Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Centro de Computación Gráfica

Simulación de interacciones partícula-partícula y partícula-escena con sombreado y coloración

Trabajo Especial de Grado en la Lic. de Computación
Autor: José Daniel Contreras Moorle
Tutor: Esmitt Ramírez

Caracas, 20 / 02 / 2015

Índice

Resumen

A fin de exaltar el realismo observable de los sistemas de partículas y los fenómenos o cuerpos difusos que representan, se han desarrollado una gran diversidad de técnicas y procedimientos que pueden ser utilizados a fin de implementar sistemas de partículas capaces de interactuar con las escenas tridimensionales en las que se encuentran.

Debido a la gran cantidad de elementos posibles que pueden conformar una típica escena tridimensional, la mayoría de las investigaciones se han enfocado en lograr la interacción entre las partículas y cuatro elementos principales: la coloración, el sombreado, las propias partículas del sistema y los objetos tridimensionales concretos. Actualmente existen técnicas capaces de lograr un grado de interacción considerable con cada uno de los elementos mencionados, además de ser compatibles con aplicaciones que demandan el despliegue de gráficos en tiempo real gracias al uso eficiente de los recursos de los equipos en los que se ejecutan. Sin embargo, aún no se ha efectuado un estudio concreto de sistemas de partículas que combinan todas estas metodologías a fin de reaccionar a la mayor cantidad de elementos de escena posible al mismo tiempo, por lo que es requerido el análisis del grado de interacción y rendimiento alcanzable por la combinación de todas estas técnicas. Es en este punto donde se enfoca el presente trabajo de investigación, buscando ofrecer un marco de referencia para desarrolladores interesados en implementar sistemas de partículas altamente interactivos capaces de ejecutarse en aplicaciones de despliegue gráfico dependientes de cálculos en tiempo real

Palabras Clave: Sistema de partículas tridimensionales, despliegue de gráficos en tiempo real, interacción entre partículas, interacción entre partículas y escenas, iluminación de partículas.

Abstract

To exalt the observable realism of particle systems and the phenomena or diffuse bodies they represent, a variety of techniques and procedures that can be used to implement particle systems able to interact with three-dimensional scenes in where they are have been developed.

Due to the large number of possible elements that can form a typical three dimensional scene, most research has focused on achieving the interaction between the particles and four main elements: the coloring, shading, system's particles themselves and three-dimensional concrete objects. Currently there are techniques capable of achieving a considerable degree of interaction with each of the above elements, as well as being compatible with applications that require real-time graphics rendering thanks to the efficient use of computers' resources of the on which they run. However, there has not been made any particular study about particle systems that combine all these methods in order to react to as many scene's elements as possible simultaneously, so it is still required to analyze the degree of interaction and performance achievable by combining these techniques. It is at this point that the present research is focused, seeking to offer a framework for developers interested in implementing systems able to run real time applications calculations with highly interactive particles

Keywords: three-dimensional particle system, real time rendering, interaction between particles, interaction between particles and scenes, lighting particles

Agradecimientos

Quiero agradecer la dirección y el apoyo del profesor Esmitt Ramírez Jacobo, sin el cual este trabajo de investigación no se hubiese emprendido.

A mis padres y a mi hermano, que sacrificaron mucho tiempo y esfuerzo para hacer esto posible. Por enseñarme desde temprana edad el valor de la educación y perseverancia. Por escucharme cuando necesitaba un consejo y acompañarme cuando lo necesitaba. A ellos más que a nadie.

A mis amigos, a todos. Porque fueron la fuente de distracción y apoyo que me acompaño en este largo camino. A los profesores, que guiaron mis estudios y me formaron como profesional. A todos aquellos que de manera directa o indirecta, influyeron, hicieron su aporte o ayudaron en este trabajo.

Introducción

En el transcurso del tiempo las técnicas tradicionales de despliegue han probado ser considerablemente exitosas cuando trabajan con objetos de superficies claramente definidas. Sin embargo, en el mundo real existen tipos de objetos que no poseen superficies concretas y por lo tanto no pueden ser desplegadas como polígonos o inclusive superficies basadas en B-splines racionales no uniformes (NURBS). Fenómenos naturales como las nubes, humo, polvo o fuego pertenecen a esta clase de objetos de contornos difuminados.

Como respuesta a la problemática anterior surgen los sistemas de partículas, primeramente introducidos en 1983 por William T. Reeves [1], quien definió a las partículas como una entidad elemental que puede moverse y desplegarse de varias maneras, pero que en esencia es simple. Los sistemas de partículas son un conjunto coherente de estas entidades, significando que las partículas dentro del sistema deben poseer el mismo tipo de atributos y estar expuestas a las mismas fuerzas. Por ejemplo, las partículas pueden poseer diferentes coloraciones, pero sus colores deben ser calculados empleando los mismos algoritmos.

Los sistemas de partículas son esencialmente estructuras dinámicas que poseen un ciclo de vida y que pueden o no interactuar en tiempo real con escenarios cambiantes y alteraciones en sus parámetros internos. Las posibilidades de configuración de un sistema de partículas son muy extensas: el movimiento, color o la representación final en pantalla son solo unos pocos de los atributos que pueden ser modificados con libertad a fin de producir diferentes resultados visuales durante el despliegue de un sistema de partículas. Los atributos y las fases o etapas consideradas fundamentales para la mayoría de los sistemas de partículas de la actualidad son expuestos dentro del Capítulo 1.

El realismo observable de un sistema de partículas es afectado principalmente por el grado de interacción y coherencia que el mismo presenta con el entorno en el que se encuentra. A lo largo de los años se han desarrollado una gran diversidad de mecanismos a fin de lograr que las partículas de un sistema interactúen de una forma u otra con los elementos típicos que conforman a una escena tridimensional: luces, objetos modelados tradicionalmente de superficies definidas e inclusive otras partículas del mismo u otros sistemas. En el Capítulo 2 se expone la teoría asociada a la implementación propuesta de un sistema de partículas capaz de reaccionar simultáneamente a todas las facetas de una escena señaladas previamente, mientras que en el Capítulo 3 se profundiza en los algoritmos y estructuras que componen el programa desarrollado.

En el Capítulo 4 se analizan las pruebas y resultados obtenidos mediante la ejecución del sistema de partículas desarrollado, haciendo especial énfasis en estudios asociados al impacto de múltiples factores de la escena y del propio sistema de partículas sobre el desempeño, en términos de tiempo de ejecución, registrado durante su ejecución.

Finalmente, las conclusiones de la presente investigación y propuestas de trabajos futuros a desarrollar sobre la implementación modular del sistema de partículas implementado son expuestas en el Capítulo 5.

Objetivos

Objetivo general

Desarrollar un sistema de partículas altamente interactivas capaces de reaccionar dinámicamente con factores como iluminación, sombreado y colisiones dentro de una escena.

Objetivos específicos

a) Emplear técnicas para garantizar la iluminación de las partículas de forma precisa y consistente con la iluminación de los demás objetos presentes en la escena.
b) Implementar mecanismos de interacción en tiempo real entre las partículas del sistema.
c) Generar mapas de sombra tanto de los objetos tridimensionales modelados tradicionalmente, como del sistema de partículas para lograr la recepción y proyección de sombras de ambos.
d) Emplear técnicas basadas en cálculos en espacio de pantalla para lograr la colisión de partículas con objetos tridimensionales que conforman los escenarios.
e) Realizar pruebas representando diversos sistemas de partículas y comparando su rendimiento.

Capítulo 1: Marco teórico

Los sistemas de partículas son un tipo de técnica de simulación de objetos y fenómenos difusos muy empleado en sistemas computacionales. Su nombre proviene del hecho que la mayoría de los mismos están basados en los sistemas de partículas de la mecánica clásica newtoniana como observada en [2].

En el área de la computación gráfica, los sistemas de partículas tienen tres particularidades claves que los diferencian de las técnicas tradicionales de modelado, animación, despliegue y manejo en general de objetos tridimensionales. En primera instancia, los objetos son interpretados y manejados como un conjunto o nube de partículas (elementos atómicos o compuestos cuyas dimensiones son irrelevantes) que definen su volumen. Segundo, los sistemas de partículas no son entes estáticos ya que su forma y dimensiones varían en el tiempo debido a las transformaciones de las partículas que lo componen. Por último, las transformaciones dentro de un sistema de partículas se basan en procesos estocásticos por lo que un objeto simulado dentro del mismo posee atributos calculados con considerable influencia de elementos aleatorios. En las subsecciones siguientes estudiaremos de qué forma se representan estos sistemas dentro de las aplicaciones de acuerdo al esquema propuesto por [1].

1.1 Modelo básico de los sistemas de partículas

A pesar de la gran diversidad de objetos y fenómenos difusos posibles, [1] señala como todos los sistemas de partículas comparten un esquema de trabajo general basado en efectuar una serie de pasos en los que se manejan uno o varios conjuntos de partículas que representan al objeto o fenómeno a simular. El dinamismo del sistema viene dado por las etapas de generación, transformación y eliminación de las partículas del sistema, las cuales mediante cómputos procedimentales y típicamente estocásticos determinan los atributos del cuerpo difuso en cada instante de tiempo y, producto de una fase de despliegue que interpreta la información producida, dicta la apariencia visual del cuerpo difuso en cada cuadro (*frame*) de animación que posea al sistema de partículas.

Los cálculos de los sistemas de partículas pueden dividirse en dos fases secuenciales: la fase de simulación compuesta por las etapas de generación, dinámica y extinción de partículas, y la fase de despliegue del sistema.

A continuación se desarrollan definiciones bases de las etapas y componentes clásicos que componen a la mayoría de los sistema de partículas, los cuales son puntos pivótales para comprender las implementaciones y etapas más especializadas explicadas en capítulos posteriores.

1.1.1 Generación

Típicamente las partículas son generadas dentro del sistema mediante la inserción de cantidades constantes en cada cuadro de animación o a través el uso de procesos aleatorios interrelacionados. El proceso central de la generación se encarga de determinar la cantidad exacta de nuevas partículas que serán introducidas en cada instante de tiempo y, consecuentemente, en cada cuadro de animación generado. Este proceso es particularmente relevante debido a que influencia fuertemente la densidad final del objeto difuso simulado.

La implementación básica de la inserción basada en cálculos aleatorios propone dos alternativas para determinar la cantidad de nuevas partículas. El primer método consiste en, dado un valor de *MediaPart* y *VarPart* predeterminada, calcular el número de nuevas partículas en el cuadro f mediante la ecuación:

$$NPart = MediaPart + Rand\ () \times VarPart$$

donde Rand es un procedimiento que retorna un número aleatorio uniformemente distribuido entre [–1.0, 1.0], *MediaPart* representa la media del número de nuevas partículas, y *VarPart* representa a la varianza.

En el segundo método, el número de nuevas partículas es calculado considerando el tamaño que el sistema de partículas ocupa en la pantalla. Al igual que el método anterior el diseñador provee de la respectiva varianza del sistema pero en este caso la media se establece *por unidad de espacio de pantalla*. El sistema de partículas puede entonces determinar los parámetros de la vista actual en cada instante de tiempo, calcular el espacio de pantalla que el mismo ocupa, y establecer el número de partículas nuevas de forma acorde. La ecuación propuesta es:

$$NParts = (MeanParts + Rand\ (\ ) \times VarPart) \times AreaPantalla$$

donde *MeanParts* es la media por área de pantalla, *VarParts* es su varianza, y *AreaPantalla* el área que ocupa el sistema de partículas en la pantalla. Este método controla el nivel de detalle del sistema de partículas y, por lo tanto, el tiempo necesario para desplegar su imagen. Por ejemplo, con este método no hay necesidad de generar 100.000 partículas en un objeto que solo cubre 4 píxeles de la imagen final.

Ambas metodologías explicadas anteriormente pueden ser refinadas mediante la adición de un grado adicional de dinamismo producto de variar el número medio de partículas generadas por cuadro de animación, esto es típicamente efectuado mediante el uso de una simple ecuación lineal:

$$MediaPart = MediaPartInicial + DeltaMediaPart\ (f - f_0)$$

donde f es el cuadro actual y f_0 el primer cuadro durante el cual el sistema de partículas inicio su respectiva generación. *MediaPartInicial* representa la media del número de partículas en cada cuadro, y *DeltaMediaPart* indica la razón o proporción de cambio. El valor de *VarPart* es constante en todos los cuadros. Es notable señalar que la ecuación lineal propuesta puede ser fácilmente cuadrática, cúbica o estocástica.

Es importante señalar que es posible que los sistemas de partícula establezcan un número máximo de partículas de forma global, por lo que al alcanzarse este valor el proceso de generación se detiene hasta que la cantidad de partículas del sistema sufra un decremento.

1.1.2 Atributos

Para cada nueva partícula generada, la mayoría de los sistemas deben establecer los valores para los siguientes atributos:

1. Posición inicial
2. Tamaño, color, transparencia y velocidad inicial (compuesta de vector dirección y modulo de rapidez)
3. Textura
4. Forma
5. Tiempo de vida

Varios parámetros controlan la posición inicial de cada una de las partículas que son generadas. Un sistema de partículas tiene una posición tridimensional que define su punto de origen y, opcionalmente, dos ángulos de rotación en torno al mismo definen la orientación del sistema. Las nuevas partículas pueden ser generadas exclusivamente en el punto origen o emplear algoritmos de distribución inicial como por ejemplo los basados en forma de generación (también conocidos como emisores de partículas), los cuales establecen una región tridimensional, con pivote en el origen del sistema, dentro de la cual las nuevas partículas son aleatoriamente colocadas durante su creación. La Figura 1 muestra una forma de generación esférica.

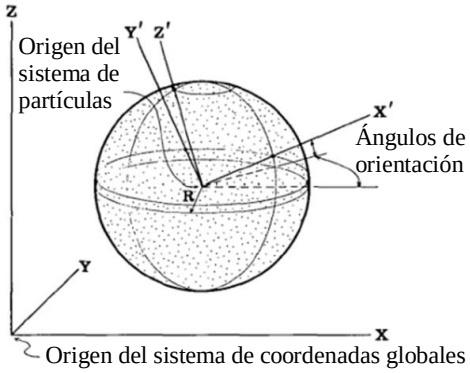


Figura 1. Sistema de partículas típico con forma de generación esférica

El vector de dirección de la velocidad inicial de cada nueva partícula es frecuentemente establecido de acuerdo a dos posibles alternativas: generar el vector de forma pseudo-aleatoria o, en caso de que el sistema haga uso de una forma de generación, el vector de velocidad inicial puede estar intrínsecamente asociado a la forma empleada. Por ejemplo, en el caso de una forma de generación esférica las partículas pueden tender a alejarse del origen del sistema. En el caso de formas de generación bidimensionales como círculos y rectángulos, es típico que las nuevas partículas

sean generadas en el plano descrito por la forma de generación y sus vectores de velocidad sean variaciones de vectores tangentes a dicho plano perturbados por ángulos de eyección aleatorios. La Figura 2 ilustra este último ejemplo.

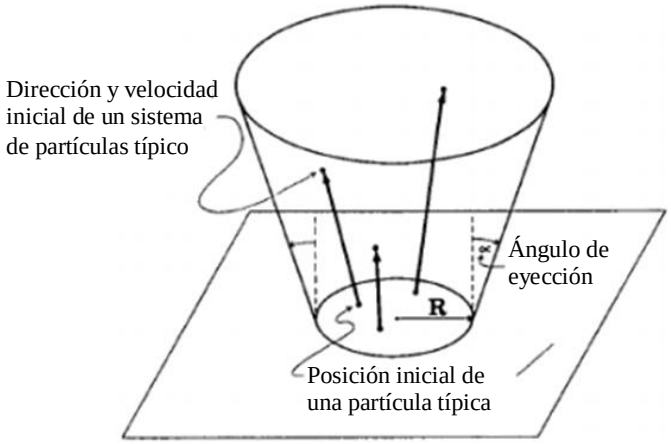


Figura 2. Forma de un sistema de partículas para modelar una explosión

De forma complementaria, el módulo del vector de la velocidad inicial de cada partícula, definido como el escalar rapidez, es típicamente calculado mediante la ecuación:

$$RapidezInicial = MediaRapidez + Rand\ (\ ) \times VarRapidez$$

donde *MediaRapidez* y *VarRapidez* representan a la rapidez media y su respectiva varianza dentro del sistema de partículas. Al igual que en las sub-secciones anteriores de este documento, *Rand* es un procedimiento que devuelve un número aleatorio uniformemente distribuido entre [–1.0, 1.0].

El atributo de textura es opcional, y en los casos en los que se emplea define la imagen que será expuesta en cada partícula del sistema. Para determinar el color, la transparencia, el tamaño y tiempo de vida inicial de cada nueva partícula se definen valores promedios y se establece la desviación máxima de dichos valores que son permitidos dentro del sistema. Las respectivas ecuaciones son similares a la dada arriba para la rapidez inicial de las partículas.

Finalmente, los sistemas de partículas tienden a definir la forma de las partículas que generan: en la mayoría de los casos dicha forma es un punto o un plano rectangular bidimensional, pero la misma puede ser cualquier forma bidimensional geométrica e inclusive objetos tridimensionales arbitrarios.

El número de posibles atributos iniciales controlables y sus respectivas variaciones son virtualmente infinitos ya que dependen de las especificidades de cada sistema de partículas creado. Los presentados en esta sección constituyen los que son considerados estándar pero es imperante señalar que los mismos no son ni obligatorios ni limitantes para todo sistema de partículas a diseñar.

1.1.3 Dinámica

Una vez generadas, es común que las partículas se muevan en un espacio tridimensional y también cambien de color, transparencia y tamaño a través del tiempo de la simulación.

Mover a una partícula entre un cuadro de animación y otro consiste simplemente en añadir su vector de velocidad a su vector de posición actual. Para añadir más complejidad, el sistema de partículas puede usar un factor de aceleración para modificar entre cuadros a la velocidad inicialmente establecida para cada partícula. Con este simple parámetro de aceleración se pueden simular la interacción de fuerzas complejas externas al sistema como el viento y la gravedad y causar que las partículas produzcan trayectorias arbitrarias en vez de las líneas rectas obtenidas mediante el uso exclusivo e inalterado del vector de velocidad inicial. Implementaciones más complejas pueden inclusive considerar a las propias partículas del sistema y la geometría tradicional de la escena durante el transcurso de su simulación, modificando sus trayectorias de acuerdo a la presencia y características de estos tipos de elementos en tiempo real.

El color, la transparencia y el tamaño de cada partícula del sistema también pueden cambiar en el tiempo frecuentemente mediante el uso de algoritmos de interpolación que, dados o generados estocásticamente los valores finales para cada uno de estos atributos de forma global al sistema o por cada partícula, determinan valores intermedios por partícula para cada instante de tiempo considerando el tiempo de vida y/o distancia del origen del sistema de la partícula. Es frecuente que el modelo de interpolación empleado sea lineal, exponencial o polinómico.

1.1.4 Extinción

Típicamente cuando una partícula es generada, a la misma se le asigna un valor de tiempo de vida medido en segundos o en cuadros de animación. Al inicio de cada cuadro el tiempo de vida de cada partícula es reducido y una vez que el mismo alcanza un valor igual o menor a cero la partícula es eliminada del sistema.

Existen otros mecanismos de eliminación frecuentemente empleados en conjunto al anterior: las partículas pueden ser eliminadas si se determina que no contribuyen en nada a la imagen final generada. Un criterio para determinar si una partícula contribuye o no puede ser discernir si su intensidad, calculada por su color y transparencia, toma un valor inferior a un umbral predeterminado y eliminarla en dichos casos. Otro criterio empleado es la distancia, en el cual toda partícula que supere cierta distancia de una posición predeterminada o sistema al que está supeditado es inmediatamente eliminada. Este último criterio también puede ser empleado para recortar del sistema a cualquier partícula que se salga de una región de interés predeterminada.

1.1.5 Despliegue

Una vez que la posición y parámetros de apariencia de todas las partículas han sido calculados para un determinado cuadro, un algoritmo de despliegue se encarga de producir la imagen final. Los procesos de despliegue de partículas pueden llegar a poseer un grado de complejidad comparable con el despliegue de objetos compuestos de primitivas gráficas más comunes, como los polígonos y superficies curvas. Sin embargo, existen ciertas particularidades de los sistemas de partículas que deben ser consideradas en esta fase como: la oclusión de partículas, la transparencia y la proyección de sombras sobre el escenario y otras partículas. Adicionalmente, las partículas pueden tener que coexistir con objetos concretos modelados con primitivas geométricas tradicionales, e inclusive ser intersectadas por los mismos.

Debido a la gran densidad de elementos con los que frecuentemente trabajan, los sistemas de partículas que no requieren de una elevada fidelidad visual tienden a efectuar reducciones de las capacidades mencionadas previamente: la posibilidad de intersección con objetos concretos tradicionales es removida mediante la separación de los procesos de despliegue de las partículas y los objetos modelados con otras técnicas; una etapa de composición se encarga de combinar las imágenes resultantes y producir la imagen final.

Otra simplificación frecuentemente efectuada consiste en desplegar cada partícula como una fuente puntual de luz. Con este procedimiento el determinar oclusión mutua entre partículas ya no es un inconveniente ya que cada partícula añade un poco de luz a los píxeles que cubre, y en consecuencia, una partícula detrás de otra no es ocluida sino que en vez de eso añade más luz a los píxeles cubiertos. La cantidad de luz añadida y su color dependen de la transparencia y color de cada partícula y, en implementaciones un poco más complejas, también se considera la distancia entre la partícula y el punto de vista de la escena. Los píxeles cubiertos por cada partícula son determinados por su forma, tamaño y deformación producto del punto de visión.

Con las simplificaciones desarrolladas no es necesario ordenar las partículas antes de desplegarlas por lo que estas son desplegadas en el *frame buffer* en el orden en el que son generadas (despliegue conmutativo de partículas). Consecuentemente, la proyección de sombras es excluida de los sistemas de partículas que emplean estas técnicas debido a que las partículas que producen no reflejan sino emiten luz.

En los casos donde el realismo visual es imperativo y las simplificaciones visuales mencionadas no son aceptables para el resultado deseado, los desarrolladores deben hacer especial énfasis en la eficiencia de cómputo de los cálculos de coloración y sombreado de las partículas debido a que la gran densidad de elementos típicamente trabajados por estos sistemas implica una equiparable cantidad de fragmentos a procesar durante la fase de despliegue, esto es exacerbado al considerar que la transparencia de las partículas produce que inclusive los fragmentos de las partículas ocluidas por otras desde el punto de vista de la escena también deban ser procesadas.

1.1.6 Jerarquía

En el caso que las partículas de un sistema sean a su vez otro sistema, se forma lo que se denomina jerarquía de sistemas de partículas. Un sistema que posee sistemas subyacentes es denominado sistema de partícula padre y

cuando el mismo es transformado, también lo son todos sus sistemas descendientes y sus respectivas partículas y/o sistemas que contiene. La media y varianza del color del sistema de partícula padre es típicamente empleado para determinar la media y varianza de los sistemas de partículas hijos usando las ecuaciones desarrolladas en las sub-secciones anteriores; otros parámetros del padre afectan de forma similar a aquellos de sus hijos. El número de nuevos sistemas de partículas generados por cuadro de animación es determinado por el número de nuevas partículas por cuadro del sistema padre. Tradicionalmente, la estructura de datos usada para representar este tipo de jerarquía es un árbol.

Una jerarquía puede ser usada para ejercer control global sobre objetos difusos complejos que sean compuestos por muchos sistemas de partículas. Por ejemplo, una nube puede estar compuesta por muchos sistemas de partículas, cada uno representado una región de partículas de agua. Un sistema de partículas padre podría agrupar todas estas regiones y controlar el movimiento y apariencia general de la nube producto de la influencia del viento y terreno.

1.2 Tipos de sistemas de partículas

Con el paso del tiempo ha surgido una gran diversidad de modelos de sistemas de partículas diferentes que han tomado como base al modelo explicado en la sección anterior. A fin de simplificar y facilitar el entendimiento de futuras secciones en la presente se explican dos modelos clasificatorios de los sistemas de partículas: uno que considera la estructura empleada para implementar a los mismos, y otro que toma como base el comportamiento de dicho sistemas a fin de clasificarlos.

1.2.1 Tipos de sistemas de partículas según su implementación

El tiempo de cómputo y el almacenamiento requerido que los sistemas de partículas con gran densidad de elementos deben efectuar para generar nuevas partículas y transformar los atributos de todas las partículas supervisadas por el mismo, han provocado que gran parte del foco de investigación en el área de simulación de sistemas de partículas por computador haya sido encausado en el desarrollo de modelos eficientes en el uso de estos recursos. Debido a que los sistemas de partículas modernos son frecuentemente empleados en secuencias animadas en tiempo real, la velocidad de procesamiento de los mismos es un factor determinante para su éxito en las aplicaciones de actualidad. Considerando esto, dos alternativas de implementación han ido siendo investigadas y desarrolladas con el paso del tiempo: los sistemas de partículas basados en la unidad central de procesamiento del computador (CPU) y los sistemas de partículas basados en la unidad gráfica de procesamiento (GPU).

Los sistemas de partículas basados en la CPU implementan la generación, dinámica, extinción y organización de las partículas en la CPU del sistema, dejando a la GPU del mismo exclusivamente para la fase de despliegue. Los sistemas de partículas implementados de esta forma, sin importar que tan rápido sean capaces de simular las partículas en la CPU, deben de ser diseñados considerando dos graves cuellos de botellas intrínsecos a este tipo de modelos: en primer lugar, todos los atributos de todas las partículas deben ser transferidos a la GPU para su despliegue, lo cual en caso de trabajar con elevadas cantidades de partículas, puede llegar a sobrecargar el ancho de banda que comunica a ambas unidades de procesamiento. En segundo lugar, en caso de que el proceso de despliegue no sea conmutativo y se requiera organizar las partículas de atrás hacia adelante, la carga de cómputo de la CPU es casi duplicada debido al manejo de todas las partículas nuevamente luego de la fase de simulación.

Considerando estos factores adversos, es común que los sistemas de partículas basados en las CPU modernos solo sean capaces de simular fluidamente entre 1.000 y 100.000 partículas [3].

Los sistemas de partículas basados en la GPU implementan casi todas las fases de un sistema de partículas en la unidad de procesamiento grafico del sistema, liberando a la CPU de toda la carga asociada a la simulación del sistema de partículas, y al mismo tiempo nulificando casi completamente los cuellos de botella (*bottleneck*) presentes en su homólogo basado en la CPU al integrar las fases de simulación y despliegue dentro de un mismo componente de hardware.

Este tipo de técnica explota la gran cantidad de núcleos de procesamiento para paralelizar el procesamiento del sistema de partículas. Las implementaciones más frecuentes emplean texturas para almacenar, acceder y modificar todos los atributos de las partículas; adicionalmente, se efectúan los respectivos cómputos de la fase simulación en el píxel shader del sistema. El principal problema de este tipo de modelo, el cual comparte con su equivalente basado en la CPU, está en los cálculos de iluminación de las partículas: la iluminación de objetos concretos consiste en una mezcla de iluminación difusa, especular, sombras e iluminación global; las partículas que componen objetos difusos simulados mediante sistemas de partículas no poseen el vector normal necesario para calcular la iluminación difusa

y especular, por lo que los sistemas de este tipo deben implementar métodos para simular o suplir dicho vector a fin de producir una iluminación consistente para todas las partículas.

En hardware moderno, es frecuente que los sistemas de partículas basados en la GPU sean capaces de simular en tiempo real y de forma fluida más de 1 millón de partículas simultáneamente.

1.2.2 Tipos de sistemas de partículas según su comportamiento

Los sistemas de partículas pueden ser clasificados según la metodología que emplean para calcular los valores de los atributos de sus partículas y el tipo de interacción que poseen con el entorno en el que se encuentran, específicamente, de acuerdo a si las partículas generadas por el sistema interactúan o no con su entorno de forma dinámica.

Los sistemas de partículas sin estado o sistemas de partículas paramétricos son los sistemas de partículas que computan los valores de los atributos de cada partícula desde su nacimiento hasta su muerte mediante el uso de expresiones de forma cerrada (conjunto de funciones finitas claramente definidas), las cuales reciben como parámetros a el tiempo actual y valores iniciales de cada atributo para calcular el valor de los mismos en cada instante de tiempo en el que se solicitan. Debido a que las funciones que se emplean en estos cálculos son estáticas, las mismas son incapaces de adaptarse a modificaciones en el entorno en el que se encuentran las partículas y por ende solo son capaces de generar trayectorias fijas que ignoran cambios de escenario dinámicos y presencia de otras partículas.

Como ventaja, no solo no necesitan almacenar los valores intermedios de los atributos de cada partícula sino que también la naturaleza rígida y predefinida de los cálculos de este tipo de sistema de partículas permite que los mismos sean considerablemente veloces y en consecuencia este tipo de sistema sea la elección por defecto empleada para simular fenómenos difusos exclusivamente afectados por fuerzas externas predecibles como la gravedad [4].

Los sistemas de partículas con estado son los sistemas de partículas que emplean operaciones sucesivamente sobre los atributos de sus partículas para calcular sus valores en cada instante de tiempo, usando como parámetros a los valores previamente calculados de dichos atributos y, en caso que la implementación requiera interacción dinámica con su entorno, descripciones de la escena en la que se encuentran. Debido a que este tipo de sistemas efectúa cálculos sucesivos, sus implementaciones son muy flexibles y permiten extenderse mediante el cálculo de colisiones con otros elementos en la escena como otras partículas u objetos concretos modelados de forma tradicional, además de permitir la interacción de fuerzas externas calculadas de forma estocástica como el viento y turbulencia.

Como desventaja, las implementaciones más frecuentes de este tipo de sistemas deben efectuar fases adicionales de cómputo por cada partícula a fin de calcular las respectivas interacciones dinámicas con su entorno, lo cual en sistemas de partículas densamente poblados puede producir una considerable carga de cómputo adicional.

Es importante señalar que ambos tipos de comportamiento de sistemas de partículas pueden incluirse en un mismo sistema, siendo una práctica frecuentemente realizada a fin de optimizar los sistemas de partículas al aprovechar las ventajas circunstanciales de los sistemas con y sin estado. Generalmente los híbridos implementan el cálculo basado en estado para los atributos que interactúan con la escena como la velocidad y posición de las partículas, dejando los cálculos sin estado para atributos que cambian sin influencia de la escena como el color, transparencia y tamaño de las partículas.

1.3 Algoritmos básicos

Con el paso del tiempo han surgido una gran cantidad de implementaciones notables de sistemas de partículas, permitiendo la creación de modelos genéricos que han servido como base para la creación de sistemas subsiguientes. El presente documento ilustra cuatro de estos modelos que funcionan como base para la creación de sistemas de partículas de las categorías estudiadas en la sección anterior.

1.3.1 Sistema de partículas con estado basado en la CPU

Este tipo de sistema implementa la simulación de partículas en la CPU y su respectivo despliegue en la GPU, almacenando los valores de los atributos de todas las partículas del sistema en la memoria RAM del equipo o tarjeta de video. Este tipo de implementación es la que sigue de forma más pura al modelo básico explicado en las secciones anteriores, ejecutando las siguientes fases de forma secuencial [5]:

1. Extinción
2. Generación
3. Actualización de velocidades

4. Actualización de posiciones
5. Ordenamiento (opcional)
6. Transferencia de atributos actuales de las partículas a la GPU
7. Despliegue

Las primeras cinco fases expuestas son ejecutadas de forma local en el CPU. Durante estas fases los algoritmos encargados acceden y modifican a una predeterminada estructura de almacenamiento que posee los valores de los atributos de todas las partículas manipuladas por el sistema. Estas estructuras de almacenamiento eran implementadas exclusivamente en la memoria RAM del equipo, pero con el pasar del tiempo surgieron implementaciones hibridas que emplean la memoria de la tarjeta de video y sus VBO (*Vertex Buffer Object*) para almacenar y acceder a los atributos de las partículas.

Independientemente de la estructura empleada para almacenar la información de las partículas es posible que deba efectuarse una fase de ordenamiento de las mismas si las especificidades del sistema así lo requieren. Los sistemas de partículas que manejan partículas con transparencia y en donde el orden en el que las partículas se ocluyen durante el despliegue es relevante deben implementar esta fase debido a que no pueden depender del búfer de profundidad del sistema debido a que el mismo es deshabilitado para que el efecto de transparencia pueda emplearse. En la Figura 3 se observa la diferencia entre el despliegue de un mismo sistema de partículas con y sin fase de ordenamiento [6].

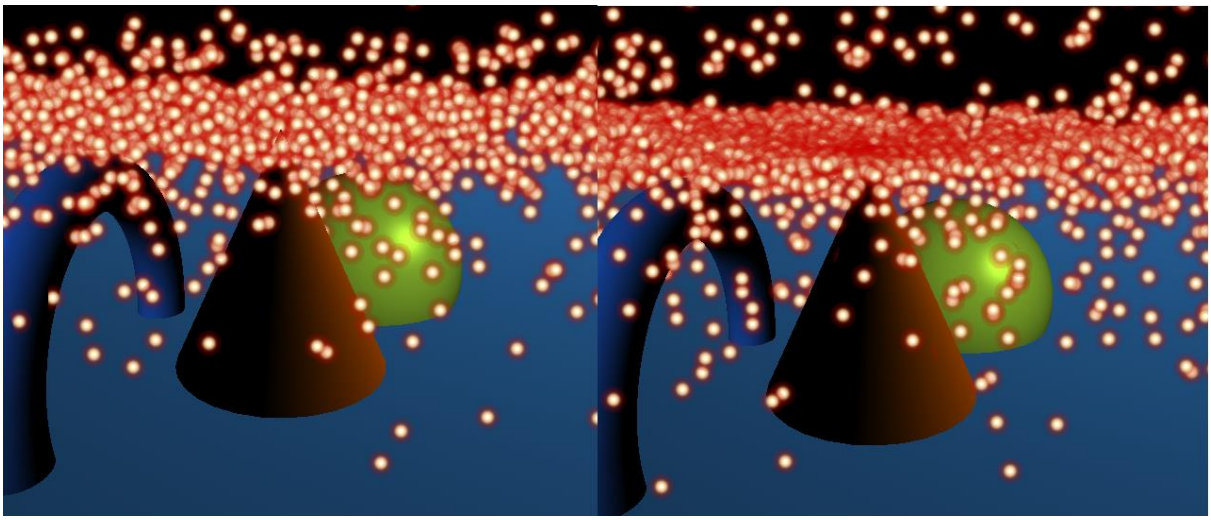


Figura 3. Comparación entre sistema de partículas sin fase de ordenamiento (izquierda) y con fase de ordenamiento (derecha)

El proceso de ordenamiento consiste en reorganizar las posiciones de las partículas dentro de la estructura de memoria que las almacena de acuerdo a un criterio predeterminado. Los algoritmos de ordenamiento más frecuentemente empleados son aquellos basados en profundidad (*depth sort*), los cuales reorganizan los contenidos de la estructura de memoria que contiene las posiciones de las partículas a fin de que aquellas más alejadas del punto de vista de la escena se desplieguen de primero y sean eventualmente ocluidas por aquellas más cercanas al punto de vista.

Una vez que todos los atributos de las partículas han sido actualizados, se procede a transferir dicha información a la GPU para su despliegue. En el esquema de trabajo de las alternativas que emplean VBO se debe transferir la instancia de dicha estructura con la información de las partículas desde el GPU a la CPU, modificar la información que contiene y finalmente volverlo a transferir a la GPU para su respectivo despliegue. Esta alternativa tiene el problema que disminuye el rendimiento del sistema al duplicar el tiempo empleado para transferencia de datos, por lo que es más empleada en aplicaciones que no demandan el despliegue de gráficos en tiempo real.

1.3.2 Sistema de partículas con estado basado en la GPU

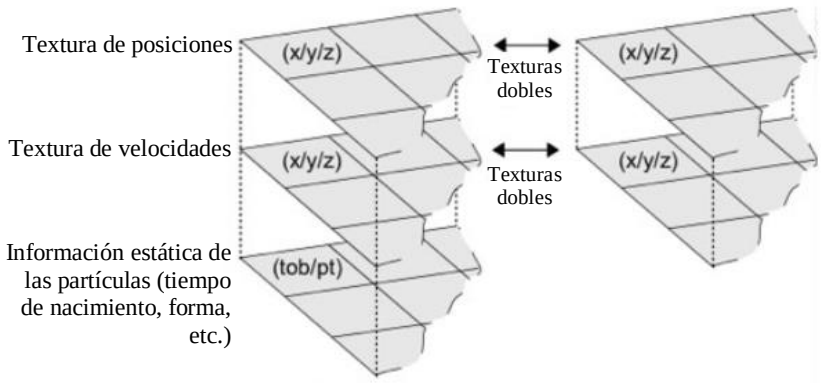
Este tipo de sistema de partículas implementa casi todas las fases en la GPU y almacena explícitamente la información de todas las partículas en una o varias estructuras de memoria localizadas en dicho dispositivo, efectuando operaciones iterativas en paralelo sobre las mismas para determinar sus valores en cada cuadro de

animación. Las fases de este tipo de sistemas son exactamente las mismas y en el mismo orden que las del tipo anterior, sin embargo, su implementación cambia notablemente debido al dispositivo en el que se ejecutan. En la actualidad existen dos tendencias para la simulación de sistema de partículas dentro de la GPU: una basada en cálculos efectuados dentro del procesador de geometría o el de fragmentos y una basada en cómputos de propósito general efectuados en la GPU (GPGPU).

En la tendencia basada en cálculos efectuados dentro del procesador de geometría o de fragmentos, se aprovecha el flujo de trabajo tradicional de las tarjetas de video para efectuar la creación y simulación de las partículas dentro de este. En estos sistemas de partículas la cantidad de elementos es estática durante la simulación y la posición y velocidad de todas las partículas son almacenadas en texturas con canales RGBA, donde los tres primeros canales son empleados para almacenar las respectivas posiciones y velocidades, dejando el canal alfa para almacenar atributos como el tiempo de vida restante, el tamaño u otro atributo especializado de las partículas [7].

La mayoría de los cálculos de simulación como la actualización de posiciones y velocidades de este tipo de sistema se efectúan en el procesador de fragmento o en el procesador de geometría del GPU, dejando al procesador de vértices para operaciones intermediarias en implementaciones especializadas. Los programas ejecutados en el procesador de fragmento acceden a los atributos de cada partícula mediante coordenadas de textura, factor que le permite a cada instancia acceder a una coordenada diferente y trabajar de forma paralela con las partículas asociadas.

La principal problemática que inicialmente presentaban este tipo de sistemas es que el hardware de la GPU no está diseñado para efectuar operaciones de lectura-modificación-escritura de forma concurrente sobre una misma estructura de datos; la solución más globalizada es el uso de algoritmos de "ping-pong" en los cuales se lee la información de las partículas de la textura original y se procede a escribir la información actualizada en una textura secundaria. Al finalizar la actualización de los atributos de todas las partículas se intercambian los contenidos de la textura original con los de la textura secundaria a fin de que la misma tenga los valores actualizados para el siguiente cuadro de animación. La Figura 4 ilustra el intercambio entre las texturas primarias y secundarias.



Las texturas iniciales (izquierda) se intercambian con las secundarias actualizadas (derecha)

Figura 4. Ilustración de algoritmo de ping-pong empleado por la GPU [7]

La fase de generación de nuevas partículas se efectúa en la CPU, el cual lee de la GPU los índices disponibles, ya sea mediante una textura especializada que indique cuales partículas están inactivas o un VBO, y procede a emplear algoritmos de asignación rápida para determinar las posiciones en memoria de una nueva cantidad de partículas calculada con las metodologías explicadas en secciones anteriores [4].

La extinción de las partículas es frecuentemente implementada tanto en la GPU como en la CPU y es efectuada evaluando el tiempo de vida restante de cada partícula el cual es frecuentemente almacenado en el canal alfa de la textura que contiene la posición de todas las partículas del sistema. Una vez que el tiempo de vida de la partícula llega a cero esta es reemplazada por una nueva partícula generada durante la fase de generación del sistema.

La fase opcional de ordenamiento se diferencia de la de su homólogo en la CPU, en que en esta versión se explota el paralelismo para ejecutar algoritmos de ordenamiento capaces de reorganizar los contenidos de las texturas rápidamente a pesar de tener que ignorar si las mismas ya se encontraban en secuencia debido a que el hardware paralelo de la GPU inhibe el chequeo eficiente de este tipo de verificaciones secuenciales.

Finalmente, las posiciones, colores y tamaños almacenados en las respectivas texturas son empleadas para desplegar primitivas de forma tradicional. En caso de usar primitivas cuadradas (*quads*) alineados al punto de vista de la escena (*billboards*) el algoritmo debe efectuar rotaciones bidimensionales para efectuar la alineación y generar las coordenadas de textura de los mismos.

En la tendencia que hace uso de GPGPU, los sistemas de partículas emplean esta metodología de programación para aprovechar el paralelismo del GPU de forma independiente del flujo de trabajo tradicional de las tarjetas de video, permitiendo flexibilidad y libertad de desarrollo carente en la tendencia previa. Al igual que los sistemas de partículas basados en procesadores de fragmentos o geometría, estos sistemas de partículas son capaces de emplear texturas para almacenar los estados de las partículas, sin embargo, también pueden optar por usar arreglos de memoria tradicionales para añadir y modificar partículas fácilmente [8].

Independientemente de la forma en que los datos hayan sido almacenados, las partículas pueden ser actualizadas y ordenadas con las mismas metodologías referidas en la tendencia anterior. Sin embargo, gracias al paradigma de GPGPU el desarrollador puede optar por emplear estructuras de memoria de tamaño dinámico y proceder a generar y/o destruir las partículas directamente dentro de los programas en la GPU, evitando de esta forma que sea necesario transferir entre la CPU y la GPU información alguna sobre las partículas y sus estados.

En la actualidad existe un gran diversidad de interfaces de programación de aplicaciones que permiten desarrollar bajo esta metodología, entre las que destacan: CUDA, DirectCompute, Compute Shader y OpenCL [15]. La principal diferencia entre estas alternativas radica en que mientras DirectCompute, Compute Shader y CUDA están diseñados para explotar la capacidad de paralelismo de la GPU, OpenCL fue creado para trabajar con una gama de dispositivos más heterogéneos como CPUs multi-núcleos, GPUs, Procesadores de Señal Digitales (DSP), etc. [16]. Es notable señalar que de las alternativas expuestas previamente, DirectCompute y Compute Shader resaltan debido a que emplean lenguajes de programación similares a los que usan los programas ejecutados en el flujo tradicional de despliegue de las tarjetas de video (HLSL y GLSL respectivamente), por los que los desarrolladores que previamente habían desarrollado programas en este entorno pueden emplear estas APIs con relativa naturalidad.

1.3.3 Sistema de partículas sin estado basado en la GPU

Este tipo de sistema de partículas paramétrico implementa casi todas las fases de simulación en la GPU y almacena únicamente los valores iniciales de los atributos de todas las partículas texturas o VBOs contenidos en la memoria de dicho dispositivo, calculando los valores de los atributos de todas las partículas en cada cuadro de animación mediante expresiones de forma cerrada que solo consideran los valores iniciales y descripciones estáticas del escenario [9]. El esquema de trabajo general de este tipo de sistemas se describe en la Figura 5.

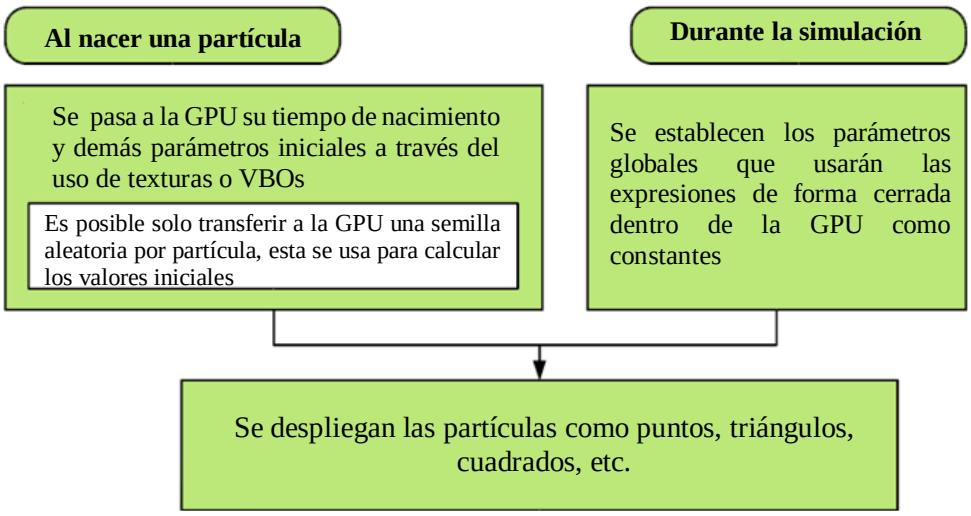


Figura 5. Diagrama general del esquema de trabajo de los sistemas de partículas sin estado basados en la GPU

La fase de generación de nuevas partículas consiste simplemente en inicializar o reinicializar los parámetros iniciales de una predeterminada cantidad de partículas típicamente calculada mediante metodologías explicadas en secciones anteriores. Por otra parte, la fase de extinción calcula el tiempo de vida restante de las partículas del sistema

mediante la resta del tiempo máximo de vida menos el tiempo de vida actual de cada partícula, el cual a su vez es frecuentemente calculado mediante la ecuación:

$$TVida = tiempoGlobal - TNacimiento$$

donde *tiempoGlobal* representa el tiempo actual del sistema de partículas y *TNacimiento* al instante de tiempo en el cual la partícula estudiada fue generada. Cuando el programa GPGPU o de vértices o fragmentos calcula que el tiempo de vida restante de una partícula llega a cero la misma es marcada como partícula inactiva, lo cual la indica como disponible para su re-inicialización de valores durante la fase de generación en la CPU.

Durante la fase actualización de atributos de las partículas el procesador de vértices se encarga de resolver una serie de ecuaciones a fin de determinar los valores de todos los atributos de una partícula necesarios para su despliegue. A continuación se ilustran fórmulas empleadas frecuentemente en sistemas de partículas sin estado implementadas en la GPU cuyas partículas son uniformemente aceleradas durante la simulación:

Atributo	**Fórmula**	**Explicación**
Posición	$P(t) = p_0 + v_0 * t + (1/2) * g * t^2$	t representa el tiempo actual, p_0 la posición inicial de la partícula, v_0 su velocidad inicial y g el vector de la gravedad que afecta al sistema.
Orientación	$\omega(t) = \omega_0 + \varphi * t$	t representa el tiempo actual, ω_0 la orientación inicial de la partícula y φ la velocidad de rotación
Color y transparencia	$f(tv) = f_0 + (f_f - f_0) * tv/t_{max}$	tv representa el tiempo de vida actual de la partícula, f_0 su color inicial, f_f su color final y t_{max} su el tiempo máximo de vida

Finalmente, a pesar de que la fase de despliegue es virtualmente idéntica a la del tipo de sistema de partículas anterior, implementar la fase de ordenamiento no es trivial debido a que como no se almacenan los valores actuales de los atributos de las partículas sino que son calculados como un paso previo al despliegue, no existe la posibilidad de comparar los valores de los atributos entre múltiples partículas de forma paralela.

1.3.4 Sistema de partículas sin estado basado en la CPU

Este tipo de sistema de partículas paramétrico implementa casi todas las fases de simulación en la CPU y emplea la GPU únicamente para la fase de despliegue de las partículas. A excepción de esta única distinción, este tipo de sistema de partículas es conceptualmente idéntico a los sistemas de partículas sin estado basados en la GPU, pudiendo emplear exactamente las mismas expresiones de forma cerrada durante la fase de simulación.

Capítulo 2: Solución propuesta

Los escenarios virtuales en los que se encuentran los sistemas de partículas típicamente están conformados por plétora de elementos diferentes como luces, cámaras, objetos concretos modelados tradicionalmente e inclusive las propias partículas del sistema. A fin de simular la interactividad característica de los fenómenos y cuerpos difusos que representan, los sistemas de partículas deben emplear métodos especializados que les permitan reconocer las estructuras y estados de los demás elementos de la escena durante las fases de simulación y despliegue de cada partícula. En la mayoría de los sistemas de partículas interactivos, los cálculos de dinámica e iluminación (en el caso de partículas que no emiten su propia luz) son los principalmente afectados por la estructura y cambios del medio ambiente: mientras que la geometría de la escena y propias partículas del sistema influyen en la dinámica del mismo, las luces y sombras del ambiente afectan el cálculo de iluminación de las partículas del sistema durante la fase de despliegue. Nuestra propuesta implementa e integra estas facetas de interacción sobre un sistema de partículas básico a fin de proveer una solución modular para sistemas de partículas densos altamente interactivos basados en la GPU.

2.1 Sistema base

Nuestra implementación propone un sistema de partículas con estado y cantidad variable, cuyas fases de simulación y despliegue son ejecutadas completamente en la GPU del dispositivo a fin de explotar las ventajas explicadas en el capítulo anterior.

Durante la fase de simulación el sistema se encarga de controlar la creación, actualización y extinción de las partículas en la GPU. Los atributos de las partículas que son considerados durante estas fases son la posición, velocidad, tiempo de vida, color y distancia existente con el punto de vista de la escena.

En la etapa de creación el sistema inserta una cantidad de nuevos elementos en cada cuadro de animación, asignándoles como posición inicial a la posición puntual del emisor y como vector de velocidad a un vector predeterminado perturbado por otro generado mediante cálculos estocásticos. Finalmente, el tiempo de vida y color de cada partícula son asignados de acuerdo valores establecidos previamente.

La etapa de actualización básica consiste en actualizar los atributos de las partículas del sistema considerando el tiempo que ha transcurrido entre el cuadro actual y el anterior. En el caso del tiempo de vida simplemente se le sustrae este intervalo de tiempo y, en caso que alcance un valor menor o igual a cero, la partícula es inmediatamente removida del sistema. Durante la actualización del atributo de posición a este se le añade el vector de velocidad actual de la partícula multiplicado por el intervalo de tiempo transcurrido a fin de evitar la interrelación entre la cantidad de cuadros por segundo que el sistema logra procesar y la cantidad de movimiento de las partículas en un intervalo de tiempo dado. En el caso del vector de velocidad, este es opcionalmente actualizado mediante la adición de un vector de gravedad multiplicado por el intervalo de tiempo transcurrido por las mismas razones que en el escenario de la actualización de posiciones. Finalmente, durante esta fase el sistema actualiza el atributo de las partículas que contiene la distancia que cada una posee con el punto de vista de la escena a fin de que este pueda ser utilizado durante etapa opcional de ordenamiento de las partículas previo a su despliegue.

Durante la fase básica de despliegue el sistema se encarga de generar una primitiva cuadrada por cada partícula en existencia, orientándolas al punto de vista de la escena y asignándoles el color y posición de cada partícula. Finalmente, se despliegan estas primitivas empleando su color multiplicado por los de la textura establecida para las partículas.

2.2 Coloración

Al igual que los objetos opacos modelados tradicionalmente, las partículas pueden ser iluminadas empleando cálculos en procesadores de vértices en la GPU logrando una coloración consistente con todas las fuentes de luz de la escena. La principal problemática de esta aproximación es que mediante cálculos simples de iluminación por vértice no se puede proporcionar la apariencia visual apropiada de la dirección de la luz entrante por lo que la coloración de las partículas es considerablemente plana y uniforme. Por otra parte, si se emplea iluminación basada en cálculos en procesadores de fragmentos de la GPU se puede conseguir una iluminación adecuada, sin embargo se limita considerablemente la densidad de partículas que el sistema puede manejar fluidamente.

A fin de solventar este problema, se implementó la propuesta de Persson [10], quien sugiere la proyección del entorno de iluminación a un espacio base HL2 (HL2-Basis) para obtener de forma eficiente degradados de la coloración de las partículas que indiquen la dirección de la luz entrante. La base HL2 es el nombre de la base para el

mapeado de normales de radiosidad diseñada por la compañía Valve [11], la cual consiste en 3 vectores ortogonales distribuidos uniformemente a través del hemisferio sobre la superficie de un plano. Para emplear esta base en la iluminación del sistema de partículas, se debe transformarla a espacio de vista a fin de alinear sus vectores con los *billboards* que despliegan las partículas. Los vectores finales obtenidos por este paso son ilustrados en la Figura 6.

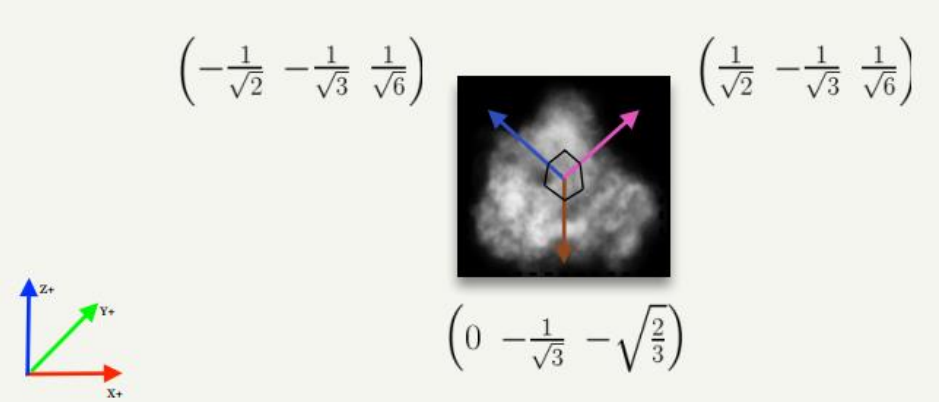


Figura 6. Vectores ortogonales de la base HL2 en espacio de vista

El algoritmo de iluminación de las partículas se distribuye entre el procesador de vértices y el procesador de fragmentos de la GPU. En el procesador de vértices se trabaja cada fuente de luz que afecta el vértice del billboard y se acumula la influencia de todas las luces incidentes. En este paso se emplean los vectores de la base HL2 en espacio de vista para calcular la influencia de cada fuente de luz mediante la multiplicación su color RGB por cada componente de su vector incidente en base HL2 (obtenido del producto punto entre el vector incidente de luz y los vectores base HL2 en espacio de vista).

En el procesador de fragmentos se procede a evaluar la iluminación por píxel. A fin de suplir las normales de cada fragmento necesarias para una iluminación adecuada, nuestra implementación considera únicamente texturas de bajo nivel de detalle que no requieren normales precisas como típicamente lo son las empleadas para simular humo y polvo, por lo que utiliza una simple aproximación de la curvatura de los *billboards* para obtener las normales. Esta aproximación es calculada mediante la interpolación entre el vector opuesto de vista de la escena y un vector contenido en el plano descrito por el billboard de cada partícula procesada.

Finalmente, se evalúa la iluminación entrante por píxel mediante la transformación de la normal a base HL2 mediante los productos puntos entre la normal y cada vector de la base HL2 en espacio de vista, seguida por la multiplicación de cada componente de la normal en base HL2 por cada componente de la luz acumulada obtenida en el procesador de vértices previamente. La Figura 7 ilustra un resultado visual obtenido mediante esta metodología.

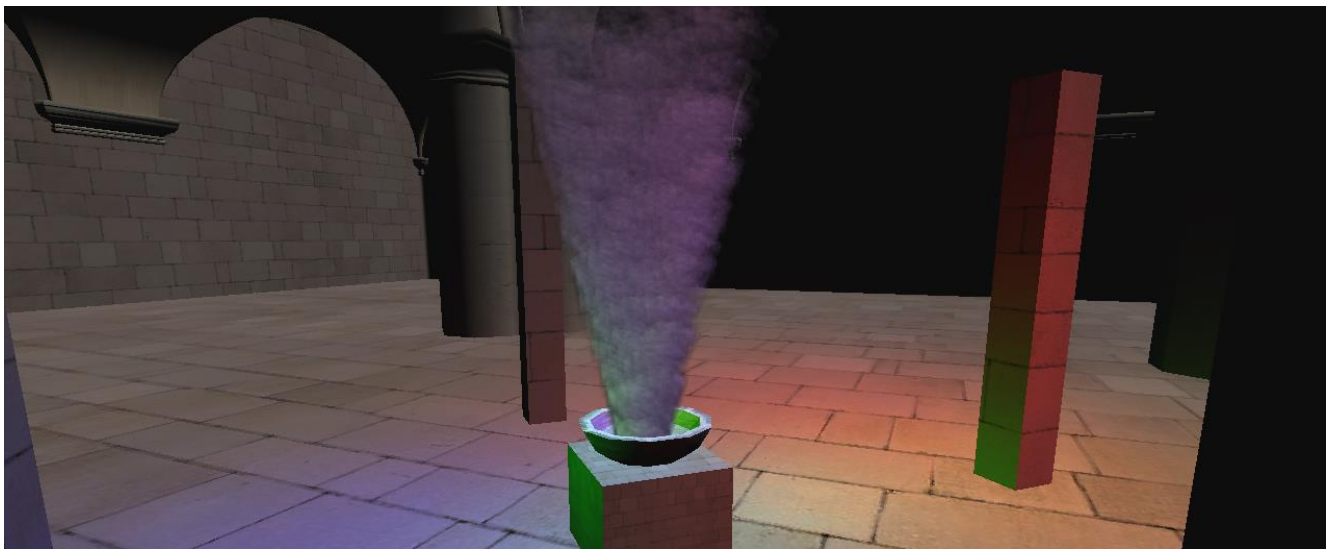


Figura 7. Partículas iluminadas de forma consistente con todas las fuentes de luz de la escena

2.3 Recepción de sombras

En la actualidad existe una considerable cantidad de técnicas capaces de lograr que objetos tridimensionales proyecten sombras sobre otros cuerpos en la escena. A pesar de que los sistemas de partículas son capaces de emplear la mayoría de las técnicas de recepción de sombras usadas por objetos tridimensionales opacos modelados tradicionalmente, la densidad de posibles partículas a trabajar en tiempo real hace imperante el uso de una metodología altamente eficiente para este tipo de circunstancias.

Nuestro sistema emplea mapas de sombras (*shadow maps*) muestreados dentro del procesador de vértices de la GPU a fin de solventar esta problemática. Los mapas de sombras son texturas que almacenan el mapa de profundidad de la geometría de la escena desde el punto de vista de las luces. Estas texturas son empleadas en un proceso de tres componentes por cada luz en la escena capaz de proyectar sombras sobre las partículas: el primero consiste en desplegar la escena compuesta por objetos modelados tradicionalmente desde la perspectiva de la luz y almacenar los valores de profundidad de las superficies observadas en el mapa de sombras, el segundo consiste en efectuar pruebas de sombreado dentro del procesador de vértices de la GPU, en las que se comparan las profundidades del mapa de sombras contra las profundidades de las partículas desde el punto de vista de la luz. Finalmente, cada partícula es considerada en sombra si su valor de profundidad es menor que el correspondiente en la muestra del mapa de sombras ya que esto indica que existe una superficie más cercana a la luz procesada.

Una consideración relevante que se tuvo que realizar en nuestra implementación fue la posibilidad de objetos opacos o luces que alteran sus atributos de posición, orientación y/o escala dinámicamente, por lo que los mapas de sombras deben ser re-calculados en cada cuadro de la animación a fin de actualizarse a estos cambios. Para reducir el impacto de esta problemática se utilizaron mapas de sombras de baja resolución, aprovechando que el muestreo de los mismos es por vértice y por lo tanto cualquier artefacto generado es considerablemente reducido durante el proceso de interpolación en el procesador de fragmento. La Figura 8 ilustra un resultado visual obtenido mediante esta metodología.

Figura 8. Partículas recibiendo sombras mediante el muestreo de mapas de sombras en el procesador de vértices

Adicionalmente, se implementó una versión alternativa del método anterior mejorado en términos de calidad de imágenes obtenidas a través del incremento de la tasa de muestreo del mapa de sombras mediante el uso de un proceso de teselación dinámico basado en el tamaño de las partículas en espacio de pantalla. La implementación de esta mejora consiste en transferir el código de muestreo del mapa de sombras antiguamente ubicado en el procesador de vértices, al procesador de dominio del sistema. En la Figura 9, se observa una comparación visual entre la metodología original y la mejorada mediante teselación en DX11.

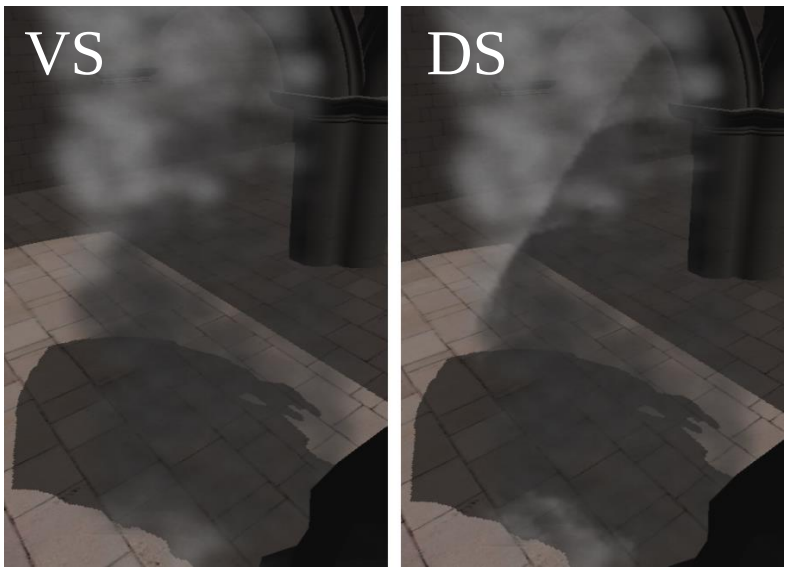


Figura 9. Partículas recibiendo sombras mediante el muestreo de mapas de sombras en el procesador de vértices (izquierda) vs muestreo realizado en el procesador de dominio (derecha) con un factor de teselado de 32

2.4 Proyección de sombras

De forma similar a la recepción de sombras, la posible densidad de elementos de los sistemas de partículas demanda el uso de un método eficiente para la proyección de sombras sobre la geometría opaca de una escena tridimensional. Como consideración adicional, la transparencia y estructura difusa de los cuerpos simulados por los sistemas de partículas hace imperante el uso de una metodología capaz de proyectar sombras de intensidad variable.

Para cumplir con estas características, se implementó la técnica de proyección de sombras basada en mapas de transparencia desarrollada por la compañía Crytek [12], la cual consiste en combinar los mapas de sombras, explicados en la sección previa, con mapas de transparencia creados de forma similar, pero sustituyendo el almacenamiento de profundidades de la escena por una etapa de acumulación de los valores de transparencia de las partículas desde el punto de vista de las luces de la escena capaces de proyectar sombras.

El procedimiento consiste en generar los mapas de sombras de forma tradicional mediante el despliegue de las partículas desde el punto de vista de las luces, con la adición del cálculo e integración del mapa de traslucidez mediante la acumulación de los valores de transparencia de los fragmentos de las partículas dentro de un canal de la textura utilizada por los mapas de sombras. Los valores de profundidad del mapa son empleados por el procesador de fragmentos para determinar las áreas oscurecidas por la luz siguiendo las pruebas de sombreado explicadas en la sección anterior aplicadas a los fragmentos de los objetos modelados tridimensionalmente encontrados dentro del punto de vista de la escena, al tiempo que se referencia a los valores de transparencia acumulada para determinar la intensidad del sombreado. En la Figura 10 se puede observar el resultado visual de una implementación de esta técnica.

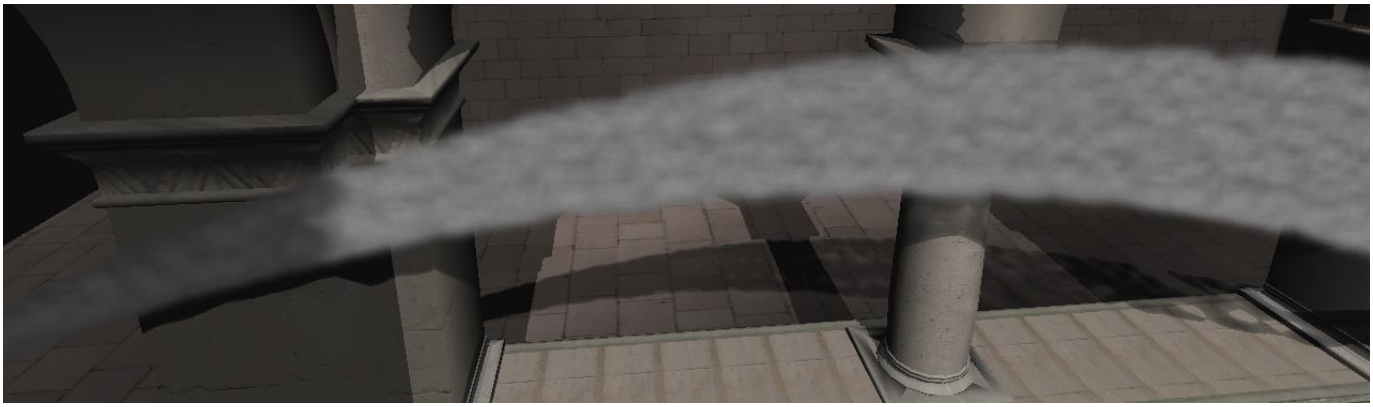


Figura 10. Simulación de partículas de humo proyectando sombras sobre geometría tridimensional opaca

2.5 Interacción entre partículas

Una de las características más enfatizadas de los sistemas de partículas en la actualidad es la capacidad de generar cuerpos difusos cuyos elementos efectúan secuencias de movimiento considerablemente complejas de forma completamente automática. Existe una gran diversidad de metodologías eficientes para lograr que los elementos de los sistemas de partículas sean capaces de generar elaborados movimientos como lo son el uso de mapas de flujo de velocidades, mapas de turbulencia e inclusive mediante el uso de fuerzas de repulsión y atracción codificadas en posiciones de las escenas tridimensionales en las que se encuentra.

Lograr la interacción entre los múltiples elementos de un sistema de partículas es una de las alternativas más empleadas a fin de generar dinámicas complejas en sistemas de partículas considerablemente densos. En nuestra implementación se utilizan dos tipos de interacción: atracción y repulsión cuyas magnitudes son dependientes de la distancia entre los elementos involucrados.

Nuestra propuesta implementa dos alternativas para lograr este tipo de interacción: la primera consiste en una resolución simple la problemática de N-Cuerpos (*N-Body problem*) en la GPU, en el cual se explota la capacidad de paralelismo de este tipo de dispositivos para lograr que toda partícula perteneciente al sistema pueda interactuar con cada una de las otras directamente. La principal problemática de esto es que la complejidad de cómputo es de $O(N^2)$, por lo que el desempeño del sistema se ve seriamente afectado en presencia de un denso número de partículas.

La segunda alternativa fue diseñada para reducir la problemática anterior y se basa en las rejillas espaciales dinámicas que permiten que la interacción solo sea calculada entre partículas que se encuentren dentro de una distancia predeterminada. Para calcular esta rejilla, se utiliza la distancia de interacción para dividir el espacio de la simulación en una rejilla uniforme. Al momento de determinar con cuales partículas interactuará un elemento del sistema la búsqueda se limita a las partículas que comparten la casilla de la rejilla uniforme con el elemento procesado más las ocupantes de las 26 rejillas vecinas.

Debido a que las densidades de los elementos de las casillas pueden variar notablemente en cada instante de la simulación, el desempeño de esta metodología puede variar drásticamente, sin embargo en la mayoría de los casos, especialmente en sistemas cuyos elementos están espaciadamente distribuidos en la rejilla, la complejidad de cómputo es menor a $O(N^2)$ ya que este pasa a convertirse en el peor escenario en vez del escenario estándar.

Es importante señalar que a pesar del superior desempeño de la segunda alternativa, los resultados visuales obtenidos por ambas son notoriamente distintos, por lo que la elección entre las dos es dependiente del resultado visual deseado y el número de elementos que el sistema de partículas maneja.

2.6 Colisiones entre partículas y la escena

Efectuar los cambios de posición de las partículas de un sistema considerando el medio ambiente y la influencia de su geometría opaca modelada tradicionalmente es una de las características que incrementan más notablemente el grado de interactividad y realismo observable de la dinámica de cualquier sistema de partículas. En la actualidad existe una gran diversidad de técnicas que permiten esta interacción con diferentes magnitudes de precisión y complejidad en sus implementaciones.

La alternativa implementada en nuestra aplicación logra este tipo de interacción mediante la técnica detección de colisiones en espacio de pantalla basada en la GPU desarrollada por Biddulph [13] en el año 2013.

La metodología consiste en cálculos basados en la información de profundidad del punto de vista de la cámara de la escena generada durante el proceso de despliegue a fin de determinar, aproximadamente, si han ocurrido colisiones entre las partículas del sistema simulado y la geometría desplegada, y en los casos afirmativos, calcular los respectivos vectores de rebote de las partículas en colisión. Debido a que normalmente solo un valor de profundidad es almacenado por pixel, un umbral es empleado para estimar el grueso de las superficies en pantalla. Este es un valor dependiente de cada escena que le permite a las partículas moverse detrás de los objetos tridimensionales y puede ser definido por emisor o inclusive por partícula. Nuestra propuesta lo establece como un atributo del sistema controlable por el usuario. La Figura 11 diagrama una simulación que emplea esta metodología para detectar colisiones en tiempo real.

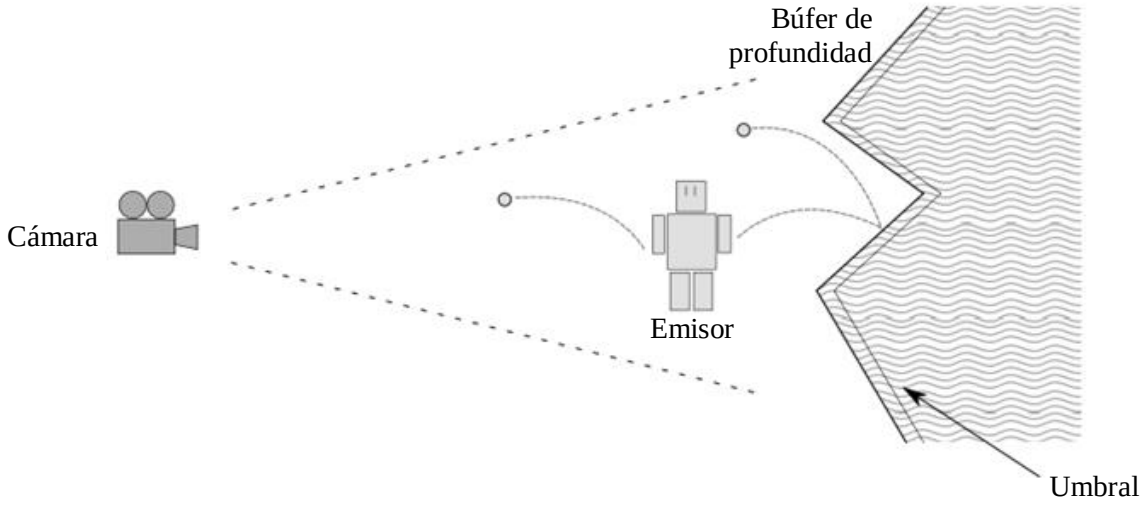


Figura 11. Diagrama de simulación de sistema de partículas empleando detección de colisiones en espacio de pantalla

El algoritmo consiste en los siguientes pasos: en primer lugar se despliega la escena desde el punto de vista de la cámara dentro de un búfer de almacenamiento (*G-Buffer*) con la información de profundidad escrita en uno de los blancos de renderizado. Posteriormente, se proyecta la posición de cada partícula en el espacio de recorte (*clip space*) de la cámara y se obtiene su valor de profundidad. Este paso es seguido por una fase de análisis de cada partícula del sistema en la que se determina si han colisionado con la geometría de la escena evaluando si sus valores de profundidades son mayores a los contenidos dentro del *G-Buffer* en las mismas posiciones. Es importante señalar que para que las partículas se consideren en colisión deben encontrarse simultáneamente dentro del umbral de profundidad preestablecido.

En los casos afirmativos, se puede calcular el vector de rebote de las partículas mediante el uso de vectores normales en espacio de vista derivados a través de dos muestreos adicionales de profundidad, cada uno a un píxel de desplazamiento del píxel central de la partícula estudiada. La combinación del valor de profundidad y la normal en un píxel permiten el cálculo un plano, el cual es empleado para calcular el vector de rebote en espacio de pantalla mediante el reflejo y amortiguamiento del vector de velocidad de la partícula transformado al espacio de pantalla. Finalmente, se procede a transformar el vector de rebote a espacio de mundo para su uso posterior por la partícula analizada. La Figura 12 ilustra varios instantes de tiempo y diferentes ángulos de punto de vista de una simulación que emplea esta técnica para detectar colisiones entre las partículas y la geometría de la escena.

Figura 12. Ejemplo de simulación de un sistema de partículas denso empleando detección de colisiones en espacio de pantalla dentro de la GPU

Adicionalmente, se implementó una versión refinada de esta metodología donde se sustituyó del mapa de profundidades simple por un mapa cubico de profundidades generado a partir de la posición de la cámara de la escena. Esto provee información adicional que reduce los artefactos de movimiento generados por la reducida cantidad de información de la geometría del entorno del sistema de partículas. En la implementación basada en mapas de profundidad simples el rápido cambio del enfoque de la cámara de la escena puede sacar abruptamente a partículas que se encontraban dentro de geometría debido a que las colisiones no son calculadas si las partículas y la geometría no se encontraban en dicho enfoque, generando notables disparidades en el movimiento de sistemas de partículas de generación continua. Alternativamente, el uso de mapas de profundidad cúbicos permite que las colisiones se ejecuten independientemente del enfoque del punto de vista de la escena, permitiendo que las mismas no se interrumpen o alteren si no se encuentran dentro de dicho enfoque.

Capítulo 3: Implementación

Como se mencionó en el capítulo anterior nuestra propuesta de sistema de partículas se encuentra basada en la GPU, empleando la CPU exclusivamente para labores de control de flujo de la aplicación e interfaz de usuario. Para lograr esto se utilizó la colección de interfaces de programación de aplicaciones DirectX 11 de Microsoft conjunto a su lenguaje de programación HLSL para la implementación de los cómputos de las fases de simulación y despliegue del sistema de partículas y su entorno de pruebas. Es imperante señalar que los programas que conforman la fase de simulación fueron implementados mediante GPGPU haciendo uso de la interfaz DirectCompute de DirectX.

A fin de implementar el control de flujo en la CPU se utilizó el lenguaje de programación C++, complementado con la librería AntTweakBar para el desarrollo de la interfaz de usuario y el motor de simulación de físicas PhysX de la compañía Nvidia [14] para la implementación de una alternativa de la fase de simulación del sistema de partículas a utilizar de marco de referencia durante las fases de prueba del presente documento. En las siguientes secciones se estudiará en profundidad la implementación y estructura del sistema de partículas conjunto a los módulos de interacción desarrollados.

3.1 Aplicación

En el flujo principal de la aplicación el sistema de partículas fue implementado como una clase única que posee todos los atributos y funcionalidades asociadas al manejo y despliegue de las partículas. La aplicación se encarga de suministrar al sistema toda la información del entorno que este requiere, además controlar la frecuencia de la ejecución de sus fases. A fin de facilitar la visualización de la estructura planteada previamente, en la Figura 13 se expone el diagrama de clases de la aplicación que controla al sistema de partículas [17], donde por motivos de legibilidad se han suprimido los atributos de sus clases.

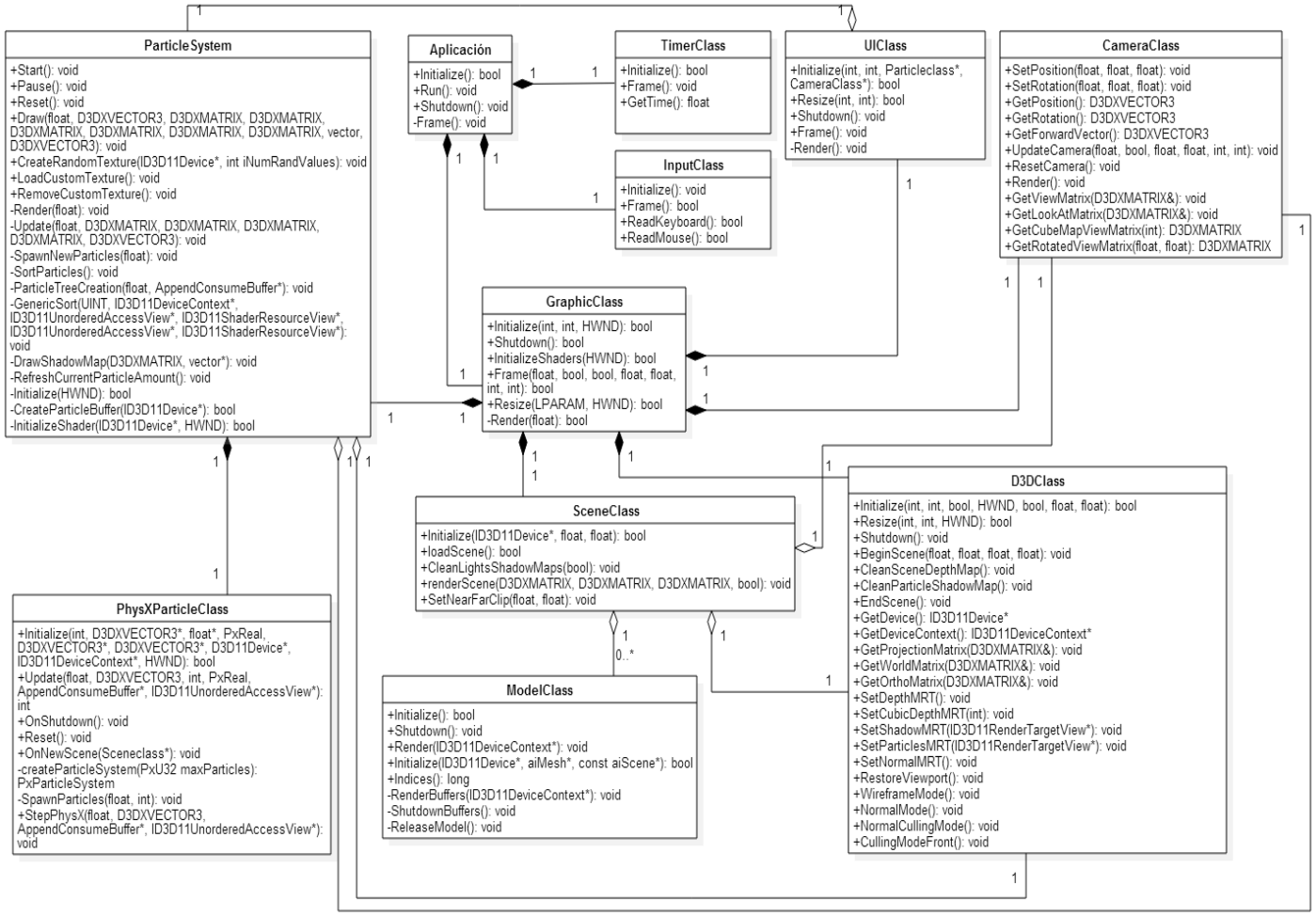


Figura 13. Diagrama de clases de la aplicación encargada del manejo de flujo del sistema de partículas

Durante la ejecución, la aplicación se encarga de mantener un bucle de despliegue de los cuadros de animación mediante la llamada cíclica de la función de despliegue contenida en la clase de manejo de gráficos, empleando la clase *Timer* para obtener y suministrar el tiempo transcurrido entre el presente cuadro y el anterior. Es la clase manejadora de gráficos la que se encarga de solicitar el despliegue del escenario y el sistema de partículas, además de suministrarle a este último la información sobre escenario en donde se encuentra.

La inicialización y actualización de la cámara de la aplicación también son ejecutados por la clase manejadora de gráficos, la cual le transmite los comandos de los usuarios captados por la clase *Input* a fin de que la cámara rote y se traslade dentro de la escena de forma acorde. Utilizando su posición y rotación en cada instante de tiempo, la cámara se encarga de generar las matrices de vista que son utilizadas tanto en los cálculos de interacción con el escenario como en los del despliegue de las partículas del sistema.

La clase *D3D* se encarga de la inicialización y manejo puntual de DirectX, definiendo el inicio y final de las operaciones y cálculos de cada cuadro de animación. Esta clase también debe suministrar las matrices de mundo y proyección definidas durante la etapa de inicialización a fin de proveérselas a la clase de manejo de gráficos durante la ejecución de la aplicación y puedan ser utilizadas, junto a la matriz de vista de la cámara, por tanto la escena como el sistema de partículas durante sus fases de despliegue.

Finalmente, la clase manejadora de gráficos también se encarga de solicitar el despliegue de la interfaz de usuario a la clase *UI* y las operaciones de comienzo y final de cuadro de animación a la clase *D3D*, las cuales se encargan respectivamente de colocar el blanco de renderizado (estructura de memoria donde se almacena la información a desplegar gráficamente en cada cuadro) y solicitar su despliegue por el monitor utilizado. En la Figura 14 se expone una abstracción del diagrama de secuencia del flujo descrito previamente [18].

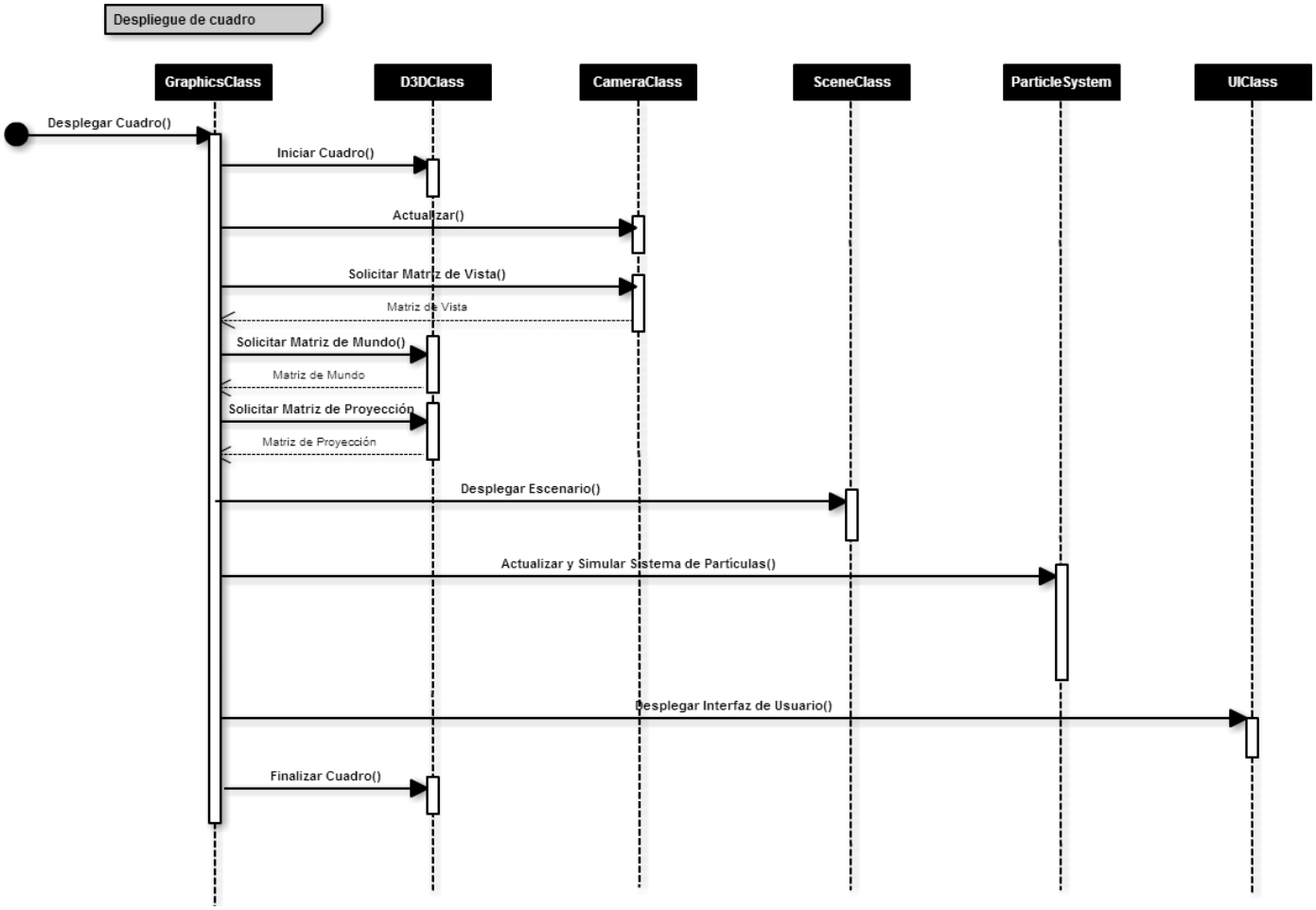


Figura 14. Diagrama de secuencia parcial de la operación de despliegue de cuadro de animación contenida en la clase manejadora de gráficos

3.2 Sistema base

Las fases de despliegue y simulación del sistema de partículas son reguladas por el CPU a través de la clase *ParticleSystem*, la cual posee todas las referencias a los métodos y atributos que son solicitados y transferidos a la GPU para su procesamiento lo que le permite dictar el orden y frecuencia de ejecución de las operaciones asociadas a los cálculos del sistema. Esta clase posee las referencias a dos interfaces de búferes de memoria de la GPU que contienen a las partículas del sistema en cada instante de tiempo dado, dentro del esquema propuesto estas interfaces son referidas como *InputBuffer* y *OutputBuffer*. Es importante señalar que ambos búferes referenciados en la GPU son del tipo *AppendConsumeBuffer*, el cual es un tipo especial de búfer de memoria en la GPU que permite la adición y extracción de elementos de forma análoga al funcionamiento de las pilas tradicionales. La Figura 15 ilustra esta funcionalidad aplicada a nuestra propuesta.

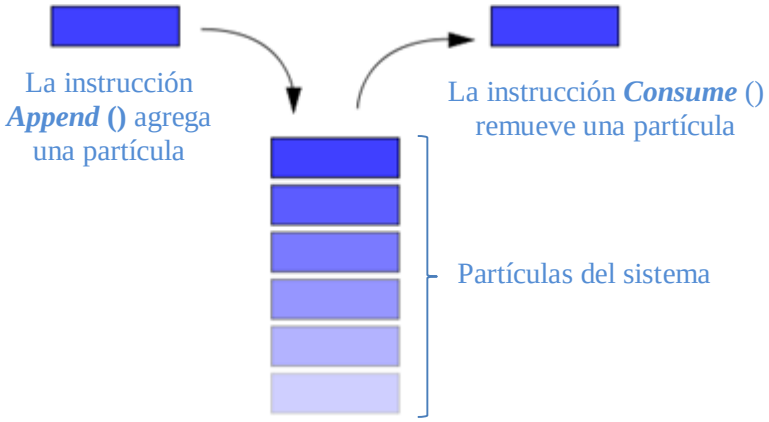


Figura 15. Los AppendConsumeBuffers son utilizados para el almacenamiento dinámico de las partículas del sistema, permitiendo la fácil inserción y eliminación de partículas.

Nuestra propuesta utiliza dos búferes para contener a las partículas debido a que la GPU no acepta operaciones de escritura y lectura simultanea de datos contenidos en una misma estructura de memoria, por lo que el sistema se ve incapaz de acceder y actualizar los estados de las partículas en la GPU si utiliza un único buffer para almacenar sus estados. Para emplear los dos búferes se implementó un esquema de trabajo denominado *ping-pong* [19], el cual consiste en leer los datos contenidos en uno de los búferes mientras que se escribe en el otro. En el siguiente cuadro de animación se intercambian los roles y se utiliza el búfer con los estados actualizados como el nuevo búfer de lectura, mientras que el anterior se emplea como nuevo búfer de escritura para los procesos en la GPU encargados de insertar, actualizar y eliminar partículas. Este procedimiento rotativo permite que el sistema siempre tenga acceso a un búfer con los estados actualizados de las partículas que lo conforman en cada cuadro de animación.

Para la ejecución de los programas en la GPU, la aplicación debe suplir los valores dinámicos requeridos en cada escenario dentro de cada instante de tiempo mediante la transferencia de datos de la CPU a la GPU. Esta transferencia de datos es posible gracias a los búferes *D3D11Buffer*, los cuales funcionan como interfaces que permiten la interacción con los búferes de memoria en la GPU a los que los programas ejecutados en este dispositivo acceden para leer y modificar. A fin de realizar estas transferencias se emplea el comando *Map*, el cual bloquea temporalmente el acceso al búfer de memoria mientras se efectúa el copiado de datos de la CPU a la GPU.

3.2.1 Generación de partículas

Durante la etapa de generación de nuevas partículas la aplicación se encarga de transferirle a la GPU los valores establecidos por el usuario en cada instante de tiempo para la posición del emisor, tiempo de vida de las partículas, velocidad inicial y su varianza, tiempo transcurrido entre el cuadro actual y el anterior, y la cantidad de nuevas partículas a introducir al sistema en el cuadro actual. Adicionalmente, durante la inicialización de la aplicación se le transfiere a la GPU una textura unidimensional estática con canales RGB en los que se introducen valores aleatorios generados en la CPU, la cual será utilizada en cálculos estocásticos de esta etapa para proporcional aleatoriedad al vector de velocidad inicial de las partículas generadas.

El algoritmo de generación de nuevas partículas consiste los siguientes pasos: primero se extrae un vector aleatorio de la textura previamente mencionada utilizando el índice de los hilos como la coordenada de muestreo de la textura. Posteriormente, este vector es multiplicado por la varianza de la velocidad y sumado al vector de velocidad inicial establecido por el usuario a fin de calcular el valor final de la velocidad, el cual es asignado a una partícula local de la función. En el caso de los atributos de posición inicial y tiempo de vida, estos son simplemente asignados a la misma partícula según los valores que han sido copiados desde la CPU, los cuales a su vez son determinados por el usuario de la aplicación.

Finalmente, la partícula local es agregada al búfer que contiene al resto de las partículas del sistema mediante el comando *Append (elemento a insertar)* ilustrado en la Figura 15, garantizando su existencia durante la consiguiente fase de simulación.

Cabe destacar que el mencionado algoritmo de generación no será ejecutado si el programa determina que la cantidad de partículas emitidas durante el cuadro actual excede el valor establecido por el usuario. Para efectuar esta comprobación simplemente se verifica que el índice del hilo actual no exceda dicho valor. En el Algoritmo 1 se puede observar el código de la función principal utilizada por esta etapa.

```
[numthreads (1024, 1, 1)]
void CSMAIN(uint3 GroupThreadID : SV_GroupThreadID)
{
	// Identificador del hilo actual
	uint myID = GroupThreadID.x;

	// Límite de emisión
	if (myID < EmissionLimit)
	{
		// Se crea la partícula local
		Particle p;

		// Se extrae un vector de la textura de valores aleatorios utilizando
		// como coordenada de muestreo al índice del hilo actual
		float3 vRandom = normalize(RandomDir (myID));

		// Se inicializa la velocidad inicial de acuerdo a un valor predefinido
		// perturbado por otro de valor aleatorio según la varianza establecida
		p.velocity = BaseVelocity + vRandom * Variance;

		// Se inicializa la posición de la partícula a la posición actual del
		// emisor del sistema
		p.position = EmmiterPos.xyz;

		// Se inicializa el tiempo de vida de la partícula en segundos
		p.time = LifeTime;

		// Se introduce la partícula al sistema
		CurrentSimulationState.Append (p);
	}
}
```

Algoritmo 1. Código utilizado para la inserción de partículas en el sistema

3.2.2 Actualización y extinción de partículas

Durante la etapa de actualización la aplicación verifica si existe al menos una partícula en el sistema mediante el comando *CopyStructureCount*, el cual copia a la CPU el número de partículas almacenadas en el búfer que será actualizado. Si la condición previa se cumple, la aplicación se encarga de transferirle a la GPU los valores de tiempo transcurrido entre el cuadro actual y el anterior, vector de gravedad, número actual de partículas en el sistema, posición del emisor, masa de las partículas, umbral de colisión, tipo de interacción entre las partículas y con el escenario, posición de la cámara, además de las matrices de proyección y vista (regular y cúbica) de la escena.

La mayoría de los datos previamente señalados son utilizados en los módulos de interacción avanzados que serán explicados en las siguientes secciones, sin embargo, se puede señalar que los parámetros de vector de gravedad

y tiempo transcurrido entre el cuadro actual y el anterior son utilizados por el algoritmo de actualización básico a fin de modificar los atributos de posición y velocidad de las partículas, a través del código expuesto en el Algoritmo 2.

```
[numthreads(512, 1, 1)]
void CSMAIN(uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid : SV_GroupThreadID, uint GI :
SV_GroupIndex)
{
        // Se obtiene el identificador del hilo actual
        uint myID = DTid.x + DTid.y * 512 + DTid.z * 512 * 512;

        // Se cerciora que no se intente procesar más partículas de las que actualmente existen
        if (myID < NumParticles){

                // Se extrae la información de la partícula asociada al hilo actual
                Particle p = CurrentSimulationState.Consume();

                // Se actualiza el tiempo de vida de la partícula mediante la substracción del tiempo
                // transcurrido entre el cuadro actual y el anterior
                p.time = p.time - DeltaTime;

                // A partir de este punto solo se consideran las partículas que aún siguen vivas, el resto no son
                // añadidas al búfer con los estados actualizados por lo que son efectivamente eliminadas del
                // sistema
                if (p.time > 0)
                {
                        // Update the particle velocity
                        p.velocity += Gravity * DeltaTime;

                        // Se calcula la nueva posición de la partícula considerando el nuevo vector de velocidad
                        p.position += p.velocity * DeltaTime;

                        // Interacción entre las partículas y el escenario
                        if (InteractionType >= 3) particleSceneInteraction(p);

                        // Se añade el índice y distancia de la partícula actual con el punto de vista de la escena
                        // a un búfer de índices, esto es considerado solo para la etapa opcional de ordenamiento
                        uint index = g_IndexBuffer.IncrementCounter();
                        g_IndexBuffer[index] = float2(length(p.position - CameraPosition.xyz), index

                        // Se añade la partícula al búfer con los estados actualizados, efectivamente manteniéndola
                        // viva en el sistema
                        NewSimulationState.Append(p);
                }
        }
}
```

Algoritmo 2. Código utilizado para la actualización de las partículas

En el código anterior se pueden observar un par de particularidades que necesitan ser comentadas. En primer lugar se observa que se puede extraer la información de un *AppendConsumeBuffer* mediante la mencionada operación *Consume ()*, la cual elimina el elemento del búfer afectado pero simultáneamente devuelve una referencia al elemento a fin de que pueda ser manipulado y transferido a otra memoria para su almacenamiento.

La segunda particularidad reside en *g_IndexBuffer*, el cual es un búfer especializado encargado de contener los índices de las partículas y las distancias que poseen con el punto de vista de la escena. Este búfer es utilizado durante la etapa opcional de ordenamiento y a diferencia de los estudiados previamente, es de tipo *RWStructuredBuffer*, lo cual significa que aunque no posee las facultad de ejecutar las operaciones de *Append ()* y *Consume ()*, se puede acceder y modificar a sus elementos mediante índices de forma análoga a un arreglo de memoria tradicional.

Finalmente, se puede observar que la etapa de extinción es ejecutada de forma implícita gracias al modelo de *ping-pong* utilizado por la aplicación: debido a que el búfer donde se almacenan los estados actualizados se convierte en el búfer fuente en el siguiente cuadro de animación, toda partícula que no se encuentre en el mismo es efectivamente removida del sistema ya que es consumida pero nunca es reintroducida. El condicional que evalúa si el tiempo de vida restante de una partícula es mayor a cero se convierte entonces en el criterio de extinción de partículas utilizado por nuestra propuesta.

3.2.3 Ordenamiento

Para implementar la etapa opcional de ordenamiento de las partículas según sus distancias con el punto de vista de la escena se utilizó *Bitonic Sort*, el cual es un simple algoritmo que funciona ordenando los datos de forma alternada en secuencias ascendentes y descendentes, las cuales posteriormente pueden ser combinadas y ordenadas

para producir secuencias aún mayores. Esto es repetido hasta que el algoritmo produce una secuencia final con todos los datos ordenados.

Para nuestra aplicación se utilizó la implementación del algoritmo basada en procesadores de cómputo propuesta por Microsoft [20], la cual consiste en efectuar los siguientes pasos secuenciales sobre *g_IndexBuffer* a fin de transformarlo en un búfer que contenga los índices de las partículas ordenadas de menor a mayor de acuerdo a su distancia con el punto de vista de la escena.

1. Paso 1: Se carga la información de cada partícula en una memoria compartida por los grupos de hilos. Cada hilo carga un elemento.

```
shared_data[GI] = Data[DTid.x];
```

2. Paso 2: Posteriormente, los hilos deben ser sincronizados para garantizar que todos los elementos han sido cargados debido a que la próxima operación efectuará lecturas de acceso aleatorio.

```
GroupMemoryBarrierWithGroupSync();
```

3. Paso 3: Ahora cada hilo debe escoger el mínimo o máximo de los dos elementos que está comparando. Es en este paso donde se evalúa la distancia con el punto de vista de la escena calculado durante la etapa de actualización de las partículas.

```
float2 a = shared_data [index];

float2 b = shared_data [nSwapElem];

if ( a.Distancia > b.Distancia )
{
        shared_data [index] = b;

        shared_data [nSwapElem] = a;
}
```

4. Paso 4: De nuevo los hilos deben ser sincronizados para evitar que algún hilo ejecute operaciones de escritura antes que todos los hilos terminen los procesos de lectura. Luego de esto el algoritmo se devuelve al paso 3 y debe terminar todas las escrituras antes de comenzar a leer de nuevo.
5. Paso 5: Con la memoria ordenada, los resultados pueden ser almacenados de vuelta al bufer de memoria

```
for( i = 0; i<2*ITERATIONS; ++i )
{
        if( GI+i*NUM_THREADS<numElementsInThreadGroup )
                Data[ GlobalBaseIndex + i*NUM_THREADS ] = shared_data [ LocalBaseIndex + i*NUM_THREADS ];
}
```

El impacto de la fase de ordenamiento es drásticamente notable en nuestra implementación de sistema de partículas debido a que se utilizan elementos con transparencia y oclusión mutua. La Figura 16 representa un caso de estudio comparativo de la importancia de esta fase opcional dentro de nuestra aplicación.

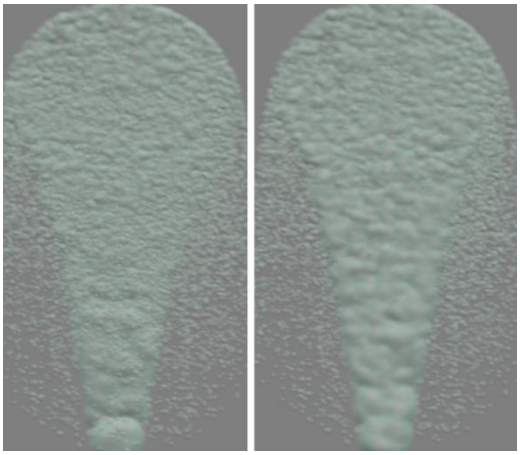


Figura 16. Comparación entre sistema de partículas sin fase de ordenamiento (izquierda) y con fase de ordenamiento (derecha)

3.2.4 Despliegue

Finalmente, en la fase de despliegue se procede a procesar todas las partículas contenidas en el respectivo búfer de memoria a fin de representarlas gráficamente. En esta fase la aplicación transfiere a la GPU los datos correspondientes al color base y tamaño de las partículas, vector delantero y posición de la cámara de la escena, matriz de vista, matriz de mundo-vista-proyección, inversa de la matriz de vista, y la información de todas las luces de la escena, es decir, sus posiciones, direcciones, matrices de mundo-vista-proyección y coloraciones. Adicionalmente, se le indica a la GPU el búfer donde se encuentra las partículas actualizadas y el búfer donde están almacenados los índices de las partículas ordenadas según su distancia con el punto de vista de la escena.

Antes de solicitar los cálculos de despliegue en la GPU, la aplicación comanda el uso del modo de mezclado basado en valores de transparencia y bloquea la escritura en el buffer de profundidad del blanco de renderizado principal mediante las funciones *OMSetBlendState* y *OMSetDepthStencilState* respectivamente, lo cual permite que las partículas puedan ser desplegadas con transparencia y sin oclusión mutua. Esto último es especialmente importante debido a que si se permitiese la escritura en el buffer de profundidad durante el despliegue de las partículas, esta oclusión no consideraría la transparencia de las mismas, por lo que serían ocluidas considerando el quad completo en el que son desplegadas generando artefactos visuales como los representados por la Figura 17.

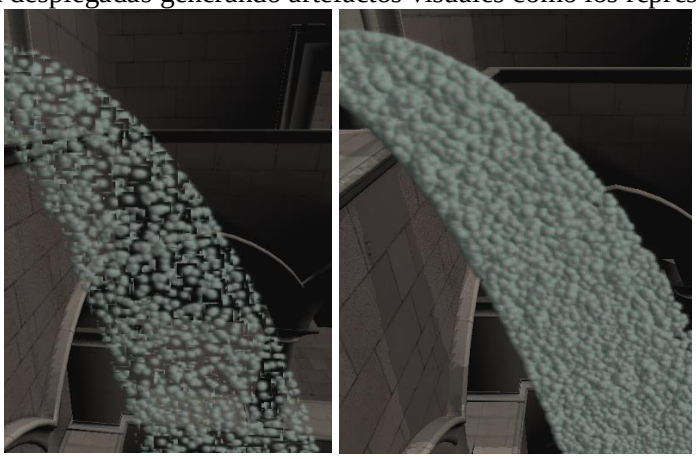


Figura 17. Artefactos visuales generados por la oclusión entre las partículas (izquierda) y representación correcta sin esta oclusión (derecha)

Los cálculos básicos de despliegue de las partículas del sistema son distribuidos entre los procesadores de vértice, geometría y fragmento. El procesador de vértices es convocado una vez por cada partícula que se encuentra en el sistema, y su labor es únicamente extraer la posición en el mundo de cada partícula y su coloración básica considerando la luz ambiental de la escena y el color de las partículas definido por el usuario. A fin de desplegar las partículas de acuerdo a la fase opcional de ordenamiento, este procesador utiliza el identificador de vértice siendo procesado como índice para extraer del búfer de índices ordenados el índice de la partícula que debe ser procesada. En el Algoritmo 3 se expone el código en HLSL utilizado para este proceso.

```
GS_INPUT VSMAIN(in VS_INPUT input)
{
	GS_INPUT output;

	// Solo se procesa una partícula por vértice, así que se utiliza el identificador de vértice como índice en el arreglo que
	// posee los índices de las partículas ordenadas
	uint particleIndex = input.vertexid;

	// Se extrae el índice global de la partícula
	uint index = (uint)g_SortedIndexBuffer[NumeroActualDeElementos - particleIndex - 1].y;

	// Se obtiene la posición de la partícula en el mundo
	output.position = SimulationState[index].position;

	// Se obtiene el color de la partícula considerando la luz ambiental y valor predefinido por el usuario
	output.color = float4(saturate(ambientColor + ParticleColor.xyz), ParticleColor.a);

	// Se transfiere esta información al procesador de geometría
	return output;
}
```

Algoritmo 3. Código utilizado en el procesador de vértices para el despliegue de las partículas

En el procesador de geometría el programa se encarga de generar los *billboards* para cada posición de partícula recibida desde el procesador de vértices. Para esto, el procesador de geometría genera, por cada posición de partícula entrante, cuatro vértices que en conjunto conforman un quad orientado al punto de vista de la escena. Durante estos cálculos se utilizan cuatro posiciones genéricas de un quad en espacio de vista multiplicadas por el tamaño de las partículas definido por el usuario; el resultado de estas cuatro operaciones es multiplicada por la inversa de la matriz de vista a fin de convertirlas a coordenadas de mundo y luego trasladadas mediante la adición de la posición de la partícula. Finalmente, el resultado es multiplicado por la matriz de mundo-vista-proyección para obtener la posición final de cada vértice del quad. En el Algoritmo 4 se expone el código en HLSL fue el utilizado para esta implementación.

```
[maxvertexcount(4)]
void GSMAIN(point GS_INPUT input[1], inout TriangleStream<PS_INPUT> SpriteStream)
{
	PS_INPUT output;

	// Se emiten dos triángulos que en conjunto conforman un quad orientado al punto de vista de
	// la escena
	for (int i = 0; i < 4; i++)
	{
		// Se introducen las coordenadas de textura genéricas para cada esquina del quad
		output.texcoords = g_texcoords[i];

		// Se escala la posición genérica de los bordes del quad en espacio de vista
		// por el tamaño de las partículas
		float3 position = g_positions[i] * ParticleSize;

		// El resultado es transformado al espacio de mundo y trasladado por la posición de la partícula
		position  = mul(position, (float3x3)g_mInvView) + input[0].position;

		// Finalmente se obtiene la posición final del vértice y se añade a la primitiva que conformará
		// al quad
		output.position = mul(float4(position, 1.0), g_mWorldViewProj);
		SpriteStream.Append(output);
	}

	// Se declara el fin de la primitiva
	SpriteStream.RestartStrip();
}
```

Algoritmo 4. Código utilizado en el procesador de geometría para el despliegue de los quads que representan las partículas

Para culminar, en el procesador de fragmentos se muestrea la textura de las partículas y se multiplica el color de esta por la coloración calculada en el procesador de vértices para determinar el color final de cada fragmento del quad siendo procesado.

3.3 Coloración

A fin de implementar la iluminación en espacio base HL2 se añadieron cálculos especializados a los programas ejecutados dentro de los procesadores de vértice, geometría y fragmentos de la GPU señalados previamente. En el procesador de vértices se añadió un ciclo que itera por todas las luces de la escena y acumula la iluminación producida por estas en espacio HL2. Para efectuar esta acumulación se utiliza una función que primero se multiplica el color de la luz por su factor de atenuación si poseen uno, y posteriormente proyecta el vector de dirección de la luz al espacio HL2. Finalmente se procede a acumular de forma independiente el resultado de la multiplicación de la coloración atenuada de la luz por cada componente de la dirección de la luz en espacio HL2. Algoritmo 5 expone el código aplicado a cada luz de la escena para efectuar esta acumulación.

```
// HL2-basis
static const float3 hl2_basis0 = float3(-1.0 / sqrt(2.0), -1.0 / sqrt(3.0), 1.0 / sqrt(6.0));
static const float3 hl2_basis1 = float3(1.0 / sqrt(2.0), -1.0 / sqrt(3.0), 1.0 / sqrt(6.0));
static const float3 hl2_basis2 = float3(0.0, -1.0 / sqrt(3.0), -sqrt(2.0 / 3.0));

void accumulate_lighting(inout GS_INPUT o, half3 light_dir, half3 light_col, half atten) {
	light_col *= atten;
	half3 weights = saturate(float3(dot(light_dir, hl2_basis0), dot(light_dir, hl2_basis1),dot(light_dir, hl2_basis2)));

	o.basis_col0 += light_col * weights.x;
	o.basis_col1 += light_col * weights.y;
	o.basis_col2 += light_col * weights.z;
}
```

Algoritmo 5. Código utilizado para acumular las intensidades de todas las luces que afectan un vértice

Para evaluar apropiadamente la luz por píxel dentro del procesador de fragmento, es requerido algún tipo de normal. Debido a que las partículas normalmente no poseen este atributo, el mismo debe ser generado por vértice de quad en el procesador de geometría. Para solucionar esta problemática nuestra propuesta genera los vectores normales de las partículas mediante una aproximación de curvatura calculada a partir de la interpolación basada en un valor de curvatura establecido por el usuario entre el vector opuesto frontal del punto de vista y un vector paralelo al plano descrito por el quad siendo procesado. El código expuesto en el Algoritmo 6 es añadido al ciclo generador de vértices del procesador de geometría expuesto en la sección anterior.

```
// Se obtiene el centro de la partícula en espacio de mundo
half3 center = mul(float3(0, 0, 0), (float3x3)g_mInvView) + input[0].position;

// Se obtiene el vector opuesto frontal del punto de vista de la escena (es decir,
// un vector orientado hacia el punto de vista)
half3 billboard_normal = -ViewVector.xyz;

for (int i = 0; i < 4; i++)
{
	// Se calcula la normal del vértice actual mediante la interpolación entre el vector
	// orientado hacia el punto de vista y un vector paralelo al plano del quad utilizando
	// un valor personalizable establecido por el usuario
	half3 n = lerp (billboard_normal, normalize(worldPosition - center), Curvatura);

	// Se le añade el nuevo vector normal al vértice
	output.normal = n;
}
```

Algoritmo 6. Código utilizado en el procesador de geometría para generar las normales de los vértices de los quads de las partículas

Finalmente, la iluminación entrante a la partícula es evaluada en el procesador de fragmentos, donde se convierte la normal del pixel procesado a espacio HL2 de forma análoga al procedimiento empleando con la dirección de la luz en el procesador de vértices, y posteriormente se realiza el producto entre los componentes de la normal transformada y los componentes de la luz en espacio HL2 calculados previamente a fin de mezclarlos en un único color difuso. En la Figura 18 se puede observar una ilustración comparativa de múltiples apariencias de luz entrante alcanzables mediante la variación del parámetro de curvatura utilizado durante el cálculo de las normales de los vértices.

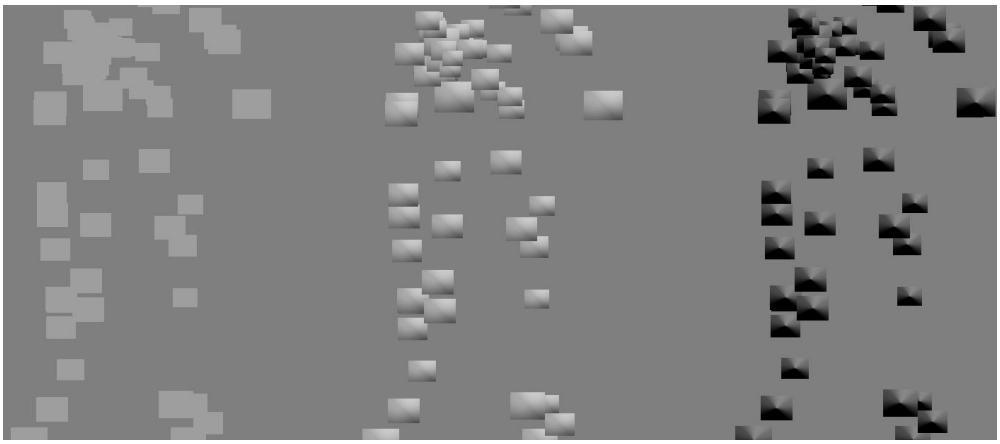


Figura 18. Partículas sin transparencia con valores de curvatura de cero (izquierda), cero punto cinco (centro) y uno (derecha)

El código utilizado dentro del procesador de fragmento para calcular la iluminación final difusa en espacio HL2 se expone en el Algoritmo 7.

```
// Se evalúa la iluminación entrante por píxel
half3 n = normalize(input.normal);
half3 w = saturate(half3(dot(n, hl2_basis0), dot(n, hl2_basis1), dot(n, hl2_basis2)));
half3 finalDiffuseLight = input.basis_col0 * w.x + input.basis_col1 * w.y + input.basis_col2 * w.z;
```

Algoritmo 7. Código utilizado para la evaluación de iluminación en el procesador de fragmento

3.4 Recepción de sombras

Para lograr la recepción de sombras por parte de las partículas del sistema primero se tuvo que implementar los algoritmos necesarios para producir los mapas de sombras de los objetos modelados tradicionalmente. El procedimiento consiste en el despliegue de los modelos de la escena desde la perspectiva de cada luz con la capacidad de producir sombras; durante esta etapa los modelos son desplegados en blancos de renderizado de canal único desde la perspectiva de cada luz. Estos blancos son las texturas almacenadas en la GPU que posteriormente son utilizadas como mapas de sombras por el sistema de partículas. En la Figura 19 se puede observar representaciones gráficas de los mapas de sombras generados por nuestra propuesta.

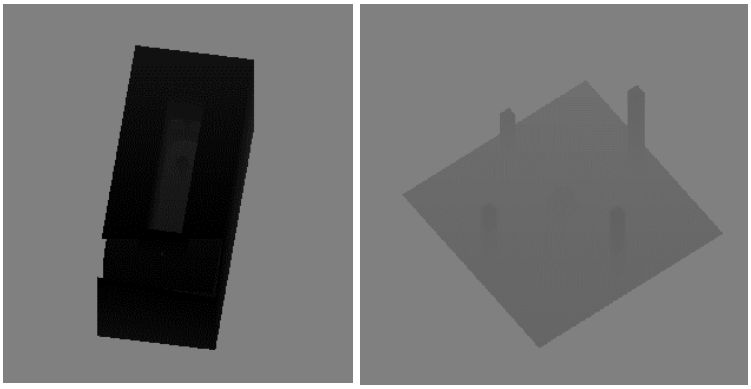


Figura 19. Ejemplo de mapas de sombras generados por aplicación. Los mapas son textura de canal único que almacena los valores de profundidad desde la perspectiva de las luces

Posteriormente, estos mapas de sombras son utilizados en la fase de despliegue de partículas durante los cálculos de recepción de sombras. En la primera alternativa implementada, cada mapa es utilizado dentro del procesador de vértices para la evaluación de oclusión de la posición de la partícula según el procedimiento explicado en capitulo anteriores. Esto es efectuado dentro un ciclo que itera por todos los mapas creados y el código utilizado se expone en el Algoritmo 8.

```
// Ciclo que itera por todas las luces de la escena
[loop]
for (int i = 0; i < NumLuces; i++)
{
	// Se resetea la variable que define la intensidad de sombreado (1.0 significa que el vértice no esta
	// en sombra)
	shadowFactor = 1.0f;

	// Se obtiene la posición de la partícula desde el punto de vista de la luz mediante la
	// multiplicación de su posición en espacio de mundo por la matriz de mundo-vista-proyección de la luz
	float4 lightViewPosition = mul(float4(output.position, 1.0), LightMatrix[i]);

	// Se homogeniza la posición
	lightViewPosition.xyz /= lightViewPosition.w;

	// Se transforma la posición a coordenadas de textura para poder efectuar las pruebas de sombreado
	// (Se pasa de coordenadas en rango [-1,1] a coordenadas en rango [0,1]
	lightViewPosition.x = lightViewPosition.x / 2 + 0.5;
	lightViewPosition.y = lightViewPosition.y / -2 + 0.5;

	// Se obtiene el valor de profundidad del mapa de sombras de la escena que coincide con las coordenadas
	// de la partícula siendo procesada
	float shadowMapDepth = GetShadowMapDepth(i, lightViewPosition.xy);

	// Si la profundidad de la partícula es mayor al valor contenido en el mapa de sombras, se considera
	// a la partícula en sombra y se inserta en esta variable la intensidad de la misma
	if (shadowMapDepth < lightViewPosition.z)
		shadowFactor = max(shadowFactor - ShadowIntensity[i].x, 0);

	// Se procede a efectuar la acumulación de iluminación estudiada en secciones anteriores, el resultado
	// se multiplica por la variable shadowFactor para aplicar el sombreado acorde
}
```

Algoritmo 8. Código utilizado para la evaluación de los mapas de sombras

La segunda alternativa transfiere el código previo al esquema de despliegue alternativo que hace uso del proceso dinámico de teselado. En este esquema se remueve el uso del procesador de geometría y añade el uso de dos nuevos procesadores: el procesador de cáscara y el procesador de dominio.

Dentro de este esquema el procesador de vértices recibe cuatro entradas de datos (idénticas a excepción del identificador de vértice que varía incrementalmente) por cada partícula en el sistema. Utilizando el identificador de instancia este procesador es capaz de extraer la posición de la partícula siendo analizada, y mediante el identificador vértice otorga a cada una de las cuatro entradas la posición de una de las esquinas del futuro billboard, además de proporcionarles sus debidas coordenadas de texturas y vectores normales, efectivamente suplantando las funcionalidades previamente asociadas al procesador de geometría. En el Algoritmo 9 se proporciona el código del procesador de vértices utilizado dentro de este esquema de trabajo.

```
VS_CONTROL_POINT_OUTPUT VSMAIN_TESSELATED(in VS_INPUT_TESSELATED input)
{
        VS_CONTROL_POINT_OUTPUT output;
        // Los cuatros vértices generados por partícula comparten el mismo identificador de instancia.
        // Este se utilizará como índice en el arreglo que posee los índices de las partículas ordenadas
        uint particleIndex = input.instanceid;

        // Se obtiene el índice global de la partícula
        uint index = (uint)g_SortedIndexBuffer[NumeroActualDeElementos - particleIndex - 1].y;

        // Se obtiene la posición de la partícula
        float3 center = SimulationState[index].position;

        // Se obtiene la posición de una de las esquinas del futuro billboard utilizando el identificador
        // del vértice actual
        float3 position = g_TessPositions[input.vertexid] * ParticleSize;
        position = mul(position, (float3x3)g_mInvView) + center;
        output.position = position;

        // Se obtiene la coordenada de textura del vértice actual
        output.texcoords = g_TessTexcoords[input.vertexid];

        // Se obtiene la normal del vértice de forma análoga a la empleada en el procesador de geometría
        // …

        // Se envía el vértice al procesador de cascaras
        return output;
}
```

Algoritmo 9. Código utilizado dentro del procesador de vértices en el flujo de trabajo con teselado

Seguidamente, en el procesador de cáscara se determina el número de vértices adicionales que serán introducidos al *billboard* de acuerdo a la distancia actual entre la posición de la partícula procesada en espacio de mundo y el punto de vista de la escena. En nuestra aplicación la mayor cantidad vértices a la que se puede incrementar un *billboard* es 32 elementos. En el Algoritmo 10 se puede estudiar el código utilizado dentro de este procesador.

```
HS_CONSTANT_DATA_OUTPUT ConstantHS(InputPatch<VS_CONTROL_POINT_OUTPUT, 4> ip, uint PatchID : SV_PrimitiveID)
{
        HS_CONSTANT_DATA_OUTPUT Output;
        // Gracias a que las partículas son representadas en forma de billboards, el programa solo necesita saber el
        // tamaño de las partículas y la distancia que poseen al punto de vista de la escena para decidir el factor
        // de teselación
        float size = ParticleSize;

        // Se calcula la distancia que posee la partícula con el punto de vista
        float view_dist = length(CameraPosition - ip[0].WorldPos);

        // Se determina el factor de teselado
        float tess = clamp(g_TessellationDensity * size / view_dist, 1.0, g_MaxTessellation);

        // Se inserta el factor de teselado para su procesamiento automatizado
        Output.Edges[0] = tess; Output.Edges[1] = tess; Output.Edges[2] = tess; Output.Edges[3] = tess;
        Output.Inside[0] = tess;Output.Inside[1] = tess;

        return Output;
}
```

Algoritmo 10. Código utilizado dentro del procesador de cascaras

Finalmente, en el procesador de dominio se procesan todos los vértices del *billboard* teselado. Es en este procesado a donde se transfieren todos los cálculos asociados a la iluminación y sombreado previamente ubicados en el procesador de vértices del sistema. El incremento de frecuencia de muestreo del mapa de sombras producido por este procesador permite un incremento de la precisión de las áreas sombreadas proporcional al factor de teselado alcanzado por cada elemento del sistema. En el Algoritmo 11 se expone el segmento del código asociado al procesador de dominio que difiere del previamente visto del procesador de vértices a comienzos de esta sección.

```
[domain("quad")]
PS_INPUT DSMAIN(HS_CONSTANT_DATA_OUTPUT input,float2 UV : SV_DomainLocation, const OutputPatch<HS_OUTPUT, 4> quad)
{
        PS_INPUT output = (PS_INPUT) 0;

        // Se interpola entre las esquinas establecidas en el billboard para determinar la posición final
        // del vértice actual en espacio de mundo
        float3 verticalPos1 = lerp(quad[0].position, quad[1].position, UV.y);
        float3 verticalPos2 = lerp(quad[3].position, quad[2].position, UV.y);

        // Esta es la posición que finalmente es utilizada para la evaluación de sombreado
        float3 finalPos = lerp(verticalPos1, verticalPos2, UV.x);
        output.position = mul(float4(finalPos, 1), g_mWorldViewProj);

        // Se interpola entre las esquinas establecidas en el billboard para determinar la coordenada de
        // textura del vértice actual
        float2 verticalTex1 = lerp(quad[0].texcoords, quad[1].texcoords, UV.y);
        float2 verticalTex2 = lerp(quad[3].texcoords, quad[2].texcoords, UV.y);
        float2 finalTex = lerp(verticalTex1, verticalTex2, UV.x);
        output.texcoords = finalTex;

        // Se interpola entre las esquinas establecidas en el billboard para determinar la normal del vértice
        // actual
        float3 verticalNorm1 = lerp(quad[0].normal, quad[1].normal, UV.y);
        float3 verticalNorm2 = lerp(quad[3].normal, quad[2].normal, UV.y);
        float3 finalNormal = lerp(verticalNorm1, verticalNorm2, UV.x);
        output.normal = finalNormal;

        // Se efectúa la evaluación de sombreado e iluminación previamente estudiada
}
```

Algoritmo 11. Código utilizado dentro del procesador de dominio

Posterior a estos cálculos, se hace uso del mismo procesador de vértices ilustrado previamente para proporcionar a los pixeles de las partículas de su coloración final.

3.5 Proyección de sombras

Para habilitar la proyección de sombras por parte de las partículas sobre la geometría de la escena se implementaron algoritmos que se encargan de generar y utilizar mapas de sombra especializados que almacenan tanto la información de profundidad como los valores de transparencia de las partículas desde la perspectiva de cada una de la luces de la escena. Una representación gráfica de este tipo de mapas puede ser visualizada en la Figura 20.

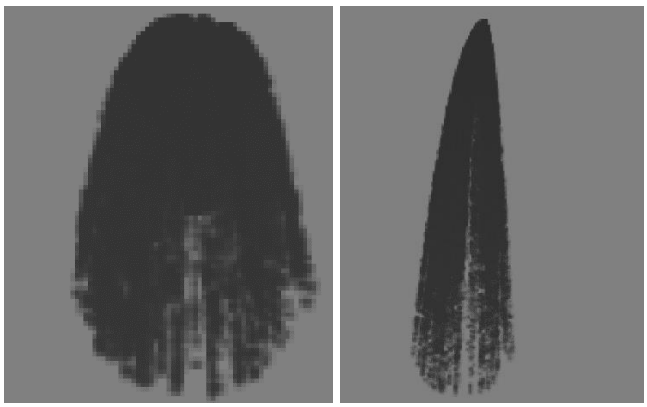


Figura 20. Ejemplo de mapas de sombras con canal dedicado a la acumulación de transparencia de las partículas.

Para la generación de este mapa se utilizó una metodología similar a la empleada durante el despliegue básico del sistema de partículas, con la diferencia de que dentro del procesador de geometría se utilizan como matriz inversa de vista y matriz de mundo-vista-proyección a aquellas asociadas a las luces de la escena a fin de producir *billboards* orientados a dichas luces. Adicionalmente, dentro del procesador de fragmentos el despliegue de los píxeles de las partículas es redirigido a un blanco de renderizado de dos canales que almacena tanto los valores de profundidad como la transparencia de las partículas mediante el código expuesto en el Algoritmo 12.

```
float2 PSMAIN(in PS_INPUT input) : SV_TARGET
{
        Float2 shadowmap = float2(0, 0);

        // Se almacena el valor de profundidad del fragmento actual desde la perspectiva de la luz siendo
        // procesada
        shadowmap.x = input.depthPosition;

        // Se muestrea la textura de la partícula a fin de obtener la transparencia del fragmento actual
        float ColorMapAlpha = ColorMap.Sample(samplerPoint, input.texcoords).a;

        // Antes de guardar la transparencia, se multiplica por  el valor de transparencia de las partículas
        // establecido por el usuario
        shadowmap.g = ColorMapAlpha * input.color.a;

        // Se imprime la información en el blanco de renderizado que actuará como mapa de sombras especializado
        return shadowmap;
}
```

Algoritmo 12. Código utilizado para la generación de mapas de sombras con traslucidez

A fin de emplear estos mapas, se modificaron las pruebas de sombreado utilizadas durante el despliegue de los objetos modelados tradicionalmente con el propósito de ejecutar un muestreo de textura adicional en donde se extrae la transparencia contenida en el mapa de sombras. Si se determina que el fragmento de la geometría siendo procesada se encuentra ocluido por el fragmento de una partícula, este valor de transparencia es utilizado para determinar la intensidad del sombreado a realizar. Este procedimiento es ejecutado en el procesador de fragmentos por el código expuesto en el Algoritmo 13.

```
// Se obtiene la posición del fragmento desde la perspectiva de la luz siendo procesada
float4 lightViewPosition = GetShadowLightPosition(input, i);

// Se transforma la posición a coordenadas de textura
projectTexCoord.x = lightViewPosition.x / lightViewPosition.w / 2.0f + 0.5f;
projectTexCoord.y = -lightViewPosition.y / lightViewPosition.w / 2.0f + 0.5f;

// Se normaliza la profundidad del fragmento desde la perspectiva de la luz
float lightDepthValue = lightViewPosition.z / lightViewPosition.w;

// Se aplica un pequeño bias al valor anterior a fin de minimizar artefactos visuales
lightDepthValue = lightDepthValue - shadowMapBias / 100.0f;

// Se determina si las coordenadas proyectadas se encuentran en el rango [0,1]. Si lo están, el
// píxel es visible por la luz y las sombras deben ser procesadas
if ((saturate(projectTexCoord.x) == projectTexCoord.x) && (saturate(projectTexCoord.y) == projectTexCoord.y))
{
        // Se efectúan las operaciones asociadas a la evaluación de sombreado entre geometría tradicional

        // Se obtiene el valor de profundidad y transparencia del mapa de sombras de la escena que coincide con
        // las coordenadas de la partícula siendo procesada
        float2 particleShadowMapDepthTrans = GetParticleShadowMapDepth(i, projectTexCoord);

        // Si la profundidad de la partícula es mayor al valor contenido en el mapa de sombras, se considera
        // a la partícula en sombra y se procede a determinar la intensidad de la misma
        if (particleShadowMapDepthTrans.r < lightDepthValue)
                // La intensidad de la sombra es definida por la intensidad de la transparencia de los fragmentos
                // de partículas ocluyendo al fragmento de la geometría (mientras menor transparencia, mayor
                // intensidad) y la intensidad de las sombras producidas por la luz siendo procesada
                shadowFactor = max(0, shadowFactor - (saturate(particleShadowMapDepthTrans.g) *
                                                      ShadowIntensity[i].x));

}
```

Algoritmo 13. Código utilizado para la evaluación de sombreado mediante el uso de mapas de sombras con traslucidez

3.6 Interacción entre partículas

El sistema de partícula emplea una de dos implementaciones en GPGPU completamente separadas dependiendo de la metodología que el usuario desea utilizar para simular la interacción entre las partículas del sistema. En el caso de la alternativa basada en la resolución simple de la problemática de N-Cuerpos, cada partícula del sistema es evaluada contra todas las otras, independientemente de la distancia que exista entre ellas. Para lograr esto, cada partícula es procesada en un ciclo que añade a un vector de velocidad final el resultado de la evaluación de la interacción de la partícula con el resto.

La evaluación de la interacción entre dos partículas es efectuada mediante una implementación del cálculo de potencial gravitacional entre dos cuerpos. En este cálculo se determina la influencia gravitacional que un cuerpo ejerce sobre otro, atrayéndolo o repeliéndolo según el tipo de interacción deseada por el usuario. En el caso de la dinámica de atracción, la dirección del vector de fuerza resultante es el vector de dirección que va desde el resto de las partículas del sistema (partícula A) hacia la partícula sobre la cual se está efectuando el estudio en el hilo actual (partícula B). La magnitud de este vector es determinado mediante la ecuación [21]

$$m_1\mathbf{a}_1 = \frac{Gm_1m_2}{r_{12}^3}(\mathbf{r}_2 - \mathbf{r}_1)$$

donde, m_1 y m_2 son las masas de dos partículas estudiadas, r_2 y r_1 sus posiciones, r_{12} la distancia existente entre ambas y G la constante de gravitación universal 6.673×10^{-11} $N(m/kg)^2$. El algoritmo utilizado para implementar estos cálculos en la GPU es el expuesto en el Algoritmo 14.

```
float3 bodyBodyInteraction(float3 posicionParticulaA, float3 posicionParticulaB)
{
	// Se obtiene el vector direccional que va desde la partícula A hasta la partícula B
	float3 direccion = posicionParticulaA.xyz - posicionParticulaB.xyz;

	// Se calcula la magnitud de la interacción entre las dos partícula basándose en la
	// distancia entre ellas y sus respectivas masas
	float distSqr = dot(direccion, direccion)
	distSqr =+ softeningSquared;
	float invDist = 1.0f / sqrt(distSqr);
	float invDistCube = invDist * invDist * invDist;
	float magnitud = ConstanteGravitacional * masaParticula * invDistCube;

	// Se multiplica la dirección por la magnitud para obtener el vector final de la
	// interacción
	if (Attract)
		return direccion * magnitud; // Atracción
	else
		return -direccion * magnitud;// Repulsión
}
```

Algoritmo 14. Código utilizado para la calcular la interacción entre dos cuerpos

La segunda alternativa utiliza el mismo algoritmo para determinar la interacción entre dos partículas del sistema, sin embargo, difiere en la metodología empleada para seleccionar cuales elementos del sistema de partículas interactúan entre sí al utilizar una rejilla dinámica tridimensional que limita la interacción de los elementos que se encuentran dentro de ella.

Para hacer uso de esta rejilla se realiza el siguiente procedimiento: en primer lugar se debe determinar en cual cuadricula se encuentra cada partícula considerando que las partículas que ocupan la misma cuadricula compartirán el mismo identificador. Una simple combinación de los componentes de la posición de las cuadriculas son utilizadas para identificarlas. Esta información conjunto el identificador de las partículas procesadas son almacenados en un búfer para el próximo paso.

Posteriormente se necesitan ordenar los búferes de identificadores de rejillas e identificadores de partículas. Esto tiene el efecto de organizar todas las partículas que ocupan la misma cuadricula una junta a la otra en la memoria. Para este proceso de ordenamiento se usa el algoritmo Bitonic Sort estudiado en capítulos anteriores.

A pesar de que en este punto se ha construido una lista organizada de cada partícula con el identificador de cuadricula que ocupa, aún se carece de una forma de utilizar esa información para indexar la rejilla dinámica. El

próximo paso es buscar en la lista las ubicaciones de inicio y final de cada cuadricula y guardar dicha información. Para hacer esto, se ejecuta un hilo por elemento de la lista. Si el elemento a la izquierda de la lista tiene un identificador de rejilla diferente, entonces se puede asegurar que el elemento siendo procesado es el primer ocupante de la celda a la que pertenece gracias al proceso de ordenamiento efectuado en el paso anterior. Similarmente, si el elemento a la derecha posee un identificador de rejilla diferente, entonces el elemento actual es el último elemento de la rejilla a la cual pertenece. Mediante el almacenamiento del índice final e inicial de cada identificador de rejilla se puede acceder fácilmente a todas las partículas en cualquier rejilla durante el próximo paso.

Finalmente se efectúa la simulación. Como se mencionó previamente el algoritmo de interacción entre partículas no es alterado de ninguna forma; sin embargo, esta vez solo se necesita buscar partículas consideradas vecinas potenciales encontradas en cuadriculas adyacentes. Para hacer esta evaluación solo se necesita determinar el identificador de cuadricula a la que pertenece la partícula actual y los identificadores de las cuadriculas vecinas. Empleando esta información se pueden buscar las partículas que pertenecen a dichas cuadriculas usando los índices construidos en el paso anterior. Posteriormente se comprueba si la partícula se encuentra dentro de un radio de interacción personalizable por el usuario y, en caso afirmativo, se aplica el algoritmo de interacción entre partículas.

3.7 Colisiones entre partículas y la escena

A fin de implementar las colisiones de las partículas con el escenario en el que se encuentran mediante cálculos basados en espacio de pantalla, fue imperante el ajuste del esquema de trabajo utilizado para desplegar los objetos tridimensionales modelados tradicionalmente. Para la implementación simple que hace uso de un solo mapa de profundidad para calcular las colisiones en espacio de pantalla se modificó el programa encargado del procesamiento de los fragmentos de la geometría tradicional para que almacene, en un blanco de despliegue adicional al utilizado para el despliegue de imágenes por el monitor, la información de profundidad desde la perspectiva del punto de vista de la escena. La Figura 21 ilustra una representación gráfica de un mapa de profundidad obtenido mediante este procedimiento.

Figura 21. Representación gráfica de un mapa de profundidad utilizado para los cálculos de colisiones en espacio de pantalla

Para la implementación que hace uso de mapas de profundidad cúbicos el procedimiento de generación es similar al previamente expuesto. La principal diferencia radica en que el procedimiento es repetido seis veces, una vez por cada cara del cubo de mapas de profundidad a generar. El programa encargado del despliegue del mapa cúbico recibe seis matrices de vista diferentes, cada una orientada en una dirección distinta. Estas direcciones son: arriba, abajo, izquierda, derecha, adelante y atrás de la posición de la cámara de la escena. Es importante señalar que para que el mapa cúbico realmente abarque todo el escenario alrededor de la cámara, cada mapa de sombras debe ser generado con un campo de visión de 90 grados.

Independientemente del mecanismo de generación de mapa de profundidad utilizado, los cálculos de colisión en espacio de pantalla son ejecutados dentro del programa de la GPU encargado de la actualización de la posición de las partículas. El algoritmo inicia proyectando la posición de cada partícula a espacio de clip para obtener su

profundidad en este espacio. Posteriormente se evalúa si este valor se encuentra más lejos de la cámara que el valor profundidad del mapa utilizado que coincide con las coordenadas del centro de la partícula siendo procesada, en caso afirmativo se considera que la partícula está en colisión y su movimiento debe ser alterado.

Para calcular el nuevo movimiento de la partícula producto de una colisión se deben efectuar dos búsquedas adicionales en el mapa de profundidad de la escena, cada una a un pixel de desplazamiento del pixel actual. Estos valores son utilizados mediante un producto cruzado para derivar una normal en espacio de vista. La combinación de la profundidad y esta normal forman un plano. Seguidamente se transforma el vector de velocidad de la partícula a espacio de vista y se refleja y amortigua usando el plano previamente generado. Para culminar se transforma el vector de velocidad alterado de vuelta a espacio de mundo y se almacena en la partícula. El código del procedimiento previamente explicado es desarrollado en el Algoritmo 15.

```
float4 mv_pos = mul(float4(position, 1.0), g_mMV); // Se obtiene la posición de la partícula en espacio de vista
float4 clip_pos = mul(mv_pos, g_mProjection);      // Posición de la partícula en espacio de proyección
float4 nclip_pos = float4(clip_pos.xyz / clip_pos.w, clip_pos.w); // Coordenadas normalizadas de dispositivo
float2 proj_tc0 = float2(.5f * nclip_pos.x + .5f, -.5f * nclip_pos.y + .5f);// Coordenadas de textura [0,1]

// Se obtiene de nuevo la  posición de la partícula en espacio de vista, pero esta vez se inserta en el
// componente Z el valor de profundidad del mapa que coincide con las coordenadas de textura de la partícula
float3 sceneViewSpace0 = viewSpaceValueForCoord(proj_tc0);

// Se determina si se produjo colisión con el escenario: la primera línea verifica que las partículas estén
// dentro del punto de vista de la escena, la segunda comprueba que  la partícula está más lejana
// de la cámara que la superficie en el mapa de profundidad, y la última comprueba que también se
// encuentra dentro del umbral definido por el usuario
if (proj_tc0.x > 0.0 && proj_tc0.x < 1.0 && proj_tc0.y > 0.0 && proj_tc0.y < 1.0 &&
    sceneViewSpace0.z < mv_pos.z &&
    sceneViewSpace0.z > (mv_pos.z - UmbralAndCubicMap.x) && sceneViewSpace0.z > 0.1)
{
        // Se obtienen las dimensiones del mapa de profundidad
        SceneDepthMap.GetDimensions(width, height);

        // Se obtiene la normal de la superficie en espacio de mundo.
        float2 eps = float2(1.0 / width, 1.0 / height);
        float2 proj_tc1 = proj_tc0 + float2(eps.x, 0.0); // Coordenada desfasada un pixel en X
        float2 proj_tc2 = proj_tc0 + float2(0.0, eps.y); // Coordenada desfasada un pixel en Y
        float3 sceneViewSpace1 = viewSpaceValueForCoord(proj_tc1);
        float3 sceneViewSpace2 = viewSpaceValueForCoord(proj_tc2);
        float3 normal = normalize(cross(sceneViewSpace1 - sceneViewSpace0, sceneViewSpace2 - sceneViewSpace0));
        normal = mul(normal, g_mInvMV);

        // Se refleja el vector de velocidad de la partícula
        float3 reflected = reflect(velocity, normal);

        // Se finaliza calculando la posición luego de la colisión y el nuevo vector de velocidad suavizado
        position += normal * 1e-4;
        velocity = reflected * 0.7 + cos(position * 1e5) * 5e-4;
}
```

Algoritmo 15. Código utilizado para la evaluación de colisiones en espacio de pantalla

En la implementación que hace uso de mapas de profundidad cúbicos el procedimiento es idéntico al anterior, con la excepción que de forma previa a cualquier cálculo se debe determinar la cara del mapa cúbico en el cual se encuentra la partícula siendo procesada, la cual será utilizada durante el muestreo del mapa de profundidad. Para esta evaluación se utiliza el código expuesto dentro del Algoritmo 16.

```
// El valor de entrada es el vector direccional que va desde la cámara hacia la partícula
int getCubeMapFace(float3 direction){
        float3 absDirection = abs(direction);
        if (absDirection.x > absDirection.y && absDirection.x > absDirection.z)
                return step(direction.x, 0.0); // La cara es la derecho o izquierda
        else if (absDirection.y > absDirection.x && absDirection.y > absDirection.z)
                return 2.0 + step(direction.y, 0.0); // La cara es la superior o anterior
        else if (absDirection.z > absDirection.x && absDirection.z > absDirection.y)
                return 4.0 + step(direction.z, 0.0); // La cara es la delantera o trasera
        else
                return -1.0; // La cara es indeterminable
}
```

Algoritmo 16. Código utilizado para determinar la cara del mapa cúbico en el que se encuentra la partícula estudiada

Capítulo 4: Pruebas y resultados

4.1 Descripción del entorno de pruebas

Todos los experimentos realizados en este trabajo de investigación fueron ejecutados en un computador de escritorio. Las características del equipo utilizado son las siguientes:

- Procesador AMD Phenom II X6 1090T 3.2 GHz
- 8 GB de memoria RAM DDR3
- Disco duro de 500 Gb
- Tarjeta aceleradora de gráficos NVIDIA 650 GTX TI 2GB
- Sistema operativo Windows 7 64 bit

4.2 Pruebas

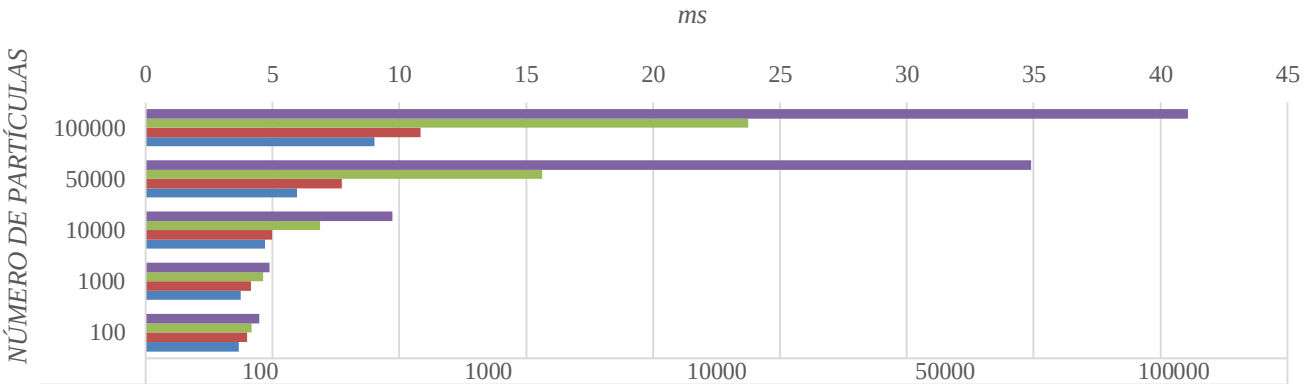
En el desarrollo de las pruebas expuestas en el presente capítulo se evaluaron características tanto cuantitativas como cualitativas del sistema de partículas implementado. En el aspecto cuantitativo, la característica más frecuentemente estudiada es el desempeño de los módulos de la aplicación en relación a la duración en milisegundos (*ms*) que la aplicación tardaba en procesar cada cuadro de animación. Durante las pruebas donde se exponen resultados cualitativos se efectuaron análisis sobre las características de las imágenes y dinámicas producidas por la aplicación. Todas estas pruebas fueron ejecutadas en una ventana de tamaño de *512* x *512* píxeles con partículas que ocupaban un tamaño de *0.6* x *0.6* unidades en espacio de vista.

Para medir la duración del proceso de despliegue de los cuadros de animación, al inicio de la función asociada al proceso de despliegue del sistema de partículas, se inicializa un contador de milisegundos, y como última instrucción de aquella función se detiene el contador para obtener la duración final del cuadro procesado.

Finalmente, para efectuar los análisis de desempeño sobre un módulo especializado en particular se desactivaron los demás módulos y se ejecutó el sistema de partículas con el contador de milisegundos previamente explicado.

4.2.1 Relación entre el número de luces y el desempeño del módulo de coloración

Para estudiar la relación entre el número de luces de la escena y el desempeño del módulo especializado en la coloración de las partículas se sometió al sistema a la influencia de una cantidad variable de luces y se captó la duración del proceso de despliegue del sistema de partículas en un cuadro de animación. Para cada caso de estudio de desempeño bajo una cantidad de luces se repitió el estudio utilizando cantidades variables de partículas a fin de obtener un panorama global de la relación entre el número de partículas, la cantidad de luces y el desempeño del módulo de coloración.



	100	1000	10000	50000	100000
4 Luces	4.47	4.88	9.73	34.9	41.08
3 Luces	4.16	4.63	6.87	15.63	23.74
2 Luces	3.99	4.15	4.98	7.74	10.84
1 Luz	3.67	3.75	4.69	5.96	9.01

Tabla 1. Rendimiento medido en milisegundos del módulo de coloración ejecutado con diferentes cantidades de luces y partículas

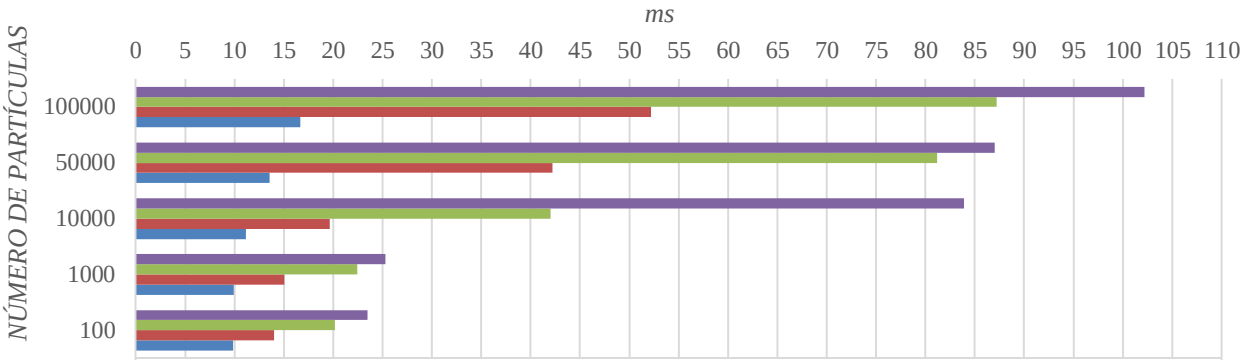
Tal como muestra la tabla 1, independientemente de la cantidad de partículas, el desempeño del módulo decremento mientras más luces eran procesadas. Esto es consistente con el hecho de que todos los cálculos de acumulación de iluminación entrante efectuados en el procesador de vértices deben ser repetidos por cada luz que se encuentre en la escena, efectivamente incrementando la cantidad de operaciones y tiempo necesario para desplegar cada partícula.

En la misma tabla se puede observar el efecto de la variación del número de partículas procesadas, donde al incrementar el número de elementos también aumenta la cantidad de información que debe ser procesada en los programas de vértices, geometría y fragmentos encargados del despliegue del sistema. Esto provoca que, sin importar la cantidad de luces que estén siendo procesadas, el desempeño de la aplicación se vea impactado negativamente a mayor cantidad de partículas. A ello se suma el hecho que todos los fragmentos de cada partícula deben ser procesados debido al componente de transparencia manejado, lo cual evita la posibilidad de oclusión mutua entre las partículas por las razones explicadas en capítulos previos (i.e. sección 3.2.4).

Finalmente, se puede observar que el módulo de coloración es capaz de ejecutarse en tiempo real de forma fluida (duración de 33.3 ms que se traduce en más 30 cuadros por segundo) solo si no excede el escenario establecido por 10000 partículas y 4 luces por escena.

4.2.2 Relación entre el número de luces y el desempeño del módulo de proyección de sombras

Para analizar la relación entre el número de luces de la escena y el desempeño del módulo especializado en la proyección de sombras por parte de las partículas se sometió al sistema a la influencia de una cantidad variable de luces y se captó la duración del proceso de despliegue del sistema de partículas en un cuadro de animación.



	100	1000	10000	50000	100000
4 Luces	23.48	25.29	83.9	87	102.2
3 Luces	20.2	22.46	42.02	81.18	87.2
2 Luces	14.01	15.07	19.67	42.23	52.21
1 Luz	9.85	9.93	11.18	13.59	16.66

Tabla 2. Rendimiento medido en milisegundos del módulo de proyección de sombras ejecutado con diferentes cantidades de luces y partículas

A partir de la información de la tabla 2 se puede inferir que el desempeño del módulo decremento mientras más luces eran procesadas. Este comportamiento es coherente con el hecho de que todas las partículas del sistema deben ser desplegadas una vez por cada luz de la escena durante la etapa de generación de mapas de sombras de traslucidez, implicando la repetición de todos los cálculos en los procesadores de vértices, geometría y fragmentos relacionados a este proceso. Adicionalmente, cada fragmento de la geometría de la escena debe hacer el proceso de muestrear cada mapa de sombreado de dichas luces para poder recibir las sombras proyectadas por las partículas. El aumento de la cantidad de las luces en escena intensifica el impacto negativo sobre el desempeño del módulo producido por la combinación las dos circunstancias descritas previamente.

Igualmente, se puede deducir que al incrementar el número de partículas, también aumenta la cantidad de información que debe ser procesada en los programas de vértices, geometría y fragmentos encargados de la generación de los mapas de sombras con traslucidez, provocando un impacto negativo en el desempeño del módulo

estudiado. Al igual que el módulo encargado de la coloración de los fragmentos de las partículas, la oclusión mutua de elementos no es posible durante la generación de los mapas de sombras con traslucidez debido a que los mismos implican el uso del componente de transparencia, lo cual obliga el despliegue de todos los fragmentos visibles e incrementa el impacto del número de partículas sobre el desempeño del sistema.

Finalmente, se puede asegurar que el módulo de proyección de sombras es capaz de ejecutarse en tiempo real de forma fluida bajo cualquier cantidad partículas contenida en el rango estudiado y en presencia de 1 luz por escena.

4.2.3 Relación entre la complejidad de la escena y el desempeño del módulo de recepción de sombras

Para estudiar la relación entre la complejidad de la escena, en términos de cantidad de polígonos, y el desempeño del módulo especializado en la recepción de sombras por parte de la geometría en su versión sin teselado, se sometió al sistema a un conjunto de escenas que contenían diferentes cantidades de polígonos alrededor de la fuente principal de partículas y se captó la duración de proceso del despliegue del sistema de partículas en un cuadro de animación. Como escena simple e intermedia se utilizó geometría arbitraria, sin embargo, como escena compleja se utilizó un modelo del Palacio Sponza de la ciudad Dubrovnik, Croacia; específicamente la versión desarrollada por el modelador Marko Dabrovic [25]. Esta última escena es reiteradamente utilizada en las posteriores pruebas que requieren la interacción entre el sistema y geometría modelada tradicionalmente.

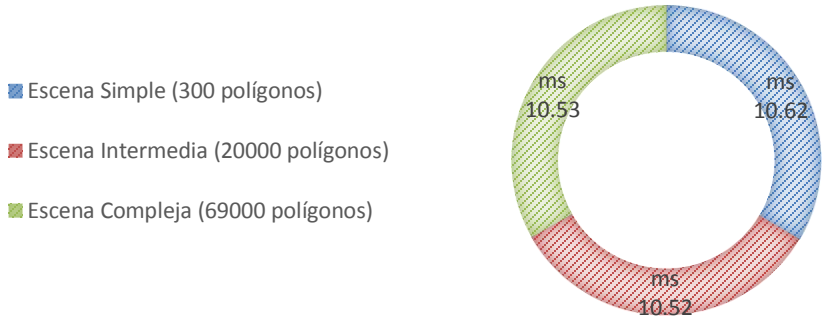
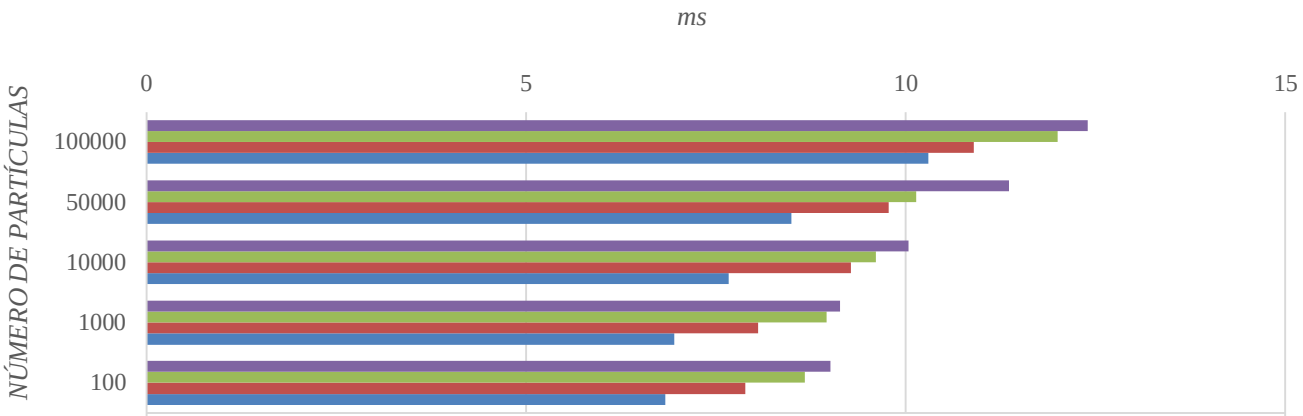


Tabla 3. Rendimiento medido en milisegundos del módulo de recepción de sombras ejecutado dentro de escenas con diferentes cantidades de polígonos

Al observar la información de la tabla 3 queda claro inmediatamente que existe una independencia entre el desempeño del módulo encargado de la recepción de sombras por parte de la geometría modelada tradicionalmente y la cantidad de polígonos que posee la escena donde se presenta el sistema de partículas. Esta independencia es lograda gracias a que la información de la geometría de la escena desde la perspectiva de la luz es calculada y almacenada en los mapas de sombras por un proceso independiente al sistema de partículas, lo cual permite que durante las evaluaciones de sombreado el sistema se pueda abstraer de la complejidad de la geometría contenida en los mapas ya que solo necesita muestrearlos para obtener los datos que requiere.

4.2.4 Relación entre el número de luces y el desempeño del módulo de recepción de sombras

A fin de estudiar la relación entre el número de luces de la escena y el desempeño del módulo especializado en la recepción de sombras producidas por geometría generada tradicionalmente se sometió al sistema a la influencia de una cantidad variable de luces y se obtuvo la duración del proceso de despliegue del sistema de partículas en un cuadro de animación. Para cada caso de estudio de desempeño bajo una cantidad de luces se repitió el estudio utilizando cantidades variables de partículas a fin de obtener un panorama global de la relación entre el número de partículas, la cantidad de luces y el desempeño del módulo de recepción de sombras.



	100	1000	10000	50000	100000
4 Luces	9.01	9.14	10.04	11.36	12.4
3 Luces	8.67	8.96	9.61	10.14	12
2 Luces	7.89	8.06	9.28	9.78	10.9
1 Luz	6.84	6.95	7.67	8.5	10.3

Tabla 4. Rendimiento medido en milisegundos del módulo de recepción de sombras ejecutado con diferentes cantidades de luces y partículas

Tal como muestra la tabla 4, el desempeño del módulo decremento mientras más luces eran procesadas. Esto es coherente con el conocimiento que todos los cálculos asociados al muestreo y evaluación de los mapas de sombras de la geometría tradicional efectuados en el procesador de vértices deben ser repetidos por cada luz que se encuentre en la escena, efectivamente incrementando la cantidad de operaciones y tiempo necesario para desplegar cada partícula.

En la misma tabla se puede observar el efecto de la variación del número de partículas procesadas, donde al incrementar el número de elementos también aumenta la cantidad de información que debe ser procesada en los programas de vértices, geometría y fragmentos encargados del sombreado de las partículas. Esto provoca que el desempeño de la aplicación se vea impactado negativamente a mayor cantidad de partículas.

Finalmente, se puede observar que el módulo de recepción de sombras es capaz de ejecutarse en tiempo real de forma fluida en todos los escenarios de prueba utilizados. A pesar de lo anterior, es importante recalcar que este estudio solo considera los tiempos de cálculo asociados a la recepción de sombras por parte del sistema de partículas, ignorando el tiempo invertido en la generación de los mapas de sombras de la geometría modelada tradicionalmente. Este procedimiento, dependiendo de la metodología utilizada, tiene el potencial de incrementar considerablemente el tiempo de cálculo necesario para el despliegue de la aplicación en general en precedencia de múltiples luces.

4.2.5 Impacto del uso de teselado en el desempeño del módulo de recepción de sombras

Con el fin de evaluar el impacto del uso del esquema de trabajo con teselado dentro del módulo de recepción de sombras se efectúo un análisis comparativo entre dos sistemas de partículas cuyo único factor divergente era el uso del teselado. Para esto, ambos sistemas de partículas fueron desplegados en una escena con una única luz y a una igual distancia del punto de vista mientras eran ejecutados con una cantidad variable de partículas; la información necesaria para el análisis comparativo fue obtenida mediante la recopilación de la duración del proceso de despliegue del sistema de partículas en un cuadro de animación. En la Figura 22 se puede observar la configuración de escena utilizada durante esta prueba, la cual consiste de aproximadamente 69000 polígonos distribuidos entre la geometría de la escena.

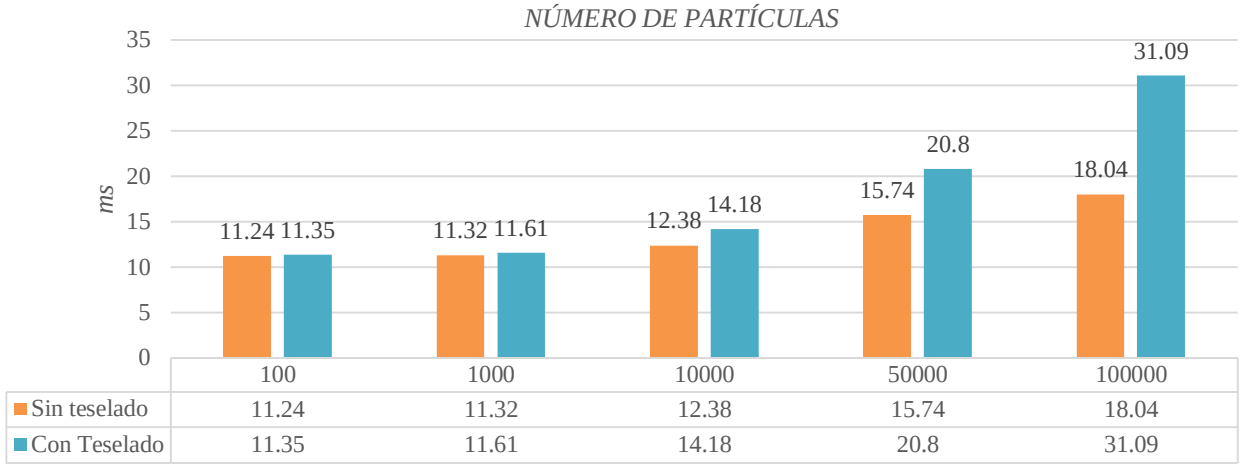


Tabla 5. Rendimiento medido en milisegundos del módulo de recepción de sombras ejecutado con diferentes cantidades de partículas en sus modalidades con y sin teselado.

Figura 22. Configuración de la escena *Palacio Sponza* (69000 polígonos) con una fuente de luz direccional y alrededor de 50000 partículas teseladas capaces de recibir sombras

A partir de la información de la tabla 5 se puede inferir que el desempeño del módulo es inferior cuando se hace uso del esquema de trabajo con teselado de partículas. Este comportamiento es congruente con el hecho de que al usar teselado, el número de vértices procesados por cada partícula sube de 4 hasta un máximo de 32 vértices dependiendo de la distancia que cada partícula posea con el punto de vista de la escena, incrementando la cantidad de veces que deben efectuarse los cálculos del proceso de muestreo de mapa de sombras y evaluación de sombreado. Sumado a ello, se debe considerar los nuevos cálculos asociados a la interpolación de los vectores de posiciones, normales y coordenadas de textura de los 4 vértices originales. Además, el esquema de trabajo por teselado implica que por cada partícula debe ejecutarse el procesador de cáscara para determinar la magnitud de teselado que se le aplicará, incrementando el tiempo de cómputo asociado al despliegue de cada partícula.

El factor de teselado utilizado para la versión del sistema que hacía uso de dicho esquema de trabajo fue de 32, mientras que se usó la topología automatizada *Integer* para distribuir los nuevos vértices entre los 4 vértices originales de los *billboards* utilizados para desplegar a las partículas. Esta topología se encarga de dividir los bordes del *quad* en segmentos exactamente iguales, dándole una apariencia uniforme. En la Figura 23 se puede observar la diferencia en los vértices que constituían los *quads* utilizados para el despliegue de las partículas durante las pruebas.

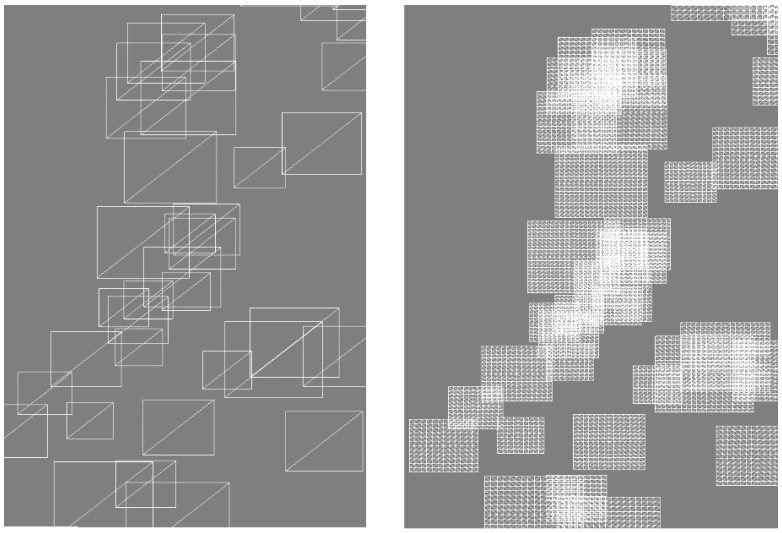
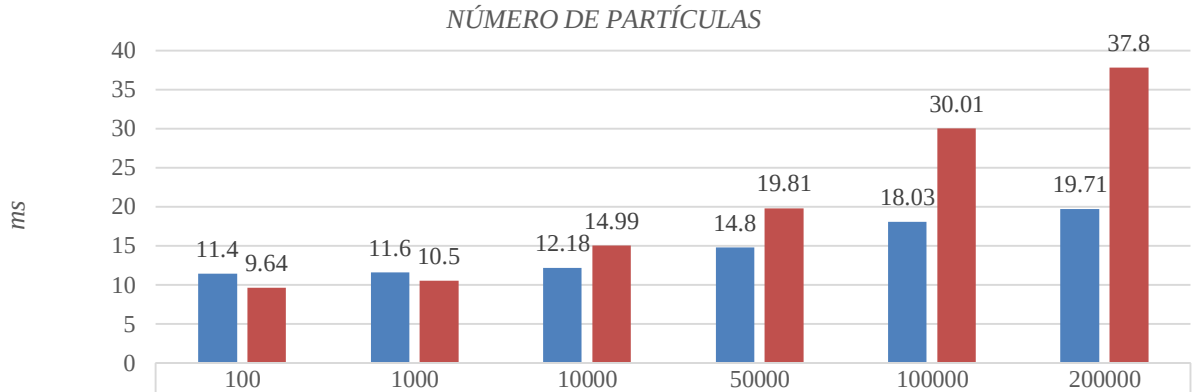


Figura 23. Partículas sin teselado (izquierda) y con teselado con factor de 32 utilizando la topología *Interger* para los nuevos vértices (derecha)

4.2.6 Relación entre el número de partículas y el desempeño del módulo de interacción entre partículas y la escena propuesto y su versión en PhysX

Para analizar la relación entre el número de partículas y el desempeño del módulo especializado en la interacción entre éstas y la geometría de las escenas donde se encuentran, se ejecutó el sistema bajo una cantidad variable de partículas y se obtuvo la duración del proceso de simulación en un cuadro de animación. A fin de generar un marco comparativo, cada caso de estudio de desempeño bajo una cantidad de partículas fue ejecutado dentro de la misma escena mediante la fase de simulación propuesta en esta investigación y mediante la implementación propuesta por PhysX.

Debido a que la simulación fue ejecutada desde un punto de vista estático, se optó por efectuar la fase de simulación que utiliza nuestra propuesta mediante el uso de mapas de profundidad simples para evitar la carga de cómputo asociada a la generación de mapas de profundidad cúbicos, aprovechando que la inmutabilidad del punto de vista evita los artefactos de trayectoria asociados con el uso de este tipo de mapas de profundidad en colisiones calculadas en espacio de pantalla.



	100	1000	10000	50000	100000	200000
Nuestra propuesta	11.4	11.6	12.18	14.8	18.03	19.71
Physx	9.64	10.5	14.99	19.81	30.01	37.8

Tabla 6. Rendimiento medido en milisegundos del módulo de interacción entre partículas y la escena ejecutado con diferentes cantidades de elementos utilizando la implementación propuesta en este documento y una implementación basada en PhysX

Tal como muestra la tabla 6, el desempeño del módulo de interacción de nuestra propuesta decremento mientras más partículas eran procesadas. Esto se asocia al hecho de que todos los cálculos relacionados al muestreo y evaluación de los mapas de profundidad de la escena, y los cálculos encargados de generar los vectores de colisión en espacio de pantalla deben repetirse por cada partícula del sistema, incrementando la cantidad de operaciones y tiempo necesario para la fase de simulación de cada partícula.

En la misma tabla se puede observar que el desempeño de la versión que hace uso de PhysX es superior solo en sistemas con menos de alrededor de 1000 partículas, punto a partir del cual nuestra propuesta empieza a superar su desempeño por un margen que aumenta a medida que incrementa el número de partículas en el sistema. Este comportamiento es debido a que mientras que el uso de mapas de profundidad para los cálculos de colisiones en espacio de pantalla abstrae a nuestra propuesta de la geometría de la escena donde se encuentra y su complejidad en términos de cantidad y ubicación de los objetos tridimensionales que la componen. Por su parte, la propuesta de PhysX debe considerar estos factores internamente para calcular la presencia de colisiones y los vectores de colisión resultantes, incrementando el tiempo necesario para su cálculo. La influencia de este factor es intensificado a medida que el número de partículas crece en el sistema, reduciendo el desempeño del módulo de interacción en un ritmo más acelerado que nuestra propuesta.

4.2.7 Estudio de la dinámica generada por el módulo de interacción entre partículas y la escena

Un resultado visual de la dinámica producida por la implementación de colisiones en espacio de pantalla desarrollada en el presente trabajo es observable en la Figura 24. En esta secuencia de imágenes se puede observar una partícula exclusivamente trasladada por la influencia del vector de gravedad en el intervalo comprendido por los cuadros 1 y 3.

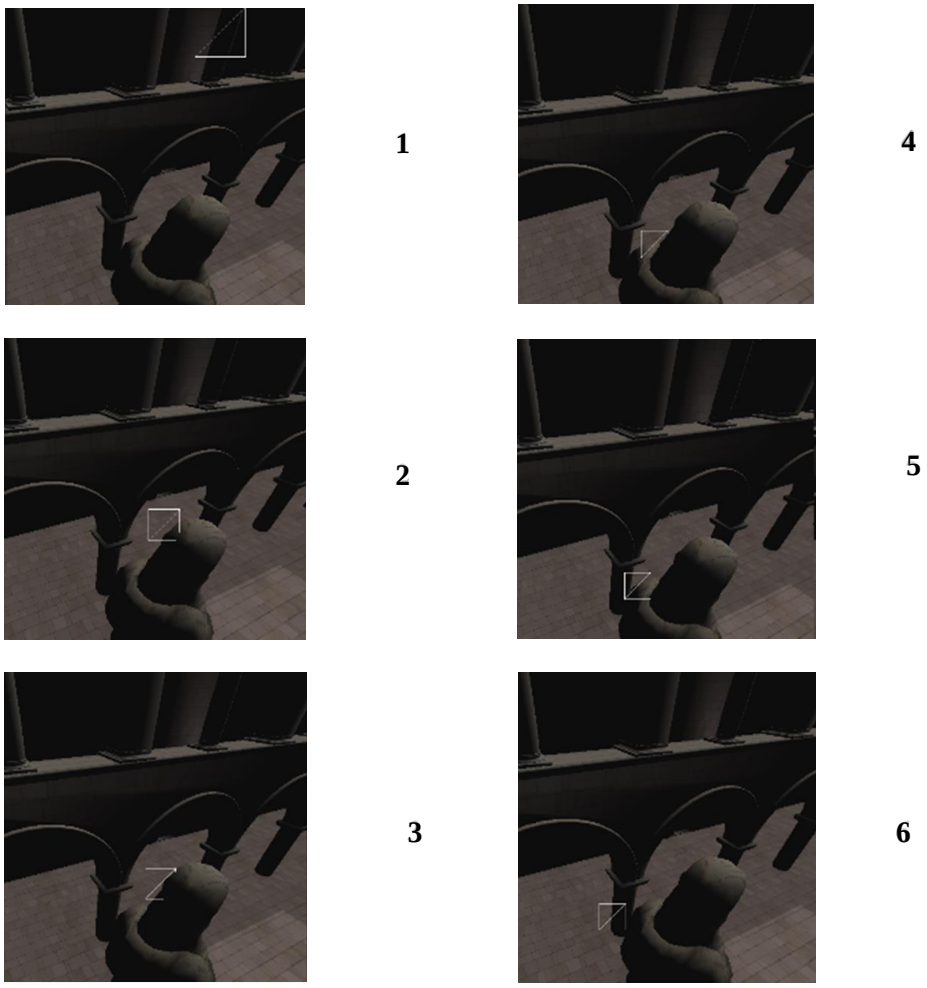
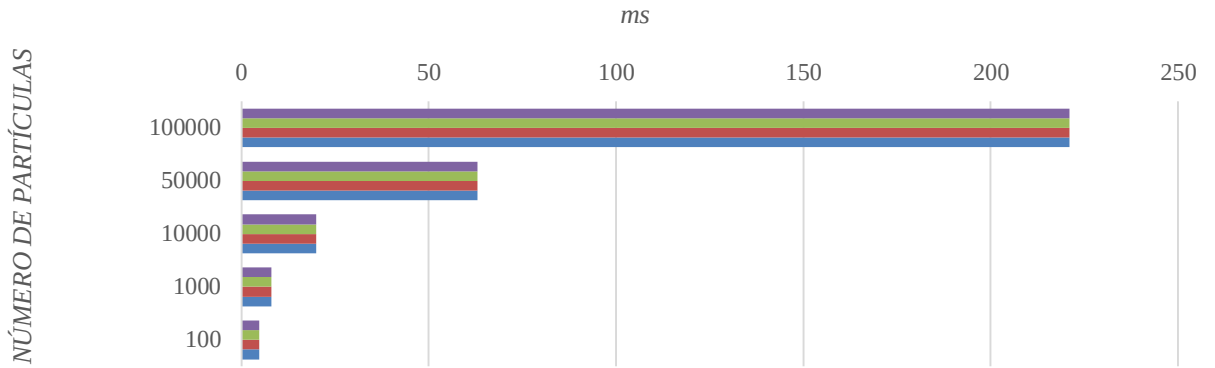


Figura 24. Secuencia de fotogramas de la colisión de una partícula con geometría arbitraria utilizando cálculos en espacio de pantalla

Posteriormente, se puede observar en el cuadro 4 el instante aproximado en el que el sistema determina que el centro de la partícula ha entrado en contacto con la superficie más cercana al punto de vista según el mapa de profundidad de la escena generado previamente. En este momento el sistema muestrea nuevamente el mapa de profundidad para calcular el nuevo vector de colisión de la partícula, utilizándolo para calcular su nuevo vector de velocidad. Para finalizar, en el intervalo comprendido entre los cuadros 5 y 6 se puede observar la nueva trayectoria de la partícula posterior a su colisión con la geometría. Esta trayectoria es producida tanto por el vector de gravedad que continúa moviendo la partícula hacia el suelo de la escena, como por el vector de colisión calculado que intentó alejar a la partícula de la geometría con la cual colisionó.

4.2.8 Análisis comparativo del desempeño alcanzado por las múltiples implementaciones del módulo de interacción entre partículas

A fin de estudiar la relación entre el número de partículas y el desempeño del módulo encargado de la interacción entre partículas mediante el uso de la metodología basada en la resolución directa de la problemática de N-Cuerpos, se sometió al sistema a una cantidad variable de partículas y se obtuvo la duración de la fase de simulación del sistema de partículas en un cuadro de animación. Para cada caso de estudio de desempeño bajo una cantidad de partículas se repitió el estudio utilizando unidades variables para las masas de las partículas a fin de obtener un panorama global de la relación entre el número de partículas, sus masas y el desempeño del módulo de interacción entre partículas.



	100	1000	10000	50000	100000
Masa 0.005 unidades	4.73	8.03	20.03	63.02	221.05
Masa 0.003 unidades	4.73	8.03	20.03	63.02	221.05
Masa 0.002 unidades	4.73	8.03	20.03	63.02	221.05
Masa 0.001 unidades	4.73	8.03	20.03	63.02	221.05

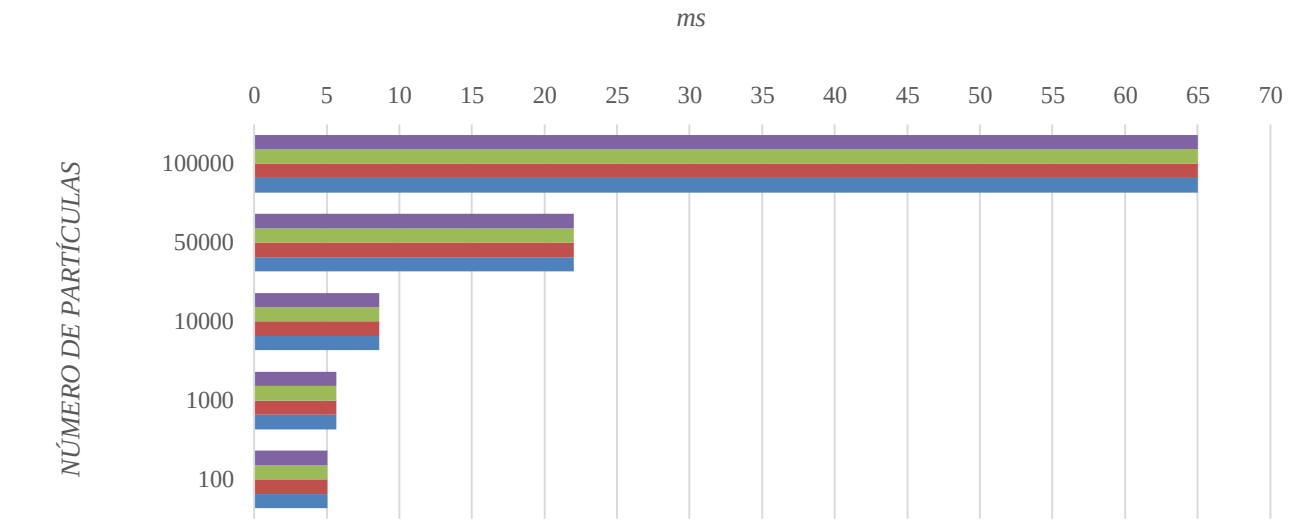
Tabla 7. Rendimiento medido en milisegundos del módulo de interacción entre partículas utilizando diversas cantidades de partículas y masas bajo la solución directa a la problemática de N-Cuerpos

Tal como muestra la tabla 7, el desempeño del módulo decremento mientras más partículas eran procesadas. Nótese que el algoritmo de resolución directa de la problemática de N-Cuerpos es un problema de complejidad $O(N^2)$, en el cual cada partícula del sistema debe evaluarse contra cada una del resto. Esto produce que al incrementar la cantidad de partículas en el sistema, también se aumente en órdenes de magnitud la cantidad de cálculos y tiempo asociado a la evaluación de la interacción de todas las partículas.

En la misma tabla se puede observar que la variación de la masa de las partículas no posee efecto alguno sobre el desempeño del módulo estudiado. Esto es debido a que aunque las masas de las partículas tienen una incidencia directa en las trayectorias generadas por la interacción, en el algoritmo utilizado únicamente son un parámetro que no posee correlación alguna con el número o frecuencia de las operaciones requeridas para determinar la interacción entre las partículas del sistema.

Adicionalmente, se puede observar que en el entorno de pruebas utilizado, el módulo de interacción entre las partículas es capaz de ejecutarse en tiempo real de forma fluida únicamente en los sistemas cuya cantidad de partículas es inferior o se encuentra alrededor de las 10000 unidades.

Para estudiar la relación entre el número de partículas y el desempeño de este módulo en su versión que hace uso de la metodología basada en el uso de rejillas dinámicas, se sometió al sistema al mismo procedimiento utilizado durante la evaluación del módulo en su versión que hace uso de la resolución directa de la problemática de N-Cuerpos.



	100	1000	10000	50000	100000
Masa 0.005 unidades	5.03	5.65	8.63	22.03	65.01
Masa 0.003 unidades	5.03	5.65	8.63	22.03	65.01
Masa 0.002 unidades	5.03	5.65	8.63	22.03	65.01
Masa 0.001 unidades	5.03	5.65	8.63	22.03	65.01

Tabla 8. Rendimiento medido en milisegundos del módulo de interacción entre partículas utilizando diversas cantidades de partículas y masas mediante el uso de rejillas dinámicas

Con los resultados de la tabla 8 se observa que al igual que la metodología basada en la resolución directa de la problemática de N-Cuerpos, el desempeño del módulo fue decayendo mientras más partículas eran procesadas. Sin embargo, al analizar ambas tablas se puede discernir que en todos los casos estudiados el desempeño de la presente metodología es superior a la previa. Esta diferencia es producida debido al uso de la rejilla dinámica para limitar la cantidad de elementos con los que cada partícula interactúa, efectivamente reduciendo la cantidad de operaciones y tiempo asociado a la fase de interacción entre partículas. A pesar de que esta técnica conlleva un tiempo de procesamiento adicional asociado a la generación de la rejilla dinámica y su uso, la reducción de los cálculos a órdenes inferiores de complejidad $O(N^2)$, provoca una ganancia de desempeño significativa en comparación a la metodología anterior.

Es importante observar que a pesar del superior desempeño de esta propuesta, el incremento de partículas sigue afectando seriamente los tiempos de cómputo del sistema; dado que cada partícula incrementa la cantidad de elementos que deben ser estudiados, esta propuesta tiene la problemática de que el desempeño es dependiente de la densidad de partículas que se encuentran en cada cuadrícula de la rejilla dinámica en un momento dado. Así, si el tamaño de las cuadrículas de las rejillas es muy amplio y/o las partículas del sistema se encuentran muy juntas, los tiempos de cómputo se empezarán a acercar a aquellos utilizados por la propuesta de resolución simple de la problemática de N-Cuerpos.

Finalmente, se puede observar que el módulo de interacción entre las partículas basado en el uso de rejillas dinámica es capaz de ejecutarse en tiempo real de forma fluida en los sistemas cuya cantidad de partículas se encuentra alrededor de las 50000 unidades.

4.2.9 Evaluación del desempeño de la integración de todos los módulos de interacción

Para analizar la relación entre el número de partículas y el desempeño del sistema de partículas con todos los módulos de interacción expuestos en durante esta investigación, se ejecutó el sistema bajo una cantidad variable de partículas y se obtuvo la duración de todos los procesos del sistema (fases de simulación y despliegue) en un cuadro de animación. A fin de generar un marco comparativo, cada caso de estudio de desempeño bajo una cantidad de

partículas fue ejecutado dentro de la misma escena pero sin la ejecución del módulo encargado de la interacción entre las partículas ya que al analizar los resultados de las pruebas anteriores, es claramente visible que este es el módulo con el menor desempeño de todos los desarrollados. En la Figura 25 se puede observar las características de la escena utilizada durante una de estas pruebas.

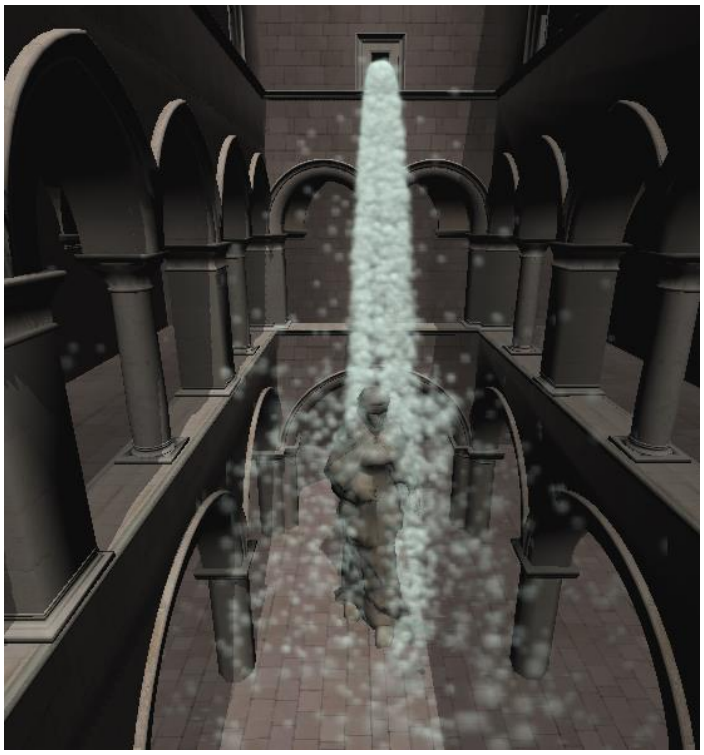
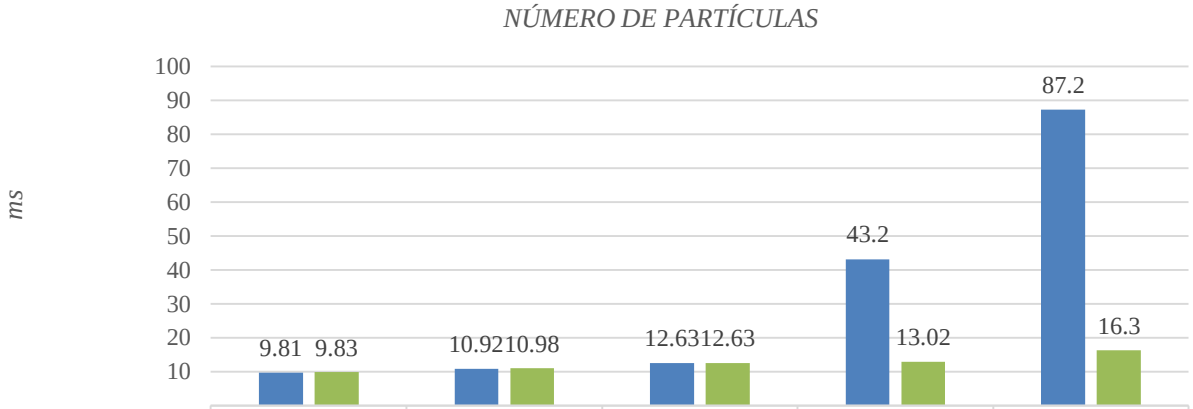


Figura 25. Configuración de la escena *Palacio Sponza* (69000 polígonos) y sistema de alrededor de 50000 partículas con todos los módulos de interacción activos

Es importante aclarar las versiones de cada módulo que fueron utilizadas: para el módulo de recepción de sombras se utilizó su versión sin teselado, en el módulo de colisiones con la escena se utilizó la versión que utiliza mapas de profundidad simples y para el módulo de interacción entre partículas se hizo uso de su versión basada en el uso de rejillas dinámicas y su dinámica de repulsión. Finalmente, la escena utilizada posee 60000 polígonos y una sola luz direccional como fuente de iluminación.



	100	1000	10000	50000	100000
Todos los módulos	9.81	10.92	12.63	43.2	87.2
Sin interacción entre partículas	9.83	10.98	12.63	13.02	16.3

Tabla 9. Rendimiento medido en milisegundos del sistema de partículas con todos los módulos de interacción integrados, y sin el módulo de interacción entre partículas

Tal como muestra la tabla 9, el desempeño del sistema de partículas se redujo a medida que más elementos eran procesados. Esto es coherente con los resultados obtenidos en las pruebas anteriores, ya que como se ha observado, todos los módulos del sistema de partículas ven su desempeño afectado negativamente a medida que se incrementa la cantidad de partículas que estos deben procesar. Al integrarse todos los módulos, los tiempos de procesamiento de cada una de sus operaciones son combinados para conformar el desempeño global del sistema de partículas interactivo.

En la misma tabla se puede observar que en todos los casos estudiados, el desempeño del sistema que no hace uso del módulo de interacción entre partículas es superior al del sistema que si lo hace. Se puede observar como la diferencia de desempeño se incrementa considerablemente a medida que el número de partículas en el sistema crece.

Finalmente, el sistema de partículas que integra todos los módulos de interacción propuestos solo puede ejecutarse de forma fluida en escenarios en los cuales procesa alrededor de 10000 partículas. Por otra parte, el sistema de partículas que abandona la posibilidad de interacción entre sus elementos logra alcanzar una ejecución fluida en todos los casos de prueba desarrollados en el presente estudio.

Capítulo 5: Conclusiones

En esta investigación se estudió e implementó una propuesta de sistema de partículas que es capaz de interactuar con una considerable cantidad de elementos que conforman a una típica escena tridimensional en tiempos de cómputos aceptables para aplicaciones que requieren el despliegue de gráficos en tiempo real. En específico, se diseñaron y desarrollaron módulos que permiten diferentes tipos de interacción de las partículas con las fuentes de luz de la escena, la geometría modelada tradicionalmente y entre las propias partículas procesadas.

El uso de un sistema de partículas basado en la GPU como fundamento de todos los módulos de interacción permitió la simulación de una considerable cantidad de partículas simultáneamente, a la vez que todos los módulos implementados alcanzaron los objetivos de interactividad propuestos como hipótesis iniciales. El módulo de coloración permitió que la dirección de la luz entrante fuera fácilmente perceptible en la apariencia visual de las partículas desplegadas, mientras que los módulos de recepción y proyección de sombras habilitaron estos tipos de interacciones con la geometría de las escenas donde se ejecutó el sistema. El módulo de interacción entre las partículas efectivamente logró que los elementos del sistema interactuasen entre sí bajo las dinámicas de repulsión y atracción establecidas, al tiempo que el módulo de colisiones entre las partículas y las escenas permitió la generación de las trayectorias de las partículas consistentes con este tipo de eventos.

La integración de todos estos módulos en un sistema de partículas logro reducir la barrera visual que naturalmente existe entre los escenarios y los sistemas que no interactúan con su entorno, alcanzando un aspecto consistente con los escenarios en los que se ejecutan tanto en el despliegue de las partículas como en sus dinámicas de movimiento producidas. Es inminente el hecho que la coherencia entre el escenario y las partículas logra exaltar la apariencia de realismo del sistema de partículas ya que, tal como en el mundo real, los observadores de una escena tienen la preconcepción que cada elemento que la compone debe reaccionar a la presencia de los otros elementos adyacentes al mismo.

El sistema de partículas propuesto cumplió con su objetivo primordial de funcionar como un caso de estudio de un sistema modular altamente interactivo partícula-partícula y partícula-escena, ofreciendo una base extensible para diseñar e implementar sistemas de partículas con estado basados en la GPU capaces de interactuar con los elementos de una escena expuestos en este trabajo, incluyendo soporte para coloración y sombreado dinámico por parte de los elementos del sistema de partículas.

5.2 Trabajos futuros

El enfoque modular del presente trabajo de investigación permite la posibilidad de extender la propuesta del sistema de partículas altamente interactivo ya sea mediante la profundización en alguno de los módulos desarrollados, como mediante la adición de nuevos módulos de interacción que permitan comportamientos no considerados en el presente trabajo.

En el primer enfoque, la optimización de los módulos desarrollados se ofrece como la más inmediata alternativa. Especialmente considerando el relativamente bajo desempeño del módulo de interacción entre partículas; cualquier investigación futura dedicada a la optimización del sistema propuesto debe enfocarse primeramente en el estudio e implementación de una metodología que permita mejores tiempos de cómputos que los alcanzables por las presentes propuestas de resolución simple de la problemática de N-Cuerpos y la propuesta basada en el uso de rejillas dinámicas.

En el módulo de coloración de las partículas se sugiere la implementación de una versión que haga uso del esquema de trabajo basado en el proceso teselado para la obtención resultados visuales más precisos en presencia de fuentes de luz con factores de atenuación que puedan ser interpolados dentro del procesador de dominio. Igualmente, es posible extender las funcionalidades de este módulo mediante el uso de mapas de normales dentro de las partículas para los casos en los que se requiera una apariencia visual más detallada que la lograda mediante la simple aproximación de curvatura.

En los módulos de recepción y proyección de sombras, la optimización de los cálculos y reducción de artefactos visuales son dos puntos posibles a extender. En el primer caso se recomienda el desarrollo de ajustes que permitan la creación y uso más eficiente de los mapas de sombras utilizados, mientras que para la reducción de artefactos es posible estudiar e implementar las alternativas propuestas por Microsoft [22], donde se destaca el uso de mapas de cascadas con sombras diferidas y el suavizado de los mapas de sombras mediantes cálculos que consideren la posición y perspectiva de la cámara.

Finalmente, el módulo de interacción entre las partículas y el escenario donde se encuentran puede ser mejorado mediante la creación de mapas de profundidad de menor resolución y la generación de mapas cúbicos de profundidad mediante una metodología más eficiente. Para esto último se recomienda la propuesta de Jason Zink [23], donde se utiliza el procesador de geometría para generar el mapa cúbico en un único pase de despliegue. Otra rama de estudio dentro de este módulo es el diseño e implementación de algoritmos que permitan el cálculo automatizado del umbral de colisión óptimo a utilizar por el programa encargado del cálculo de colisiones en espacio de pantalla.

El término de interactividad es uno considerablemente amplio, por lo que futuros trabajos tienen el potencial de añadir una gran cantidad de diferentes tipos de interacción a la base expuesta en este documento. Un ejemplo relevante es lograr la reacción de la geometría modelada tradicionalmente frente a las partículas del sistema, esta interacción no solo puede ser del tipo de colisión estudiado en este trabajo de grado durante el desarrollo del módulo de interacción entre las partículas y la escena, sino que también puede ser basada en efectos más singulares (ver [24] para más detalle).

Otros posibles tipos de interacción sugeridos para su desarrollo sobre la base expuesta en este documento son la interacción entre las partículas y cuerpos volumétricos, la interacción entre las partículas utilizando dinámicas de fluidos y la emisión de luz de las partículas sobre la geometría modelada tradicionalmente en un entorno de despliegue basado en la metodología de despliegue diferido.

Referencias

[1] William T. Reeves, " Particle Systems A Technique for Modeling a Class of Fuzzy Objects " in *Computer Graphics Volume 17*, Lucasfilm Ltd, 1983, pp. 359-375.

[2] Steve Rotenberg, "Particle Systems " , Febrero 2004. [En línea]. Disponible en: http://graphics.ucsd.edu/courses/cse169_w04/CSE169_15.ppt

[3] Matt Swoboda, "A thoroughly modern particle system" , Octubre 2009. [En línea]. Disponible en: http://directtovideo.wordpress.com/2009/10/06/a-thoroughly-modern-particle-system/

[4] Lutz Latta , "Building a Million-Particle System", 2004. [En línea]. Disponible en: http://www.gamasutra.com/view/feature/130535/building_a_millionparticle_system.php?print=1

[5] Veronica Orvalho , "Particle Systems", notas del curso de Computación Gráfica, Departamento de Ciencias de la Computación, Universidad de Porto, 2009. [En línea]. Disponible en: http://bit.ly/1CL5BOy

[6] Muhammad Karim, "Blended Particles", 2009. [En línea]. Disponible en: http://www.cs.unm.edu/~mskarim/cs513_assign43.html

[7] Luke Durant, Tamas Szalay & Russell McClellan ,"GPU Programming", notas del curso CS 179, Departamento de las Ciencias Matematicas y de Cómputo, Instituto de Tecnología de California, 2013. [En línea]. Disponible en: http://courses.cms.caltech.edu/cs101gpu/lec6_particles.pdf

[8] Mike Houston, "Advanced Programming (GPGPU)", 2007. [En línea]. Disponible en: http://graphics.stanford.edu/~mhouston/public_talks/cs448-gpgpu.pdf

[9] Behzad akbari, "Particle Systems", notas del curso CSE 4431, Departamento de Ciencias de Computación e Ingeniería, Universidad de York, Invierno 2011. [En línea]. Disponible en: http://bit.ly/1z98o1e

[10] Tobias Persson, "Practical Particle Lighting". En Game Developers Conference, San Francisco, CA, 2012. [En línea]. Disponible en: http://www.bitsquid.se/presentations/practical-particle-lighting.pdf

[11] Gary McTaggart, "Half-Life 2 / Valve Source Shading". En Game Developers Conference, San Jose, CA, Marzo 2004. [En línea]. Disponible en: http://www2.ati.com/developer/gdc/d3dtutorial10_half-life2_shading.pdf

[12] Nickolay Kasyan, Nicolas Schulz & Tiago Sousa, "Secrets of CryENGINE 3 Graphics Technology". En SIGGRAPH, Vancouver, Columbia Británica, 2011. [En línea]. Disponible en: http://bit.ly/1uzim7W

[13] Edd Biddulph , "Screenspace Particle Physics" , 2013. [En línea]. Disponible en: https://sites.google.com/site/eddbiddulph/cg/screenspace-particle-physics

[14] Nvidia, "Physx/APEX", 2010. [En línea]. Disponible en: https://developer.nvidia.com/apex

[15] Nvidia, "DirectCompute para NVIDIA", 2013. [En línea]. Disponible en: http://www.nvidia.es/object/directcompute_es.html

[16] Neil Trevett, "OpenCL Introduction", 2013. [En línea]. Disponible en: https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf

[17] Microsoft, "Diagramas de clases de UML: Referencia", 2015. [En línea]. Disponible en: http://msdn.microsoft.com/es-es/library/dd409437.aspx

[18] Microsoft, "Diagramas de secuencia de UML: Referencia", 2015. [En línea]. Disponible en: http://msdn.microsoft.com/es-es/library/dd409377.aspx

[19] Stephen Schieberl, "Wrap Your Mind Around Your GPU", 2010. [En línea]. Disponible en: http://www.bantherewind.com/wrap-your-mind-around-your-gpu

[20] Microsoft, “DirectCompute Basic Win32 Samples”, 2014. [En línea]. Disponibles en: https://code.msdn.microsoft.com/windowsdesktop/DirectCompute-Basic-Win32-7d5a7408

[21] Carl Källman, “On the n-body problem, chaos and computability”, Marzo 2012. [En línea]. Disponible en: http://www.lakeudenursa.fi/tiedostot/planetary.pdf

[22] Microsoft, “Common Techniques to Improve Shadow Depth Maps”, 2013. [En línea]. Disponible en: https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx

[23] Jason Zink, “Single Pass Environment Mapping”,2011. [En línea]. Disponible en: http://content.gpwiki.org/index.php/D3DBook:Single_Pass_Environment_Mapping

[24] Shannon Drone, “Real-Time Particle Systems on the GPU in Dynamic Environments”, 2007. [En línea]. Disponible en: http://bit.ly/1H9Lb62

[25] Morgan McGuire, “Meshes”,2013. [En línea]. Disponible en: http://graphics.cs.williams.edu/data/meshes.xml