PRÁCTICA 1

EMISOR DE PARTÍCULAS

1. INTRODUCCIÓN

Este documento incluye el enunciado de la práctica 1 de la asignatura Programación Avanzada del Grado de Diseño y Desarrollo de Videojuegos. En este enunciado se plantea un problema, se incluyen ciertas mejoras en el código que es necesario implementar y se plantean los requisitos de la entrega. La práctica 1 es una práctica obligatoria que debe realizarse en grupos.

2. DESCRIPCIÓN GENERAL

En esta práctica se plantea un ejercicio que parte del código desarrollado en las clases de la asignatura. El problema que se plantea consiste en implementar un emisor de partículas, de forma que se puede integrar en el programa realizado hasta ahora como un elemento más que irá generando partículas durante la ejecución del programa.

Un emisor de partículas es una pieza fundamental en los denominados sistemas de partículas. En el ámbito de los videojuegos se entiende como sistema de partículas a la técnica mediante la que se generan múltiples partículas con unas determinadas propiedades de posición, velocidad, aceleración, tiempo de vida, color, tamaño, etc. El comportamiento de las partículas puede verse afectado, además, por fuerzas externas como la gravedad, colisiones u otro tipo de interacción con el entorno.

Los sistemas de partículas pueden usarse para simular fenómenos complejos de modelar con geometría tradicional. Por ejemplo, son especialmente útiles para representar elementos como humo, fuego, lluvia, explosiones o chipas, donde cada elemento individual (partícula) es un objeto simple, pero el conjunto crea efectos visuales dinámicos y efectivos.

Otra posible aplicación de un emisor de partículas es el de usarlo como *spawner*, es decir, como un generador de ítems según unas propiedades determinadas de generación. Podría usarse este emisor como generador de partículas en la búsqueda de la representación de estos efectos visuales, pero también como emisor de enemigos, *NPCs*, objetos del juego, proyectiles de un arma, etc. Las propiedades que pueden caracterizar al emisor son la posición, la frecuencia de generación, la duración del proceso de generación, la activación y desactivación del mecanismo y, como se ha comentado antes, las propias características de las partículas.

Con el desarrollo actual del programa resultante del desarrollo guiado en clase tenemos ya las piezas fundamentales para implementar un emisor de partículas. También contamos con varios candidatos a partículas que, de hecho, heredan de la misma clase padre (*Cube*, *Sphere*, *Teapot*, *Cuboid*, *Cylinder* y *Toroid* que heredan de *Solid*).

3. PLANTEAMIENTO

Se descompone a continuación el problema en clases o funcionalidades que pueden resolverse por separado.

3.1. CLASE EMMITERCONFIGURATION

Es necesario implementar una clase que únicamente usaremos para gestionar la configuración que queremos darle al emisor y a las partículas. Para ello simplemente es necesario añadir una clase llamada *EmmiterConfiguration* en la que se incluirán, como mínimo, las siguientes propiedades:

- Número de partículas: número máximo de partículas que emitirá el emisor. Será de tipo entero.
- Periodo de emisión: intervalo de tiempo que espera el emisor entre la creación de una partícula y la siguiente. Será de tipo entero y vendrá en milisegundos.
- Partícula: se trata de la partícula que se toma como referencia. Se trata de un puntero a Solid para así poder aceptar cubos, esferas, prismas, etc.

Será necesario incluir los métodos de acceso y un constructor que acepte los tres argumentos. Si se desea, se pueden añadir más propiedades para cualquier aspecto de la configuración del emisor que se desee implementar.

3.2. CLASE EMMITER

Se trata de la propia clase del emisor. Esta clase debe heredar de Solid, contará así con las propiedades ya definidas en Solid (posición, orientación, etc.) y será necesario que implemente el método Render y que redefina el método *Update*. Esta clase tendrá una propiedad de la clase *EmmiterConfiguration* donde se guardará la configuración del emisor. También contará con un vector de punteros a Solid donde se irán almacenando las referencias a las partículas generadas (de forma similar al vector de la clase *Scene*).

¿Qué debe hacer el método Render?

Debe invocar al método *Render* de todas las partículas generadas, similar a lo que se hace en el método *Render* de la clase *Scene*.

¿Qué debe hacer el método Update?

Este método es el que se encarga de la generación de las partículas, para ello los pasos que debe resolver son:

1) Comprobar si ha pasado el intervalo de tiempo de generación desde la última emisión (ver apartado 3.3).
2) Crear una nueva partícula (ver apartado 3.4).
3) Modificar las propiedades de la nueva partícula: posición, color, velocidad, orientación, velocidad de rotación, etc. En este punto es donde entra en juego vuestra creatividad (ver apartado 3.5).
4) Añadir la nueva partícula al vector de partículas para que se renderice en la próxima iteración.
5) Repetir mientras no se hayan generado todas las partículas indicadas en la configuración.

En los siguientes apartados se incluye más información para resolver los puntos 1, 2, y 3.

3.3. GESTIONAR LA FRECUENCIA DE EMISIÓN

Para poder emitir una partícula cumpliendo el periodo de emisión es necesario poder registrar el momento en el que se ha emitido una partícula y poder medir el tiempo que ha pasado desde ese instante.

Para ello es necesario, en la clase *Emitter,* lo siguiente:

- Incluir un par de propiedades para almacenar los valores de tiempo que permiten este control:

  - *initialMilliseconds*, que almacenará el tiempo de inicio de la ejecución del emisor, y
  - *lastUpdateTime*, que registrará el tiempo de cada emisión de una nueva partícula.
- Estas propiedades son de tipo *milliseconds* y *long* respectivamente; para poder usar objetos de tipo *milliseconds* es necesario incluir la biblioteca *chrono.*
- Deben inicializarse correctamente en el constructor (ver Figura 1).
- En el método *Update*, que es el que se encarga de emitir nuevas partículas, hay que medir si ya ha pasado el intervalo de emisión indicado en la configuración para generar una nueva partícula. Esto lo hacemos con una variable para registrar el tiempo en el momento que se está ejecutando el método *Update* y con un *if* que evalúa si ya se ha pasado el tiempo definido en la configuración (ver Figura 2).
- Finalmente, al final del método *Update* debemos actualizar el valor de la propiedad *lastUpdatedTime* (ver Figura 3).

```
this->initialMilliseconds = duration_cast<milliseconds>(system_clock::now().time_since_epoch());
this->lastUpdatedTime = 0;
```

Figura 1. Inicialización de las propiedades de gestión del tiempo de emisión.

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

if ((currentTime.count() - this->initialMilliseconds.count()) - this->lastUpdatedTime > this->configuration.GetEmissionIntervalMilliseconds())
{
```

Figura 2. Control de flujo de ejecución evaluando si ya ha pasado el intervalo de emisión.

```
 }
 this->lastUpdatedTime = currentTime.count() - this->initialMilliseconds.count();
```

Figura 3. Actualización de la propiedad lastUpdatedTime al final del método Update.

Con este mecanismo, en el cuerpo del *if* podemos implementar el mecanismo de generación de la nueva partícula, modificación de sus propiedades e inserción en el vector de partículas, ya que lo estaremos haciendo conforme a la frecuencia de emisión definida en la configuración.

3.4. EL PATRÓN PROTOTYPE

El patrón *prototype* (ver [1]) es uno de los patrones de diseño creacionales que ilustran cómo crear nuevos objetos copiando una instancia existente en lugar de crearlos desde cero. Este enfoque resulta útil cuando queremos crear varias instancias de un objeto complejo que requiere configuraciones iniciales que pueden ser costosas. De este modo, en lugar de inicializar n veces un objeto nuevo, se parte de un clon de uno creado previamente y se modifica solo lo necesario.

Este patrón es el que vamos a aplicar en el emisor de partículas y es el motivo por el cual en nuestra clase de configuración incluimos una propiedad que es precisamente un puntero a una instancia de la clase Solid que es la que queremos usar como partícula.

Seguir el patrón nos permitirá que la emisión de la partícula consista en clonar la partícula dada como referencia en la configuración y repetir tantas veces como sea necesario. Las modificaciones de las partículas son las que provocarán que el resultado final sea diferente en cada caso.

Además, dado que aquí estamos aprovechando el polimorfismo que nos proporciona la jerarquía de clases con herencia, el código de nuestro método *Update* será igual para cualquier tipo de partícula que queramos usar, ya que todas ellas heredan de la clase *Solid*.

Para poder implementar este patrón, lo primero que debemos hacer es añadir en *Solid* la declaración del método *Clone*, así ya sabremos que todas las clases hijas contarán con un método para clonar objetos. Este método es el que invocaremos desde el método *Update*. El método *Clone* tendrá que ser implementado en cada clase hija y su objetivo es devolver un puntero a una nueva instancia que tenga las mismas propiedades que la actual.

El ejemplo para la clase *Cube* se muestra en Figura 4, donde se hace uso del constructor copia de la clase.

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

Figura 4. Implementación del método Clone de la clase Cube.

3.5. UTILIZAR NÚMEROS (PSEUDO)ALEATORIOS

Una forma de generar números aleatorios en C++ es usando *rand* y *srand*. La función *rand* de la biblioteca estándar genera números enteros aleatorios, y *srand* establece la semilla para la secuencia de números aleatorios. Para asegurarnos de que los números sean diferentes en cada ejecución del programa, podemos usar el tiempo actual como semilla, lo cual permite obtener resultados aleatorios en cada ejecución. Un ejemplo de uso se muestra en Figura 5.

```
srand(static_cast<unsigned int>(std::time(nullptr)));

// Generar un valor float aleatorio entre 0 y 1
float randomValue = static_cast<float>(rand()) / RAND_MAX;
```

Figura 5. Ejemplo de generación de números aleatorios en C++..

Existen múltiples posibilidades de aplicación de los números aleatorios para que modifiquéis el comportamiento de vuestras partículas y, por tanto, del sistema de partículas. Una opción básica, por ejemplo, es cambiar aleatoriamente el color de cada partícula, pero también podéis probar a modificar la velocidad o la rotación. Tampoco es necesario recurrir a los números aleatorios, podéis definir variaciones en as partículas en base a incrementos fijos o a cualquier expresión matemática que se os ocurra. Esta es la parte libre de la práctica, aunque seguramente, cuantas más variaciones probéis, más fácil será que deis con algún resultado espectacular.

4. METODOLOGÍA

La práctica se realiza en grupos. Es necesario superar esta práctica en tiempo y forma superar la parte práctica de la asignatura.

5. REQUISITOS MÍNIMOS Y CRITERIO DE EVALUACIÓN

Los requisitos mínimos que debe cumplir el trabajo son:

- El programa generado debe compilar, no puede tener errores.
- Al ejecutarse se debe ver en pantalla al menos un sistema de partículas que evolucione en el tiempo.
- El programa debe implementarse partiendo del programa desarrollado en clase.
- El programa debe cumplir las pautas descritas en este documento y seguir el paradigma de la programación orientada a objetos.

Aparte de esto, se valorará la limpieza y el orden de la memoria y del código. También se valorará la creatividad, complejidad y resultado visual obtenido del sistema de partículas.

6. ENTREGABLES

Se deben entregar, por cada grupo:

- Memoria en *pdf*. La memoria debe incluir una explicación de cómo se ha implementado la solución, no debe incluir todo el código, sino solo los fragmentos importantes que se desee explicar en la memoria.
- Archivo comprimido (*zip* o *7z*) con el código fuente de la solución de Visual Studio. Es necesario eliminar los archivos y carpetas generadas en tiempo de compilación, así como la carpeta oculta *vs* antes de generar el archivo comprimido. El archivo resultante debe tener un tamaño del orden de KB, si ocupa más es que no se han eliminado todas las carpetas que hace falta.
- Archivo comprimido con el vídeo de una ejecución en la que se vea el resultado del sistema de partículas implementado. Una opción para grabar vídeo es *OBS studio* (ver [2]).

7. RECOMENDACIONES PARA LA REDACCIÓN DE MEMORIAS DE PRÁCTICAS

Se recomienda seguir las recomendaciones de escritura de documentos técnicos en general y de memorias de trabajos prácticos en particular. Principalmente:

- Las memorias se entregan en formato pdf.
- El documento debe incluir:
    - Portada: con información sobre curso, grado, asignatura, autores y título.
    - Índice. Opcionalmente, índice de tablas, de figuras y de expresiones matemáticas.
    - Introducción: con una descripción breve de la finalidad del documento, contexto y contenido de la memoria.
    - Conclusiones: consideraciones personales sobre lo que ha supuesto la realización del trabajo.
    - Bibliografía: listado de referencias empleadas en el documento (usar referencias cruzadas para referirse a ellas en el documento).
- Usar un lenguaje basado en formas no personales.

- Incluir siempre títulos en las figuras, tablas y expresiones matemáticas (si se diera el caso), de forma que se pueda hacer referencia a ellas en cualquier parte del texto.
- Incluir paginación en el documento.
- Numerar los apartados, capítulos o secciones para facilitar la referencia a ellos si fuera necesario.
- Si se incluye código en la memoria, no incluirlo como imagen sino como texto aplicando un formato distinto. ***En este documento se ha incumplido esta norma a propósito, para evitar que el código se pueda copiar fácilmente y que seáis vosotros quienes escribáis vuestros programas.***
- Revisar a conciencia la ortografía y gramática, incluyendo los signos de puntuación.
- Justificar el texto.
- Si el documento incluye gráficas, estas deben incluir títulos y unidades de cada eje y una leyenda con la descripción de cada serie de datos. También título y opcionalmente un subtítulo.
- Se deben minimizar los espacios en blanco entre figuras o tablas, el texto debe rellenarlos y hacer referencia a las figuras o tablas cuando corresponda.

REFERENCIAS

[1] Prototype · Design Patterns Revisited · Game Programming Patterns

[2] Download | OBS