSystemManager::GetManager();

    // crea un sistema de particulas y ajusta parametros del emisor
    Emitter &em1=mg.AddParticleSystem(2000);
    em1.SetImage(i);
    em1.SetEmmitVel(70, 70);
    em1.SetEmmitLife(1, 0.5);
    em1.SetBlendMode(sf::Blend::Add);
    em1.SetSpawnRate(2000);
    em1.Spawn(false);

    // crea 2 efectos y los aniade al emisor
    Affector *g=new Gravity(0,-9.8);
    Affector *f=new Fade(1.2);

    em1.AddAffector(*g);
    em1.AddAffector(*f);

    // la entrada
    const sf::Input& Input = w.GetInput();

    while(w.IsOpened()) {
        sf::Event e;
        while(w.GetEvent(e)) {
            if(e.Type == e.Closed)
                w.Close();
            // hace que se emitan particulas segun los clicks
            if(e.Type == e.MouseButtonPressed) em1.Spawn(true);
            if(e.Type == e.MouseButtonReleased) em1.Spawn(false);
        }
        // el tiempo transcurrido
        elapsedTime=w.GetFrameTime();

        // ajusta la posicion del emisor segun el mouse
        em1.SetPosition(Input.GetMouseX(),Input.GetMouseY());

        // limpia la pantalla, simula y dibuja el sistema
        w.Clear(Color(0,0,0,255));
        mg.Simulate(elapsedTime);
        mg.Render(w);

        // actualiza la pantalla
```

```
        w.Display();
    }
    return 0;
}
```

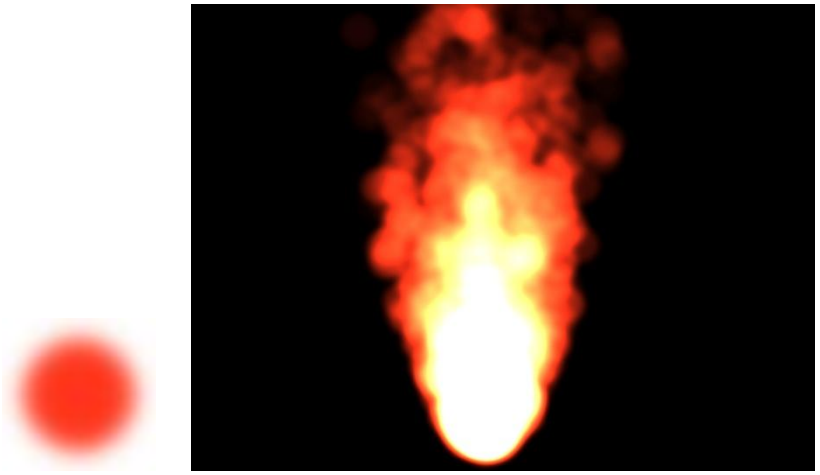

Figura 7. Partícula utilizada y efecto de fuego logrado a partir de la misma, utilizando blending aditivo.

3. Optimización del sistema de partículas

Dijimos que la optimización es un factor clave en los sistemas de partículas, ya que se trata de una técnica que consume una ingente cantidad de recursos. A continuación, discutiremos algunos defectos y posibles mejoras de las clases presentadas anteriormente.

La implementación mostrada posee varios aspectos que podrían optimizarse. El más notable a grandes rasgos es el uso de afectadores para que sea posible acoplar distintos efectos a los sistemas de partículas.

Este diseño implica varias desventajas. Una de ellas es la necesidad de recorrer varias veces el sistema de partículas para aplicar los distintos efectos. Otra desventaja es la invocación polimórfica de la función *Apply* para la clase *Affector.* El polimorfismo muchas veces es enemigo del rendimiento, ya que hace necesario que al momento del llamado a la función, el programa deba decidir en tiempo de ejecución a qué tipo de objeto apunta el puntero.

Una forma posible de superar los defectos comentados sería proveer al sistema de partículas con una única función que, durante una sola iteración sobre el sistema, actualice el comportamiento completo de las partículas aplicando todos los efectos. Además, la clase *Emitter* podría recibir los efectos como templates de funciones para que la resolución a las llamadas de las mismas pueda realizarse en tiempo de compilación en vez de ejecución. Estas modificaciones complicarían la utilización de las clases y la composición modular de los sistemas de partículas creados, por lo que el buen diseño y la eficiencia muchas veces son factores entre los cuales es difícil encontrar un balance adecuado.

Asimismo, comentamos que uno de los factores más importantes que limitan el rendimiento de los sistemas de partículas es el ancho de banda y la velocidad de transferencia del bus entre la GPU y la CPU. En el siguiente parágrafo aprenderemos a utilizar una extensión de OpenGL que nos permitirá economizar la utilización de dicho bus.

3.1. La extensión GL_POINT_SPRITE

La biblioteca SFML dibuja los sprites como cuadrados con textura. Esta forma de representación es también utilizada por otras plataformas, como OpenGL, y es una de las maneras más comunes de dibujar una partícula. Sin embargo, hacer esto implica enviar cuatro vértices al pipeline gráfico por cada sprite o partícula que se dibuje.

La *Figura 8* ilustra esto. Si el centro de la partícula está en las coordenadas *cx* y *cy,* y la textura del sprite tiene como dimensiones *t_ancho* y *t_alto,* los cuatro vértices enviados al pipeline gráfico tienen coordenadas de posición (*cx-t_ancho*/2, *cy-t_alto*/2), (*cx+t_ancho*/2, *cy-t_alto*/2), (*cx+t_ancho*/2, *cy+t_alto*/2) y (*cx-t_ancho*/2, *cy+t_alto*/2).

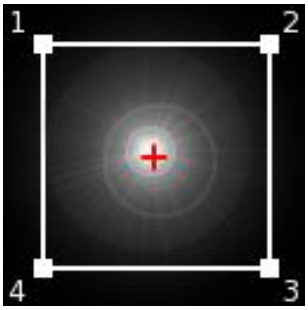

Figura 8. Partícula representada con un sprite, dibujado con cuatro vértices.

Esta situación puede mejorarse utilizando la extensión *GL_POINT_SPRITE,* la cual permite enviar sólo el punto central de la partícula y que el hardware gráfico se encargue de computar los cuatro vértices a partir del mismo. Antes de usar una extensión, los programas deben comprobar su disponibilidad y después acceder a las nuevas funcionalidades ofrecidas. Este proceso es dependiente del hardware de video, pero existen bibliotecas como GLEW y GLEE que simplifican este proceso.

SFML utiliza la biblioteca GLEW, por lo que no es necesario inicializar la biblioteca para utilizarla. El siguiente código muestra cómo averiguar si el hardware gráfico soporta point sprites:

```
// averiguamos si podemos usar pointSprites
if(!glewIsSupported("GL_ARB_point_sprite"))
        cout<<"ERROR: extension GL_ARB_point_sprite no disponible”;
```

El siguiente código muestra la función *Render()* de la clase *ParticleSystemManager* para que renderice las partículas utilizando point sprites:

```
void ParticleSystemManager::Render_PointSprites(sf::RenderWindow &w){
    list<ParticleSystem *>::iterator ps=particlesystems.begin();
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);

    glEnable(GL_POINT_SPRITE);
    glTexEnvi(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);

    while(ps!=particlesystems.end()){
        ParticleSystem::Particle *p=(*ps)->particles;
        glPointSize(p->GetSize().x);
        (*ps)->emitter->GetImage()->Bind();
        glBegin(GL_POINTS);

        sf::Vector2f particlePosition;
        for(unsigned i=0; i<(*ps)->nMaxParticles; i++, p++){
            if(p->life>=0){
                glColor4f(1,1,1,p->GetColor().a/255.0);
                particlePosition=p->GetPosition();
                glVertex2f(particlePosition.x, particlePosition.y);
            }
```

```
        }
        ps++;
        glEnd();
    }
    glDisable(GL_POINT_SPRITE);
}
```

Para habilitar la utilización de point sprites es necesario llamar a *glEnable()* con la constante GL_POINT_SPRITE y a *glTexEnvi()* de la forma mostrada en el código, para que las coordenadas de textura sean manejadas de forma acorde a la extensión.

El próximo paso es habilitar la textura para que sea renderizada. Esto puede hacerse con la función *Bind()* de la clase *sf::Image.* Luego, cada partícula es dibujada especificando solamente su punto central mediante la orden *glVertex().* En tanto, el tamaño de la partícula es detallado mediante la orden *glPointSize().*

La utilización de point sprites tiene sus limitaciones, ya que no se puede alterar el tamaño del punto ni la textura utilizada para el dibujado entre las sentencias *glBegin()* y *glEnd(),* por lo que todas las partículas del sistema deberán tener la misma textura y tamaño. Hay formas posibles de superar esta dificultad, pero son un poco más complicadas o ineficientes.

Mediante la utilización de point sprites se reduce en un cuarto la cantidad de vértices enviados al pipeline gráfico. Por tal motivo, es de esperar que la tasa de cuadros por segundo se cuadruplique. En la práctica, el incremento en el rendimiento no siempre es ese, pero sí se consigue un aumento notable en la cantidad de cuadros por segundo.

Bibliografía

Lutz Latta. “Building a Million Particle System” en Game Developers Conference 2004 [en línea] [2LDigital] http://www.2ld.de/gdc2004/

Reeves, William. “Particle Systems. A Technique for Modeling a Class of Fuzzy Objects”. Computer Graphics, v. 17, n° 3, 1983.

Wikipedia [en línea]

http://en.wikipedia.org/wiki/Particle_system

http://en.wikipedia.org/wiki/Numerical_ordinary_differential_equations