PRÁCTICA 1. EMISOR DE PARTÍCULAS

Lucas Joglar, Lucía González,

Sergio Ponce, Hugo García

2ºA-TARDE PROGRAMACIÓN AVANZADA

ÍNDICE

1.Introducción

En el siguiente documento se describe cómo se ha llevado a cabo la elaboración del código que tiene como finalidad la creación de un emisor de partículas con colisiones.

2.Desarrollo del código

Tomando de partida anteriores proyectos realizados en clase, para la realización del emisor ya se cuenta con elementos fundamentales ya creados, como las clases *Cube*, *Sphere* o *Teapot*. Sin embargo, tal y como es apreciable en el resultado final, de dichas clases para las partículas únicamente se ha hecho uso de *Sphere.*

Clase EmmiterConfiguration

En la clase EmmiterConfiguration se lleva a cabo la configuración del número de partículas, el tiempo de emisión y el puntero a la clase *Solid* para el emisor.

Para ello se han declarado en dicha clase dos variables de tipo entero y un puntero.

Posteriormente se han creado tres constructores, siendo el primero el principal para poder inicializar los objetos, seguido de este se encuentra un segundo constructor que asigna valores por defecto si el usuario sólo proporciona la partícula, y finalmente el tercer constructor que asigna valores por defecto si estos no han sido asignados con anterioridad.

```
EmmiterConfiguration(Solid* particle) :
    numParticles(25), emissionPeriod(10), particle(particle) {}
```

Este fragmento corresponde al segundo constructor, que asigna por defecto un máximo de 25 partículas y un periodo de emisión de 10 milisegundos.

```
EmmiterConfiguration() :
    numParticles(500), emissionPeriod(500), particle(NULL) {}
```

El fragmento superior es el tercer constructor que asigna un máximo de 500 partículas, un período de emisión de 500 milisegundos y no tiene una partícula asociada.

Además, se han creado los debidos “getters” y “setters”.

Clase Emmitter

Esta clase hereda el método *Solid* e implementa el método *Render,* a su vez tiene una propiedad de la clase *EmmiterConfigurator* donde se guarda la configuración del emisor.

Para ello se ha declarado un vector de punteros a *Solid* donde se almacenan las referencias a las partículas generadas.

```
void Emmiter::Render()
{
	for (auto& particle : particles)
	{
		particle->Render();
	}
}

void Emmiter::SetParticle(Solid* particle)
{
	config.setParticle(particle);
}
```

El método *Update()* verifica si ha transcurrido el intervalo de tiempo necesario desde la última emisión, crea una nueva partícula, modifica sus propiedades y la añade al conjunto de partículas para su renderizado en la siguiente iteración. Repite este proceso hasta que termine de generar el número de partículas definido en la clase *Emmiterconfiguration*.

```
milliseconds currentTime=duration_cast<milliseconds>(system_clock::now().time_since_epoch());

	if ((currentTime.count() - initialMilliseconds.count()) - this->lastUpdateTime > this->config.getEmissionPeriod())
```

El fragmento de código superior certifica si ha pasado el tiempo establecido desde la última emisión.

Frecuencia de emisión

La clase *Emmitter* incluye las propiedades *initialMilliseconds* y *lastUpdateTime* que se encargan de almacenar el el tiempo de inicio de ejecución del emisor y de registrar el tiempo de cada emisión de una nueva partícula.

El método *Update()* emite las nuevas partículas y mide si ya ha pasado el intervalo de emisión indicado en la configuración para generar una nueva partícula. Al final del método *Update()* se actualiza el valor de la propiedad *lastUpdateTime*.

```
	if ((currentTime.count() - initialMilliseconds.count()) - this->lastUpdateTime > this->config.getEmissionPeriod())
```

```
{
	if(particles.size() < config.getParticleNum())
	{
		Solid* newParticle = config.getParticle()->Clone();

		newParticle->SetCoordinates(Vector3D(0, 0 , -2));
		newParticle->SetColor(randColor());
		newParticle->SetOrientation(randVecBinary());
		newParticle->SetOrientationSpeed(randVecBinary());
		newParticle->SetSpeed(particleSpeeds[current]);

		AddParticle(newParticle);
		current++;

	}
	lastUpdateTime = currentTime.count() - initialMilliseconds.count();
}
```

El *if* dentro de este fragmento de código se encarga de la creación de una nueva partícula una vez el tiempo establecido haya pasado. Mientras el *lasUpdateTime* se actualiza y esto sigue pasando hasta llegar al número máximo de partículas.

Patrón Prototype

El patrón prototype es muy útil para llevar a cabo el proyecto ya que, al permitir solo tener que partir de un clon de un objeto ya creado, la creación de instancias de dicho objeto serán de menos coste al llevar a cabo la ejecución del programa. Este patrón es, además, la razón de la existencia del puntero a la clase *Solid* que se mencionaba anteriormente en la elaboración de la clase *EmmitterConfiguration*.

En primera instancia, en la clase Solid se ha declarado la siguiente línea de código:

```
virtual Solid* Clone() = 0;
```

Esta permite que las clases derivadas de *Solid* hereden el método *Clone()* que está igualado a 0.

La clase *Sphere*, siendo la que se ha tomado para la creación de las partículas, toma el método *Clone()* y lo define para sí misma de manera que se cree un puntera hacia ella.

```
Sphere* Sphere::Clone()
{
	return new Sphere(*this);
}
```

Y, finalmente, el método *Clone()* se invoca en *Emmiter.cpp* en el método *Update()*, de manera que se duplican las partículas definidas previamente por su configuración y se personalizan previamente antes de añadirlas al sistema.

```
Solid* newParticle = config.getParticle()->Clone();
```

Pseudoaleatoriedad

En cuanto a la aplicación de números (pseudo)aleatorios se han aplicado en el color y en la dirección de las partículas al crearse las mismas dando así un color y una dirección distintos a cada una de las generadas gracias a los métodos creados en la clase *Emmiter.cpp*.

```
Vector3D randVecBinary()
{
	return Vector3D(rand() % 2, rand() % 2, rand() % 2);
}

Color randColor()
{
	return Color((rand() % 256) / 255.0f, (rand() % 256) / 255.0f, (rand() % 256) /
255.0f);
}
```

Colisiones

Se ha implementado la funcionalidad de colisión entre partículas y las paredes en el sistema, lo que permite una interacción más realista en las simulaciones.

```
int n = particles.size();
	for (int i = 0; i < n; ++i) {
		for (int j = i + 1; j < n; ++j) {
			if (particles[i]->CheckCollision(particles[j]))
			{
				aux = particles[i]->GetSpeed();
				particles[i]->SetSpeed(particles[j]->GetSpeed());
				particles[j]->SetSpeed(aux);
			}
		}
	}
```

Este bucle incluido en el void *Emmiter::Update()* comprueba la colisión de una partícula con otra. Cuando una de las partículas choca con otra, se cambia la dirección de la velocidad en ese eje, simulando así un rebote.

3.Conclusiones

Gracias a la realización de esta práctica el equipo ha adquirido nuevos conocimientos para la creación de partículas u otros objetos con propiedades similares en videojuegos en el ámbito de la programación. Siendo probablemente útil a futuro en caso de querer crear NPCs haciendo uso del patrón prototype, o de monstruos generados por emisores en un juego de mazmorras.

4.Bibliografía

Where developers learn, share, & build careers. Stack Overflow. (n.d.). https://stackoverflow.com/

Tutorials. cplusplus.com. (n.d.). https://cplusplus.com/