**UNIVERSIDAD DE SANTIAGO DE CHILE**

**FACULTAD DE INGENIERÍA**

**DEPARTAMENTO DE INGENIERÍA INFORMÁTICA**


# ESTRATEGIA DE PARALELIZACIÓN EN GPU DE SIMULACIONES MONTE CARLO PARA EL PASO DE PARTÍCULAS A TRAVÉS DE LA MATERIA

**DANILO ABURTO VIVIANS**

PROFESOR GUÍA: DR. FERNANDO RANNOU

TRABAJO DE TESIS PRESENTADO
EN CONFORMIDAD A LOS REQUISITOS PARA
OBTENER EL GRADO DE MAGISTER EN
INGENIERIA INFORMATICA

SANTIAGO – CHILE
2014

# AGRADECIMIENTOS

Agradezco a Dios por darme las oportunidades para avanzar y crecer. Gracias también a mi familia, mi novia y seres queridos por el apoyo que siempre me han entregado.

Extiendo los agradecimientos a mis profesores. Gracias a mi profesor guía Fernando Rannou por su tiempo y voluntad para orientarme durante el desarrollo de esta tesis. Gracias a Pamela Aguirre por soportar mis quejas durante 6 años. Agradezco también al profesor Edgardo Sepúlveda por enseñarnos a perseguir la excelencia en todo lo que uno hace.

Gracias a quienes fueron mis compañeros de universidad, por los gratos momentos, por la ayuda y también sus consejos.

Mis más sinceros agradecimientos a los socios de nuestra empresa Requies: Camilo Farfán, Diego Jaume y en particular a Mario López, por prestarme sus recursos computacionales.

*A mi familia y seres queridos*

# RESUMEN

La bioluminiscencia es una técnica utilizada en imagenología molecular. Por medio de esta técnica se logra la producción y emisión de luz en organismos vivos con el objetivo de analizar su actividad molecular. A lo largo de los años se han destinado importantes esfuerzos y recursos para realizar simulaciones de estos procesos biológicos, las cuales son muy costosas computacionalmente.

En este trabajo se aborda una estrategia de paralelización de simulaciones Monte Carlo para el paso de partículas a través de la materia, aplicada en particular al proceso de bioluminiscencia. Dicha estrategia comprende la utilización de los dispositivos GPU, para los cuales se ha generado un modelo orientado a objetos que permite aprovechar el tipo de procesamiento que éstos dispositivos ofrecen.

Los resultados obtenidos muestran un *speedup* de 33x para una GPU de bajo costo, lo cual demuestra el potencial de estas tarjetas gráficas para resolver problemas de simulación eficientemente. Sin embargo aún se deben realizar esfuerzos para obtener soluciones de mayor escalabilidad y que además permitan extender a otras áreas los resultados obtenidos.

**Palabras Claves**: GPU, Simulación Monte Carlo, Simulación de eventos discretos

# ABSTRACT

Bioluminescence is a technique used in Molecular Imaging. Through this technique the production and emission of light by living organisms is achieved with the purpose of analyzing the molecular activity. Over the years researchers have invested significant efforts and resources to perform simulations of these biological processes, which are computationally expensive.

This work discusses a parallel strategy of Monte Carlo simulations for the passage of particles through matter focused on bioluminescence process. This strategy includes the use of GPU devices, for which an object-oriented model has been generated. This strategy offers an advantage in the type of processing of these devices.

The results show a 33x speedup for a low cost GPU, demonstrating the potential of GPU devices for the efficient solving of simulation problems. Nonetheless there is still a need to improve the solution's scalability and also extend the obtained results to other areas.

**Keywords**: GPU, Monte Carlo simulation, Discrete event simulation

# ÍNDICE DE CONTENIDOS

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# ÍNDICE DE ALGORITMOS

# CAPÍTULO 1. INTRODUCCIÓN

## 1.1 ANTECEDENTES Y MOTIVACIÓN

Existen diversas técnicas utilizadas en la medicina moderna para realizar estudios no invasivos de procesos biológicos. Muchas de éstas pueden ser aplicadas tanto en humanos como en animales. La bioluminiscencia, en particular, es una técnica utilizada en imagenología molecular en el tratamiento y seguimiento, por ejemplo, de terapias de cáncer (Rehemtulla et al., 2000) y tumores cerebrales (Rehemtulla et al., 2002). El proceso consiste en la implantación de células que emiten luciferasa, en uno o más tejidos según sea el caso. La luz emitida por estas células es recolectada por dispositivos tecnológicos a partir de los cuales se estima la posición y tamaño del tumor. Por ejemplo, la figura 1.1 muestra una típica imagen de bioluminiscencia en ratones, en la cual se puede observar la presencia de un tumor en el hombro izquierdo del ratón. El dispositivo sólo capta la sección en color; el resto de la imagen es capturada con otros medios tecnológicos, en este caso rayos X. Además la tecnología actual permite capturar sólo una proyección, es decir, no es tomográfica por cuanto no permite reconstruir imágenes en 3 dimensiones.

A través de los años las simulaciones computacionales para estos procesos biológicos han sido un importante objeto de estudio, intentando pronosticar el comportamiento que los fotones tendrán en diferentes tejidos sin la necesidad de utilizar seres vivos (Rogers, 2006). Estas simulaciones podrían ser utilizadas en el diseño de nuevos sistemas de detección de fotones de luz; simulaciones de monitoreo de terapias basadas en bioluminiscencia y el diseño de algoritmos tomográficos especializados en bioluminiscencia (Alexandrakis et al., 2005, 2006).

No obstante, el tiempo requerido por una simulación puede ser de varias horas, por lo cual se continúa investigando cómo la computación de alto rendimiento puede cambiar esta situación, de modo que los tiempos se vean reducidos y de tal forma producir un avance más rápido en las investigaciones científicas.

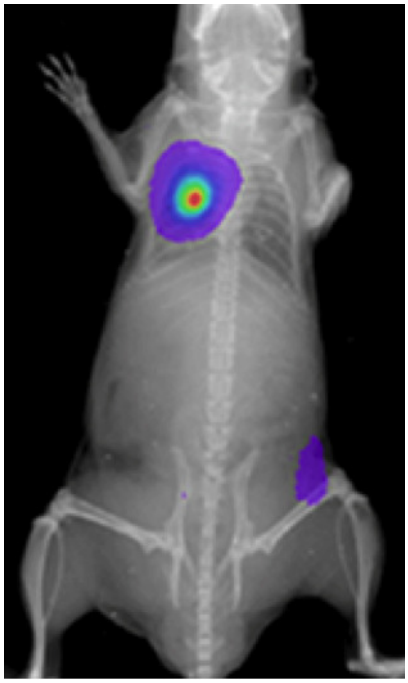


FIGURA 1.1: *Ejemplo de visualización de tumor en bioluminiscencia.*

## 1.2 DESCRIPCIÓN DEL PROBLEMA

La simulación de transporte de fotones es un proceso altamente costoso computacionalmente. Las técnicas de paralelización que actualmente existen ayudan a mejorar sustancialmente el rendimiento de los sistemas, sin embargo, los procesos de simulación pueden tomar varias horas (o incluso días) dependiendo de su complejidad (Maigne et al., 2004). De acuerdo con este contexto la definición del problema es: "¿Cómo mejorar el rendimiento de simulaciones Monte Carlo, tales como la simulación de transporte de fotones a través de la materia?". En particular, uno de los desafíos del proyecto consiste en diseñar una solución eficiente en términos de tiempo para tarjetas de procesamiento gráfico.

### 1.2.1 Modelo de computación de GPU

El modelo de computación de GPU (*Graphics Processing Unit*) es denominado *Single Instruction Multiple Thread* (SIMT) (Xu et al., 2013). El modelo consiste en que muchas hebras de ejecución ejecutan la misma instrucción al unísono. Cuando una hebra diverge de las otras, es

decir intenta ejecutar una instrucción distinta a las demás, la ejecución de éstas se secuencializa: sin importar el número de procesadores con que se cuente, sólo uno de ellos estará siendo utilizado por la hebra divergente, ya sea antes o después de la ejecución de las otras, lo cual desperdicia la capacidad de cómputo masiva de la GPU al ejecutar la hebra divergente y el resto de las hebras de manera secuencial. Esto no ocurre en un sistema multi-núcleo donde las hebras pueden seguir una línea de ejecución independiente de las otras. Así, dependiendo de la aplicación o problema a resolver, la GPU puede o no ser la tecnología adecuada para abordarlo.

Al menos dos condiciones permiten que un problema sea adecuado para ser abordado mediante GPU:

1. El problema requiere grandes cantidades de cómputo, lo cual permite aprovechar la capacidad de cómputo masiva y los overheads asociados serán despreciables.

2. El problema se ajusta al modelo SIMT descrito anteriormente.

Una simulación de partículas puede involucrar cientos de millones de fotones. Por lo tanto, el fenómeno de bioluminiscencia satisface claramente la primera condición. Sin embargo, no es claro que dicho tipo de simulación pueda ser abordado por medio de una estrategia basada en el modelo SIMT. Como se muestra en la figura 1.2, siete fotones de luz pueden seguir caminos independientes y a su vez ser afectados por diferentes procesos físicos, por ejemplo, algunos fotones pueden estar refractándose y otros reflectándose. En consecuencia no es evidente que los procesos físicos que afectan a los fotones puedan ser simulados por medio del modelo de cómputo SIMT.

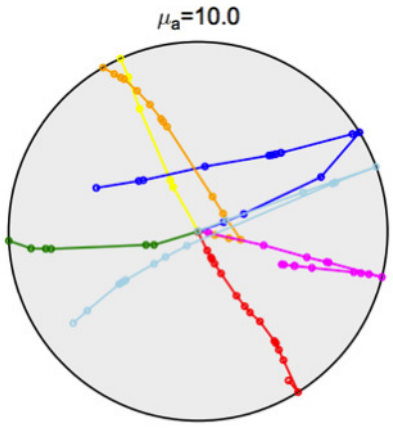



FIGURA 1.2: *Ejemplos de los caminos que pueden tomar los fotones de luz.*

La segunda condición se ajusta fácilmente a un modelo tradicional multi-núcleo, en el cual cada fotón de luz podría ser simulado por una hebra independiente. Pero como se mencionó anteriormente el modelo de cómputo de GPU requiere que todas las hebras ejecuten la misma instrucción al mismo tiempo para obtener el máximo provecho de los multiprocesadores. Debido a esto, el problema no se ajusta directamente al modelo de cómputo mencionado.

Por lo tanto, a través de este trabajo se intenta encontrar una manera de organizar las simulaciones Monte Carlo de eventos discretos e independientes de una forma que satisfaga el modelo SIMT y de este modo explotar al máximo las capacidades de cómputo de la GPU.

## 1.3 SOLUCIÓN PROPUESTA

La solución propuesta en este trabajo implica el diseño e implementación de una estrategia de paralelización de simulaciones Monte Carlo. En particular se implementará para bioluminiscencia, el cual es un fenómeno en que los fotones son independientes entre sí.

A su vez, la estrategia generada como solución tendrá como foco principal el modelo de cómputo de los dispositivos GPU. Para esto el proceso de bioluminiscencia será implementado en CUDA.

### 1.3.1 Características de la solución

Se espera que el modelo de la solución a obtener sea extensible, de modo que pueda ser utilizado posteriormente para implementaciones de características no consideradas en el presente proyecto, como es el caso de experimentos diferentes a bioluminiscencia.

Gracias al uso de CUDA en sus versiones recientes, el diseño deberá ser orientado al objeto con respecto a la programación que controla el flujo de la aplicación. De acuerdo con esto, el modelo de objetos propuesto en (Lagos, 2012) podría ser modificado y extendido para incluir clases que incorporen el uso de GPU.

La solución que se espera encontrar debería tener características similares a los modelos de computación vectorial, es decir, la simulación deberá ser organizada de modo que la GPU realice

los mismos cálculos para una gran cantidad de fotones.

### 1.3.2 Propósito de la solución

El propósito de la solución es reducir los tiempos involucrados en las simulaciones Monte Carlo de eventos discretos e independientes. De esta forma se intentará sacar un mejor provecho de los recursos computacionales que se poseen. Todo esto con el objetivo principal de acelerar los experimentos en investigaciones científicas que involucran simulaciones de este tipo.

Se espera que la solución, en su diseño, sea extensible a otras aplicaciones de eventos discretos e indepentientes, como por ejemplo PET (Positron emission tomography), SPECT (Single photon emission tomography) y rayos X. Con respecto a la implementación, se espera que sea llevada a cabo para simulaciones Monte Carlo.

Gracias a los avances de CUDA en sus versiones recientes, el diseño podrá ser orientado al objeto con respecto a la programación que controla el flujo de la aplicación. De acuerdo con esto, el modelo de objetos propuesto en (Lagos, 2012) podrá ser modificado y extendido para incluir clases que incorporen el uso de GPU.

### 1.3.3 Alcances y limitaciones de la solución

La principal limitación de la solución propuesta es que su implementación y evaluación serán llevadas a cabo sólo para bioluminiscencia. Se ha elegido bioluminiscencia como especialización del *toolkit* pues es un fenómeno relativamente más simple que otros procesos de radiación, como por ejemplo, el transporte de rayos gama a través de la materia. Sin embargo, ya que los aspectos fundamentales de la simulación son similares a muchos fenómenos basados en eventos, la arquitectura del software será diseñada pensando en el proceso genérico de transportes de partículas (discretas) a través de un medio arbitrario. Además, ya que se dispone de programas de bioluminiscencia públicamente disponibles y validados, es posible realizar comparaciones de los resultados obtenidos con los resultados de dichos programas. Específicamente, se comparará con el software desarrollado e implementado por (Wang & Jacques, 1992). Dichas comparaciones considerarán las aceleraciones

de procesamiento a través de GPU con respecto a CPU y también cómo se comporta la aceleración variando los parámetros involucrados en la simulación.

Además, no se considera un esquema de computación distribuida de GPU, por lo cual la solución abordará el problema bajo un único dispositivo distribuyendo la carga de trabajo en sus unidades de procesamiento.

Se realizarán pruebas unitarias para cada uno de los procesos físicos involucrados sin considerar en ellas rendimiento computacional. Estas pruebas están orientadas a determinar si el desplazamiento de los fotones es correcto en diferentes tipos de tejidos con distintas propiedades de absorción, difusión, refracción y reflexión.

## 1.4 OBJETIVOS Y ALCANCE DEL PROYECTO

### 1.4.1 Objetivo general

A través del proyecto se espera encontrar una solución que permita mejorar el rendimiento de las simulaciones de transporte de fotones a través de la materia, explotando las capacidades de cómputo provistas por las tecnologías de procesamiento gráfico. De acuerdo con esto, el objetivo del proyecto es diseñar e implementar una estrategia de paralelización de eventos discretos e independientes para el modelo de computación de GPU.

### 1.4.2 Objetivos específicos

Los objetivos específicos relacionados al proyecto son:

1. Encontrar oportunidades de paralelización con el fin de explotar la capacidad de cómputo de un dispositivo GPU.
2. Diseñar o adaptar un modelo de simulación de partículas para GPGPU (General-Purpose Computing on Graphics Processing Units).
3. Implementar el diseño anterior en GPU para bioluminiscencia.

4. Diseñar experimentos de bioluminiscencia para validación y evaluación de rendimiento.

5. Proponer y aplicar un método para comparar y/o evaluar los resultados de las simulaciones.

## 1.5 METODOLOGÍA Y HERRAMIENTAS UTILIZADAS

### 1.5.1 Metodología

El trabajo a realizar posee una etapa inicial de investigación, en la cual se debe por medio de la literatura, la observación y la experimentación, buscar oportunidades de paralelización adecuadas. Bajo dicha perspectiva el método científico será particularmente útil para guiar la totalidad del trabajo. Además, una vez que se haya obtenido el diseño éste debe ser implementado para un tipo de simulación en particular, por lo cual se requerirá la construcción de software. Por lo tanto, se utilizará una metodología que consta de cuatro fases, las cuales se indican a continuación:

1. **Concepción**: durante esta etapa se estudian todas las variables y aspectos fundamentales que deben ser consideradas para el diseño y construcción del proyecto.

2. **Elaboración**: esta etapa consiste en el diseño de los principales componentes involucrados. En esta etapa se definen modelos, esquemas y diagramas que se utilizarán para construir.

3. **Construcción**: durante la construcción se implementan todos los elementos definidos en la fase de elaboración.

4. **Transición**: esta fase recoge resultados obtenidos a partir de la construcción para refinar y estabilizar el producto.

### 1.5.2 Herramientas de desarrollo

A continuación se presentan las herramientas que se utilizarán para el desarrollo del proyecto. Se listan herramientas de software y de hardware.

#### *1.5.2.1 Herramientas de software*

Para la escritura del documento se utilizarán las siguientes herramientas:

1. Sistema operativo OS X Mavericks 10.9.
2. Suite Ofimática Microsoft Office:mac
3. TeXstudio.

Para el desarrollo, compilación y ejecución de las implementaciones se utilizará:

1. Sistema operativo Linux Debian.
2. CUDA 5.5.
3. uC++.

#### *1.5.2.2 Herramientas de hardware*

Las pruebas oficiales y el desarrollo serán realizados en un computador del Departamento de Ingeniería Informática con las siguientes características:

1. 4Gb de RAM
2. Procesador AMD Athlon II 630 de 4 núcleos
3. Tarjeta de Video GeForce GTX 470

Las pruebas oficiales que se hagan con software desarrollado en uC++ serán ejecutadas en el clúster del Departamento de Ingeniería Informática, sus características son:

1. 1 nodo AMD Opteron 6168
2. 2 procesadores de 12 cores de 1.8 GHz
3. 512 Mb de cache

4. 32 Gb de memoria RAM

Además serán utilizados los recursos computacionales que posee el estudiante, los cuales son:

1. Un computador de escritorio con:
   *a*) 4 Gb de RAM.
   *b*) Procesador Phenom 9850 2.5Ghz.
   *c*) Tarjeta de Video Nvidia 9800 GT.
2. Computador portátil Macbook Pro Mid 2012 con:
   *a*) 16GB de RAM DDR3.
   *b*) Procesador Intel i5 2.5 Ghz.
   *c*) Tarjeta de video integrada Intel HD Graphics 4000.

## 1.6 ORGANIZACIÓN DEL DOCUMENTO

El presente documento está dividido en 6 capítulos. Durante el primer capítulo se introduce al lector en los conceptos más generales acerca de la problemática abordada en este trabajo. Se indican aspectos de la solución, del proyecto y también de la metodología utilizada.

El segundo capítulo tiene como foco presentar los conceptos involucrados en la simulación de partículas a través del método Monte Carlo. Además son analizadas las estrategias de paralelización utilizadas sobre sistemas de simulación de partículas como Geant4 y GATE. Finalmente, se realiza un breve análisis de estrategias de simulación sobre GPU.

El tercer capítulo aborda el diseño de la estrategia propuesta en este trabajo. Comienza con una explicación acerca de las reglas involucradas en el transporte de fotones, en particular para el proceso de bioluminiscencia. A partir de esto se repasa el modelo de objetos concurrentes utilizado como base de la estrategia. Finalmente, se explica la arquitectura de la simulación, la cual recoge un modelo de objetos y un modelo de paralelización a través de procesos.

Durante el cuarto capítulo se aborda la implementación del sistema de simulación. Se describen los principales pasos para la extensión a tres dimensiones de un modelo de objetos reutilizado. Posteriormente, se indica la manera en que las reglas de simulación han sido mapeadas a procesos ejecutados en paralelo en una GPU. Por último, se indican cómo todo lo anterior es unido por medio de un modelo de control para la simulación.

En el quinto capítulo se presentan los resultados obtenidos en diferentes experimentos. Las pruebas realizadas comprenden la verificación del cumplimiento de las reglas de simulación y también análisis de rendimiento computacional.

Finalmente, en el sexto y último capítulo se exponen las conclusiones obtenidas a partir de la realización del presente trabajo, las cuales están enfocadas en el beneficio que otorga una arquitectura que incorpora orientación a objetos.

# CAPÍTULO 2. ESTADO DEL ARTE

A través del presente capítulo se describe brevemente los conceptos más importante del método Monte Carlo, y de algunos software Monte Carlo para física de partículas. Finalmente, se repasa las estrategias de paralelización de dichos software, el estado actual de la programación GPU y cómo esta nueva tecnología se ha utilizado en simulaciones similares.

## 2.1 EL MÉTODO MONTE CARLO

El método Monte Carlo abarca una clase de algoritmos computacionales, basados en el muestreo aleatorio de distribuciones de probabilidad, para obtener soluciones aproximadas a problemas complejos y cuantitativos (Binder & Heermann, 2002). El método moderno Monte Carlo se le atribuye a Stanslaw Ulam and Von Neumman quienes fueron los primeros en construir un programa computacional para estimar propiedades de experimentos nucleares cuando trabajan en el Proyecto Manhatan, en el Laboratorio Nacional Los Alamos, Estados Unidos. Desde entonces, dicho método ha sido usado en diversos campos como la química (Doll & Freeman, 1994), biología (Schellenberger, 2010) y finanzas (Glasserman, 2004).

El término Monte Carlo (MC) a veces se usa para referirise a las técnicas MC estadísticas utilizadas para aproximar distribuciones de probabilidad de algún estimador; otras veces para indicar un método aproximado de integración numérica, y finalmente, se emplea para designar simulaciones MC. En cualquier caso, todas estas formas del método involucra el uso de números pseudo-aleatorios para obtener muestras de distribuciones de probabilidad que gobiernan las variables, y con el objetivo de estimar una o más propiedades numéricas. Kalos y Whitlock (Kalos & Whitlock, 2009) emplean la siguiente definición:

*"A definition of a Monte Carlo method would be one that involves deliberate use of random*

*numbers in a calculation that has the structure of a stochastic process. By stochastic process, we mean a sequence of states whose evolution is determined by random event".*

Dichos cálculos, según James (James, 1980), siempre se pueden asociar a algún cáculo de integrales:

''*At least in a formal sense, all Monte Carlo calculations are equivalent to integrations. This follows from the definition of a Monte Carlo calculation as producing a result F which is a function of random numbers".*

Según lo anterior el método Monte Carlo puede aplicarse tanto a procesos estocásticos y determinísticos. Cuando el problema a resolver posee una naturaleza estocástica, como por ejemplo los procesos de difusión y absorción de neutrones en una barra de plomo, una *simulación* Monte Carlo constituye una simulación directa del proceso mismo, pues los números aleatorios usados están directamente relacionados con el transporte de neutrones a través de la materia. Sin embargo, el mismo proceso puede ser formulado con el método clásico en términos de integrales multidimensionales. En este caso las variables muestreadas están asociadas al dominio de la integral y no necesariamente tienen una relación directa con el proceso físico. Este tipo de simulaciones son también simulaciones MC, pero de tipo indirecto o *crudo*.

El problema de la aguja de Buffon es un ejemplo simple que ilustra cómo un mismo problema puede verse como la estimación de la esperanza de una variable aleatoria, como una simulación directa o como una simulación indirecta. A continuación se describe en detalle este problema pues su mecanismo MC es similar al usado en la simulación de bioluminiscencia.

### 2.1.1 Formulación del problema de la aguja de Buffon

Suponga que una aguja de largo $L$ y grosor cero (infinitesimal) es lanzada sobre un plano infinito que contiene infinitas líneas paralelas separadas a una distancia $d > L$, como lo muestra la figura 2.1. El problema consiste en estimar la probabilidad $p$ que la aguja, al caer sobre el plano, intersecte una de las líneas.

Con respecto a la figura 2.2, se define la posición de la aguja, $x$, como la distancia desde el centro de la misma a la línea más cercana. Asi mismo, se define la orientación $\theta$, como el ángulo

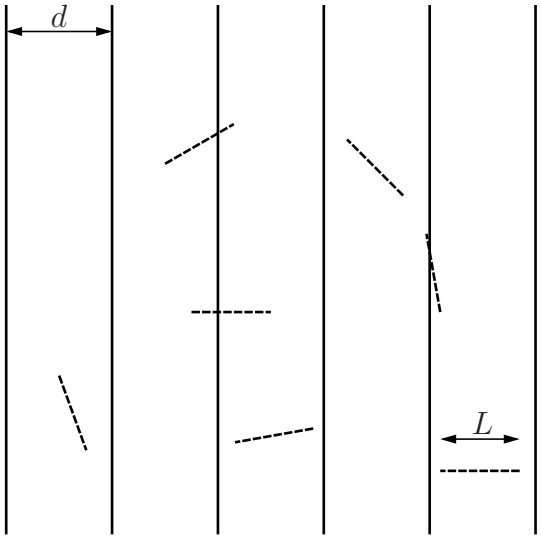



FIGURA 2.1: Diversas formas en que una aguja de largo $L$ puede caer sobre un plano infinito de líneas paralelas separadas a una distancia $d$. En este caso el largo de la aguja es menor que la separación entre líneas.

de inclinación respecto del eje $Y$. La aguja intersecta una línea cuando la distancia $x$ es menor o igual al largo de la proyección de la mitad de la aguja en la dirección del eje $X$, es decir cuando

$$x \leq \frac{L}{2} \sin\theta \tag{2.1}$$

Note que el análisis siempre es respecto de la línea más cercana, y por lo tanto el problema es simétrico respecto al eje $X$. Luego, $x$ se distribuye uniformemente entre 0 y $d/2$, y $\theta$ se distribuye uniformemente entre 0 y $\pi$, es decir

$$\begin{aligned} x &\sim U[0, d/2] \\ \theta &\sim U[0, \pi] \end{aligned}$$

donde $U[a, b]$ representa la distribución uniforme entre $a$ y $b$, incluídos ambos límites.

Antes de dar una solución al problema de la aguja de Buffon, a continuación se revisa los conceptos básicos del método de integración MC, pues constituye el mecanismo común de todos los métodos de solución.

### 2.1.2 Integración Monte Carlo

Aunque el método de integración MC es aplicable a funciones multidimensionales, se usa funciones unidimensionales para simplificar la exposición de sus fundamentos. Considere la siguiente

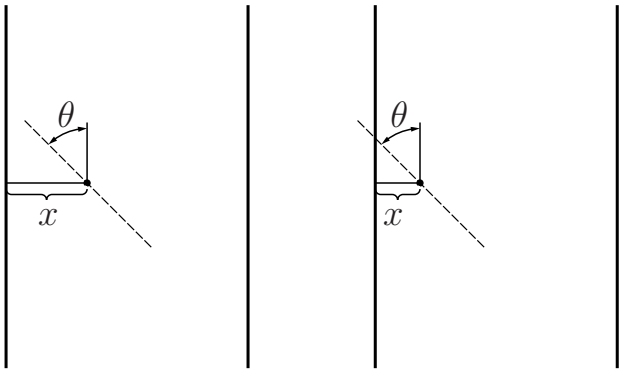



FIGURA 2.2: La aguja intersecta una de las líneas cuando la distancia desde su centro a la línea más cercana es menor que la proyección de la mitad del largo de la aguja. (a) no hay intersección, y (b) si hay intersección.

integral definida de la forma

$$F = \int_a^b f(x)\, dx \tag{2.2}$$

Si $p(x)$ es cualquier función de distribución definida en $[a, b]$, es decir que satisfaga

$$p(x) > 0, \quad \forall x \in [a, b], \qquad \int_a^b p(x) = 1, \tag{2.3}$$

entonces la integral 2.2 puede reescribirse como:

$$F = \int_a^b \left(\frac{f(x)}{p(x)}\right) p(x)\, dx \tag{2.4}$$

Esto implica que $F$ puede ser visto como la esperanza de la función $f(x)/p(x)$ respecto de $p(x)$. Matemáticamente,

$$F = E_p\left\{\frac{f(x)}{p(x)}\right\} \tag{2.5}$$

Entonces, un problema no estocástico de cálculo de una integral ha sido transformado en uno estocástico, y por lo tanto es posible usar la la *Ley débil de los grandes números*, para estimar su valor.

**Teorema 2.1 (Ley débil de los grandes números)**. Sea $X_1, X_2, \ldots, X_N$, una secuencia de variables aleatorias independientes e idénticamente distribuidas y con valor esperado $\mu = E[X] < \infty$. Entonces,

$$\frac{1}{N}(X_1 + X_2 + \cdots + X_N) \xrightarrow{p} E[X] \tag{2.6}$$

La suma del lado izquierdo de la ecuación 2.6 es el promedio de la muestra. Luego, esta ley establece que el promedio de la muestra tiende, en probabilidad, al promedio real $\mu$, a medida

que $N$ tiende a infinito. Por lo tanto, usando la Ley débil de los grandes números, la ecuación 2.5 puede aproximarse mediante el promedio de la muestra, denominada el *estimador Monte Carlo*:

$$\hat{F}_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \tag{2.7}$$

donde $x_i \sim p(x)$.

La elección de la distribución $p(x)$ es importante pues de ella depende cuán eficiente y fácil resulta el cálculo del estimador MC. Generalmente se elige la distribución uniforme en $[a, b]$, es decir $p(x) = 1/(b-a)$. Reemplazando esta función en 2.7, se obtiene:

$$\hat{F}_N = \frac{(b-a)}{N} \sum_{i=1}^{N} f(x_i) \tag{2.8}$$

con $x_i \sim U[a, b]$.

El estimador MC puede aplicarse a cualquier integral definida. Por ejemplo, considere la siguiente integral

$$F = \int_a^b f(x)g(x)\, dx \tag{2.9}$$

Si $g(x) > 0$ para todo $x \in [a, b]$, pero no es una distribución de probabilidad, basta definir una nueva función $h(x)$ de la siguiente forma.

$$h(x) = \frac{g(x)}{C}$$

donde

$$C = \int_a^b g(x)\, dx$$

Entonces, la integral 2.9 se expresa como

$$F = C \int_a^b \left( \frac{f(x)}{g(x)} \right) h(x)\, dx$$

cuyo estimador MC es

$$\hat{F}_N = \frac{C}{N} \sum_{i=0}^{N} \frac{f(x_i)}{g(x_i)}, \quad x_i \sim h(x) \tag{2.10}$$

### 2.1.3 Soluciones MC al problema de la aguja de Buffon

El problema de la aguja de Buffon puede ser resuelto en al menos dos formas distintas, dependiendo de cómo el problema es modelado. Las dos soluciones que a continuación se describen emplean el uso de integración MC.

**Simulación MC directa:** La aleatoriedad del problema está dada por la aleatoriedad de $x$ y $\theta$. Luego, una simulación directa MC para resolver este problema consiste en obtener muestras aleatorias de dichas variables para "simular" el lanzamiento de la aguja. Este procedimiento requiere de un generador de números pseudo-aleatorios distribuidos uniformemente en $[0, d/2]$ y otro generador de números distribuidos uniformemente en $[0, \pi]$. Sin embargo, una implementación computacional generalmente sólo dispone de un generador de números pseudo-aleatorios distribuidos uniformemente ente 0 y 1. En este caso, la transformación es simplemente un escalamiento lineal del rango $[0, 1]$ al rango de la variable en cuestión. Si $u_1 \sim U[0, 1]$ y $u_2 \sim U[0, 1]$, entonces tenemos que

$$\begin{aligned} x &= \frac{d}{2} \cdot u_1 \\ \theta &= \pi \cdot u_2 \end{aligned}$$

El Algoritmo 2.1 detalla los pasos para el cálculo de la probabilidad que la aguja intersecte una de las líneas. La función `Uniform()` es un generador de números-pseudo aleatorios distribuidos uniformemente en $[0, 1]$. Luego, las línea 3, 4, 5, y 6, constituyen el muestreo de las distribuciones de probabilidad del problema. En la línea 7 se consulta si existe intersección, y si es así, se incrementa el contador $n$. Finalmente, la estimación de la probabilidad de intersección se calcula y retorna en la línea 11.

**Algoritmo 2.1:** Aguja de Buffon mediante simulación directa MC.

```
1  n ← 0;
2  for i ← 1 to N do
3  |   u_1 ← Uniform();
4  |   u_2 ← Uniform();
5  |   x ← u_1 d/2;
6  |   θ ← u_2 π ;
7  |   if x ≤ L/2 sin θ then
8  |   |   n ← n + 1 ;
9  |   end
10 end
11 p = n/N
```

**Simulación MC indirecta o cruda:** Nótese que independientemente de la posición donde caiga la aguja, la probabilidad de intersección para un ángulo dado $\theta$ es la razón:

$$p(\theta) = \frac{L \sin(\theta)}{d}$$

Entonces, para calcular la probabilidad de intersección, se calcula el valor esperado de $p(\theta)$ , es decir:

$$E\{p(\theta)\} = \int_0^{\pi} p(\theta)\rho(\theta)\, d\theta \tag{2.11}$$

donde $\rho(\theta) = 1/\pi$ es la distribución probabilidad de $\theta$. De acuerdo a la ecuación 2.7, el estimador MC para la integral 2.11, es:

$$\hat{E}_N = \frac{1}{N}\sum_{i=1}^{N} p(\theta_i) \tag{2.12}$$

$$= \frac{L}{d \cdot N}\sum_{i=1}^{N} \sin\theta_i, \quad \theta_i \sim \rho(\theta) \tag{2.13}$$

El Algoritmo 2.2 describe los pasos a seguir en esta simulación MC. La líneas 3 y 4 generan una muestra $\theta$, distribuída uniformemente en $[0, \pi]$, que se utiliza en la línea 5 para aumentar el valor de la sumatoria. Finalmente, la probabilidad de intersección se calcula y retorna en la línea 7, de acuerdo a la ecuación 2.13.

**Algoritmo 2.2:** Aguja de Buffon mediante simulación indirecta MC.

```
1 n ← 0;
2 for i ← 1 to N do
3 |   u ← Uniform();
4 |   θ ← uπ;
5 |   n ← n + sin(θ);
6 end
7 p = (L · n)/(d · N);
```

### 2.1.4 Error en estimación Monte Carlo

Considere nuevamente la integral definida

$$F = \int_a^b f(x)\, dx$$

cuyo estimador MC es

$$\hat{F}_N = \frac{1}{N}\sum_{i=1}^{N} f(x_i) \quad \text{y} \quad x_i \sim p(x)$$

De acuerdo a la Ley de los grandes números $\hat{F}_N \to E_p[F]$ a medida que $N \to \infty$. Luego, cabe preguntarse, ¿cuán grande debe ser $N$ para alcanzar una presición dada?, o de otra forma,

¿cuándo $\hat{F}_N$ es un buen estimador de $E[F]$. Si fuera posible calcular el error $\epsilon$, entonces se podría corregir la estimación y obtener el valor exacto, pues

$$F = \hat{F}_N + \epsilon$$

Observando que el muestreo de $p(x)$ produce una secuencia de variables aleatorias $x_1, x_2, \ldots, x_N$ diferente en cada experimento de estimación, se puede concluir que $\hat{F}_N$ es también una variable aleatoria. Para determinar la función de densidad de probabilidad de $\hat{F}_N$, se hace uso del Teorema del Límite Central.

**Teorema 2.2 (Límite Central)**. Sea $X_1, X_2, \ldots, X_N$, una secuencia de variables aleatorias independientes e idénticamente distribuidas con valor esperado $\mu = E[X] < \infty$ y varianza $\sigma^2 = V[x] > 0$. Además, sea $\hat{F}_N$, el estimador MC de la secuencia, y $Z_N$ la variable estandarizada:

$$Z_N = \frac{\hat{F}_N - \mu}{\sigma/\sqrt{N}} \tag{2.14}$$

Entonces, a medida que $N \to \infty$, $Z_N \to N(0, 1)$.

Luego, a medida que $N$ crece, la distribución de los promedios muestreados (estimadores MC), se aproxima a una distribución normal, cuya media es justamente la media de la población, y cuya desviación estándard es la desviación estándard de la población dividida por $\sqrt{N}$. Esto significa que a medida que $N$ crece el error de un estimador MC $\hat{F}_N$, decrece proporcionalmente con $\sqrt{N}$.

Note que, a diferencia de integración por quadratura, el análisis anterior es independiente del número de dimensiones de la integral. La tabla 2.1 lista los errores de integración para diferentes métodos de integración numérica, incluído el error MC. Como se puede observar, para una dimensión, los métodos tradicionales de integración son superiores al método MC. Éste último comienza a ser superior para dimensiones superiores o iguales a cuatro.

## 2.2 SIMULACIÓN MONTE CARLO EN FÍSICA MÉDICA

Como se mencionó en la sección anterior, el método moderno de MC tuvo su origen en cálculos computacionales para estudiar propiedades de experimentos de fisión nuclear, durante el proyecto Manhatan. Desde entonces, MC ha sido usado con éxito para resolver problemas complejos

TABLA 2.1: Errores en integración numérica para varios métodos de integración.

| Método | Una dimensión | $d$ dimensiones |
|---|---|---|
| Monte carlo | $n^{-1/2}$ | $n^{-1/2}$ |
| Trapezoide | $n^{-2}$ | $n^{-2/d}$ |
| Simpson | $n^{-4}$ | $n^{-4/d}$ |
| Gauss | $n^{-2m+1}$ | $n^{-(2m-1)/d}$ |

y multidimensionales en áreas tales como finanzas, ingeniería, medicina, biología, computación, y otras.

En particular, en este trabajo estamos interesados en simulaciones MC en física médica, como por ejemplo, simulaciones de radio terapia, experimentos de Positron Emission Tomography (PET), Rayos-X, bioluminiscencia, etc. Todas estas simulaciones utilizan las propiedades de interacción entre algún tipo de radiación con diferentes materiales, incluido tejido orgánico, para diseñar y estudiar sistemas de imageneología médica (Bushberg et al., 2002).

Uno de los software MC más usados en física de partículas es Geant4, el cual consiste en un toolkit (de programación) para la simulación del paso de fotones a través de la materia (Agostinelli & et al, 2003). Está escrito en C++ y provee un conjunto comprehensivo de clases para definir un rango amplio de experimentos de física nuclear. Geant4 incluye procesos físicos altamente complejos y completos, pero bien validados en un rango amplio de energías. Algunas áreas donde Geant4 ha sido usado son reactores nucleares, física médica, física de altas energías, y ciencias espaciales. Geant4 es sin duda uno de los proyectos más grandes y ambiciosos de su tipo.

Para construir aplicaciones en Geant4, se debe programar cada aspecto de la simulación, usando la API provista. Sin embargo, como está construido en C++, es posible definir nuevas clases y especializar la simulación a un ámbito particular. Este es el caso de Gate, la aplicación de tomografía por emisión de Geant4 (Jan & et al, 2004).

Gate permite la creación de experimentos en medicina nuclear, tales como PET, SPECT, rayos-X, y también radioterapia. Usando como base a Geant4, Gate incorpora clases para la definición de fuentes radioactivas, escáners, detectores, fantomas, movimientos de cuerpos, entre otras. Además, las simulaciones en Gate se definen usando un lenguaje simple, si necesidad de programar en C++.

Uno de los aspectos más importantes de Gate es la incorporación del tiempo en las

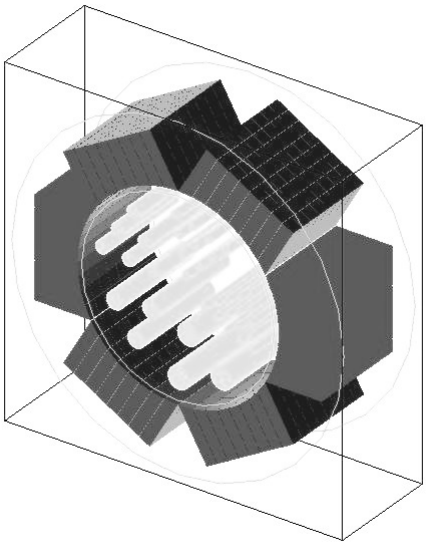


FIGURA 2.3: Visualización de un escáner PET y un fantoma en Gate.

simulaciones. En Geant4 las simulaciones son atemporales, en el sentido que no es relevante registrar el momento en que los eventos ocurren. En Gate sin embargo, si es importante incorporar la noción del paso del tiempo por tres razones. Primero, las fuentes radioactivas decaen con el tiempo, y es necesario disminuir el número de fotones que éstas emiten a medida que transcurre la simulación. Segundo, las datos recolectados por un escáner PET se construyen en base a coincidencias temporales de dos o más rayos gamas, cuando éstos son detenidos por los detectores. Y tercero, los movimientos de objetos, como rotación de escáner o de fuentes, ocurren en momentos específicos de la simulación.

La figura 2.3 muestra un modelo Gate de un escáner PET para animales pequeños, incluyendo un fantoma de tipo Derenzo..

En general, una simulación en física médica requiere simular miles de millones de fotones. Por ejemplo, un experimento PET de un animal pequeño, típicamente usa del orden de 100 uCi de radiofarmacéutico, es decir 3700000 aniquilaciones por segundo. Cada aniquilación produce dos rayos gama, los cuales a su vez pueden producir fotones secundarios al interactuar con el medio. Si el tiempo de medición es de 30 minutos, el número de fotones primarios a simular serían del orden de $13 \times 10^9$. Es por eso que se han desarrollado múltiples estrategias de paralelización tanto para Geant4 como para GATE (Staelens et al., 2006), las cuales constituyen un enorme desafío debido al tamaño y complejidad de dichos softwares. A continuación se realiza una revisión de algunas de estas estrategias.

### 2.2.1 Técnicas de paralelización en software MC para física médica

La introducción de paralelismo en código existente es un tipo de solución ampliamente explorada y ha obtenido buenos resultados de rendimiento. No obstante posee un gran problema que es la modificación de código, lo que implica que cada vez que se libera una nueva versión del software de interés, se debe introducir nuevamente paralelismo para obtener una aceleración de rendimiento. Esto ha llevado al estudio tanto de computación distribuida como de transformación automática y semiautomática de código.

El mecanismo clásico de paralelización en física de partículas consiste en distribuir los eventos en las unidades de cómputo disponibles. Esta técnica es especialmente simple de implementar cuando los eventos son atemporales, pues ellos pueden ser simulados en cualquier orden y por lo tanto pueden ser enviados a cualquier nodo de cómputo. Pero cuando los eventos dependen del tiempo, es necesario establecer un mecanismo de coordinación, no solo para distribuir los eventos, sino también para recolectar los resultados.

#### *2.2.1.1 Paralelización de Geant4*

El primer método de paralelización de Geant4 implementa un estrategia maestro-esclavo para distribuir eventos, con las tecnologías TOP-C y Marshalgen (Cooperman et al., 2006). La producción de eventos es realizada en el nodo maestro, el cual los envía uno a uno a los nodos trabajadores, quienes los simulan. Cuando un proceso trabajador termina, envía los "hist" encontrados durante la simulación al nodo maestro, el cual los combina con los resultados de otros nodos trabajadores, produciendo la salida final. La comunicación entre el proceso maestro y procesos trabajadores es siempre en base a objetos C++, lo que impone una alta sobrecarga de comunicación.

X. Dong propone un método semi automático de transformación de código para introducir capacidad de ejecución multi-hebra en Geant4 (Dong et al., 2012). El primer paso consiste en transformar el código a uno *thread-safe*, mediante la privatización de variables globales y estáticas. Luego, con la ayuda de un trazador de ejecución, los datos de sólo lectura (read-only) son detectados y mudados desde las áreas de memoria privada hacia el área de memoria compartida de las hebras.

Finalmente, para incorporar paralelismo, se escribe una nueva función `main()` en la que se crean hebras hijas, quienes ejecutan la función `main()` original de Geant4.

#### *2.2.1.2 Paralelización de Gate*

Como se dijo anteriormente, una de las grandes diferencias entre una simulación de Geant4 y una de Gate, es que en la última existe el concepto del paso del tiempo. Esto impone un ordenamiento global de los eventos que debe ser considerado por cualquier estrategia de paralelización.

En (Beenhouwer et al., 2007) se propone una técnica de particionamiento del dominio del tiempo, para dividir una simulación en varias simulaciones independientes, donde cada una simula un intervalo tiempo distinto y disjunto. La división de la simulación es generada por medio de un programa denominado *job splitter*, el cual lee la descripción de la simulación original (un script de Gate) y genera una descripción de simulación para cada una de las $N$ sub-simulaciones, como se muestra en la Figura 2.4.

Por ejemplo, suponga que se desea simular un experimento que dura dos minutos. El siguiente script Gate define el tiempo de inicio (puede no ser 0), el tiempo de término y el tiempo de un *slice*. En este caso como el slice es todo el tiempo de simulación, existe sólo uno.

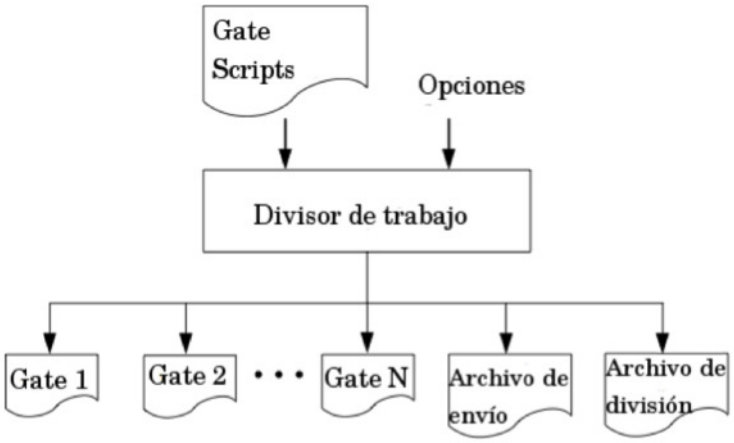



FIGURA 2.4: *Particionamiento del tiempo. Un script de Gate da origen a N nuevos scripts.*

```
# Script original
/gate/application/setTimeStart      0. s
/gate/application/setTimeStop     120. s
/gate/application/setTimeSlice    120. s
/gate/application/startDAQ
```

Luego, si se desea paralelizar esta simulación en tres simulaciones, el *job splitter* generaría los siguentes scripts, cada uno en un archivo distinto.

```
# Script 1
/gate/application/setTimeStart     0. s
/gate/application/setTimeStop     40. s
/gate/application/setTimeSlice    40. s
/gate/application/startDAQ

# Script 2
/gate/application/setTimeStart    40. s
/gate/application/setTimeStop     80. s
/gate/application/setTimeSlice    40. s
/gate/application/startDAQ

# Script 2
/gate/application/setTimeStart    80. s
/gate/application/setTimeStop    120. s
/gate/application/setTimeSlice    40. s
/gate/application/startDAQ
```

El resto de los scripts que definen la simulación no son modificados. Posteriormente, se someten a ejecución todas las sub-simulaciones a un cluster de computación, utilizando algún sistema de encolamiento. Una vez que todas las sub-simulaciones han terminado, se utiliza un mezclador de archivos de salida para generar el archivo unificado de la simulación. Los archivos de salida de las sub-simulaciones contienen identificadores de los fotones que fueron detectados por el escáner. Pero

como cada sub-simulación es totalmente independiente de las otras, dichos identifcadores se repetirán en los archivos. Luego el mezclador vuelve a enumerar los identificadores tal que éstos sean únicos en el archivo unificado. El esquema asociado se muestra en la figura 2.5.

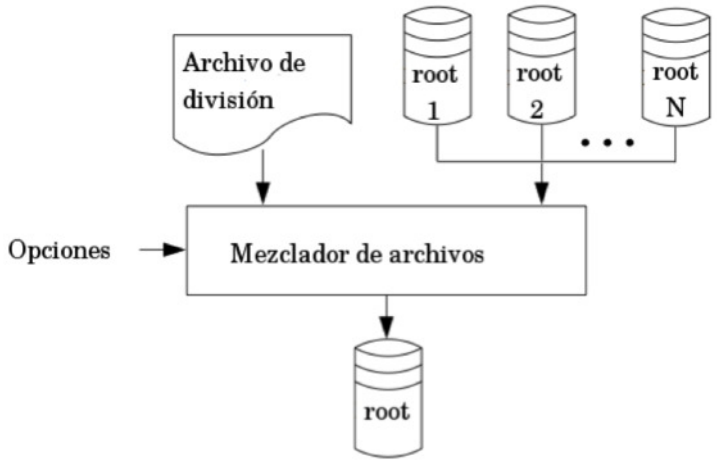



FIGURA 2.5: *Unión de resultados a través de un mezclador de archivos.*

Ya que la cantidad de cómputo que se realiza en un intervalo de tiempo dado es función de la actividad radioactiva y la vida media del isótopo, no resulta eficiente que el experimento sea dividido en porciones similares de tiempo. La figura 2.6 muestra la curva de decaimiento de una fuente radioactiva con vida media $\lambda$ y actividad inicial $A_0$. Como se puede observar, la división de la simulación en porciones iguales de tiempo produce áreas bajo la curva cada vez menores a medida que avanza el tiempo. Esto se traduce en un desbalance en la carga de trabajo, pues el número de eventos a simular decrece también con el tiempo.

Luego, para repartir el trabajo uniformemente en las simulaciones, los intervalos de simulación se dividen de acuerdo a la ecuación 2.15.

$$t_n = \frac{\log((N-n)/N)e^{-\lambda t_s} + (n/N)e^{-\lambda t_s}}{-\lambda} \tag{2.15}$$

Esta ecuación es la solución a una integral bajo la curva de decaimiento, tal que iguala la cantidad total de radioctividad en cada intervalo de tiempo. $t_s$ es el tiempo inicial del intervalo de tiempo $n$, $\lambda$ es la vida media del isótopo o también conocida por constante de decaimiento radioactivo, $t_n$ es el tiempo final de la partición $n$, y $N$ es el número de particiones requeridas. La figura 2.7 muestra las nuevas particiones para la curva anterior. Ahora, el número de eventos a simular es (aproximadamente) el mismo. Hay que recordar que el número de eventos es una variable

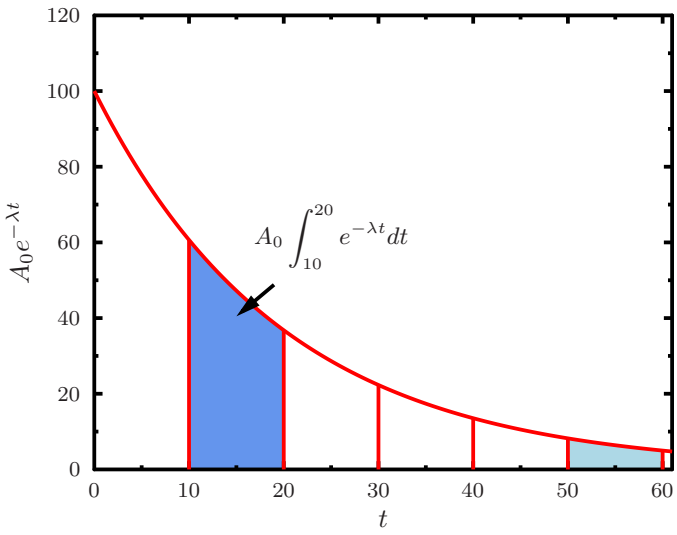



FIGURA 2.6: Función de decaimiento de una fuente radioactiva. El área bajo la curva representa la cantidad total de eventos a simlar en cada partición del tiempo. A medida que el tiempo trancurre la actividad radioactva también, y por lo tanto la carga de trabajo en cada partición.

aleatoria y por lo tanto no es necesariamente el mismo en todas las particiones.

Esta estrategia de particionamiento es bastante sencilla de implementar pues no requiere modificar el código de Gate, ni el de Geant4. Sin embargo, no es realmente una solución de computación paralela. Además, como las operaciones de división de simulaciones y mezcla de resultados no son automáticas, requiere la intervención del usuario.

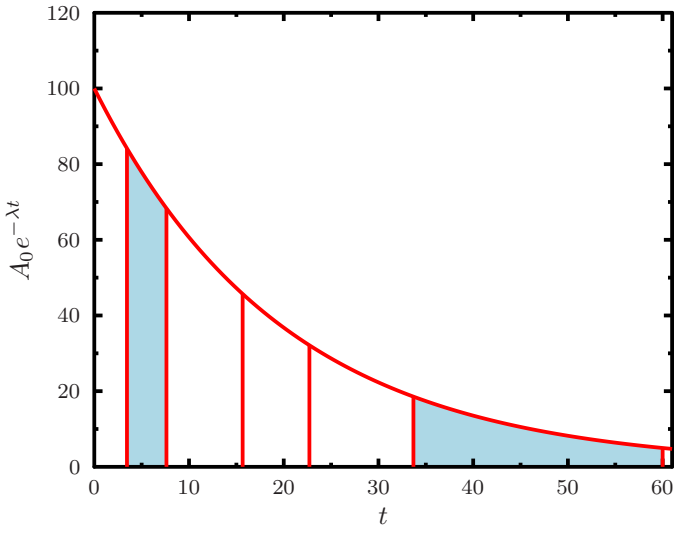



FIGURA 2.7: División temporal en base al área bajo la curva. El número de eventos a simular en cada partición es aproximadamente el mismo, produciendo una carga de trabajo homogénea entre los nodos de cómputo.

ParGate es una solución que utiliza un modelo de computación paralela para GATE

(Rannou et al., 2013). La estrategia es simple y consiste en la distribución de los eventos en tiempo de ejecución en lotes de $K$ eventos. A diferencia de la estrategia anterior, la distribución de eventos es dinámica ya que varía según la carga de cada nodo en el sistema. El parámetro $K$ puede ser variado, pero es constante en una ejecución.

La figura 2.8 muestra el modelo de computación paralela propuesto para ParGate. El modelo planteado corresponde al del tipo maestro-trabajador. El *coordinador de eventos* está encargado de realizar la asignación de los eventos a medida que los proceso trabajadores realizan trabajo y actualizan el tiempo de la simulación. El coordinador es el encargado de mantener el tiempo global de simulación. Cuando un nodo $W_i$ está disponible, le consulta al coordinador por los próximos $K$ eventos a simular. A diferencia del modelo maestro-esclavo de Geant4, el coordinador responde enviando sólo el identificador del próximo evento a simular y el actual tiempo de simulación, quedando la tarea de creación de objetos distribuida en los trabajadores. Luego, el trabajador calcula cúanto tiempo de simulación dichos eventos necesitarán, y envía el tiempo final, para que el coordinador actualice el tiempo global del sistema.

En esta estrategia, el usuario no debe realizar ningún paso adicional a lo que él haría en el caso secuencial. Note que como los identificadores de los eventos son globalmente únicos, no es necesario realizar una mezcla de los archivos resultantes.

Es importante mencionar que la estrategia está basada en procesos y fue implementada utilizando MPI (*Message Passing Interface*).

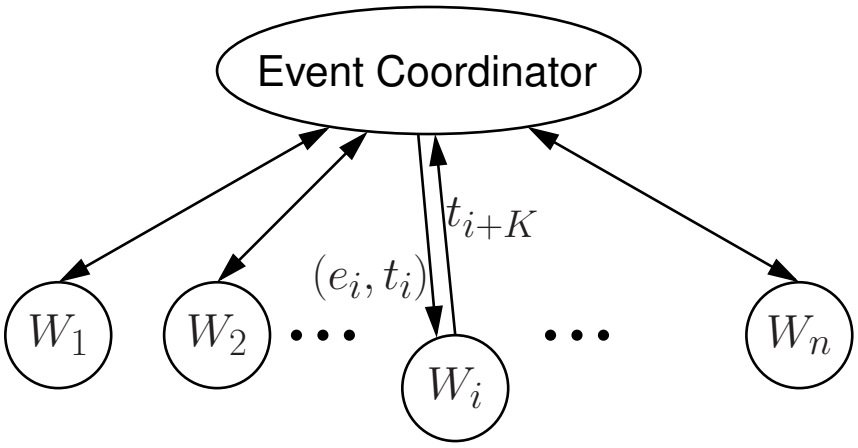



FIGURA 2.8: *Modelo de computación paralela de ParGate. El coordinado de eventos mantiene el tiempo global de simulación, y el identificador del último evento simulado. Los trabajadores pueden ore-calcular cuánto tiempo de simulación*

Siguiendo la misma idea de Dong (Dong et al., 2012) para Geant4, Torres-Tramón realiza modificaciones semi-automáticas al código de Gate, para incorporar ejecución multi-hebra (Torres-Tramon et al., 2010). La diferencia principal con el trabajo de Dong es la incorporación de mecanismos de exclusión mutua para coordinar el paso del tiempo entre las hebras. Mantener tal consistencia implica el aumento de sincronizaciones, añadiendo una mayor complejidad al modelo.

La estrategia requiere la transformación de código para convertirlo en uno *thread-safe*, es decir, uno en el cual la ejecución de mútiples hebras no produzca condiciones de carrera. La figura 2.9 muestra el esquema de computación paralela, el cual añade la exclusión mutua para manejar el identificador del último evento simulado y el actual tiempo de la simulación. A diferencia del modelo paralelo basado en proceso, no existe una hebra cordinadora, sino más bien un objeto compartido que encapsula el identificador y el tiempo de simulación, y que las hebras acceden en forma exclusiva, mediante invocación de métodos.

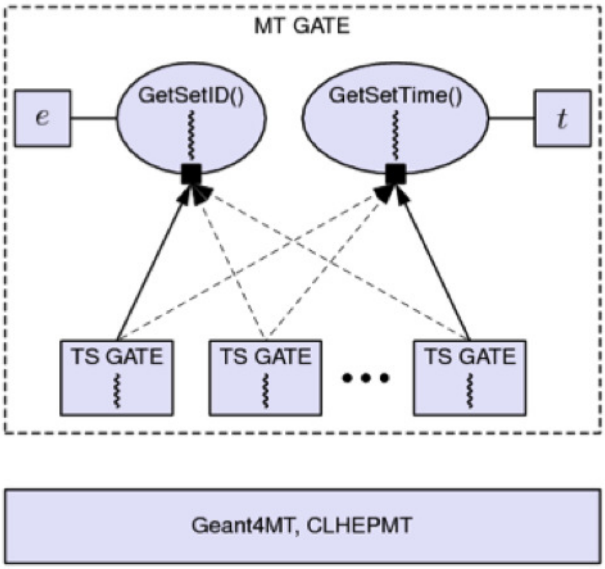



FIGURA 2.9: *Modelo de computación paralela de Multithread GATE.*

## 2.3 PROGRAMACIÓN PARALELA EN GPU

Desde hace algunos años el concepto de computación de propósito general sobre unidades de procesamiento gráfico (*General-Purpose Computing on Graphics Processing Units*, GPGPU) ha teniendo un fuerte auge y ha resultado especialmente atractivo para aplicaciones científicas que

requieren de mucho cómputo, como lo son simulaciones de sistemas complejos (Owens et al., 2008).

Los dispositivos de procesamiento gráfico ofrecen una gran cantidad de núcleos a un bajo costo. Estos núcleos, sin embargo, son más sencillos que los núcleos de propósito general de los procesadores tradicionales, y generalmente proveen una sóla unidad de control para un grupo de ellos. Dentro de las razones del auge que ha tenido esta tecnología está el desarrollo de plataformas y *frameworks* que aprovechan su capacidad. Estas plataformas además consideran las limitaciones de los dispositivos, debido a que no todo problema obtendrá un buen rendimiento con GPU.

Las dos tecnologías más relevantes para la programación en GPU son CUDA y OpenCL. El primero ofrece una plataforma de programación para dispositivos de procesamiento gráfico NVIDIA, y por lo tanto optimizado para dichos dispositivos. Por el contrario OpenCL ofrece un marco de desarrollo estándar, heterogéneo y multiplataforma.

### 2.3.1 Método Monte Carlo e implementaciones en GPU

La simulación de eventos discretos por medio de GPU es un tema que ha sido estudiado por algunos autores. El trabajo de Park propone la construcción de un *framework* que actúe como ayuda para la simulación de eventos por medio de tarjetas gráficas (Park & Fishwick, 2010). La implementación de dicho *framework* utiliza un método de planificación de eventos y colas, planificando según el tiempo que toman en ejecutarse los elementos. No obstante, la planificación realizada agrega un importante *overhead* en desmedro del rendimiento computacional, alcanzando un *speedup* máximo de 10x.

Por otro lado, Alerstam realiza una implementación para GPU de simulaciones de bioluminiscencia en capas de piel (Alerstam et al., 2010). En él se analizan las oportunidades de optimización reutilizando código C obtenido desde (Wang & Jacques, 1992), el cual es escrito nuevamente para CUDA. A través de su investigación se aborda el modelo de computación de CUDA, el impacto de rendimiento de las instrucciones atómicas, optimizaciones de memoria compartida y la utilización de múltiples GPU para procesamiento. Los resultados obtenidos varían de 64x a 97x sin ningún tipo de optimización. Sin embargo al realizar optimizaciones de memoria el rendimento aumenta hasta 621x para tarjetas de la generación Fermi.

A pesar del gran rendimiento del trabajo realizado por Alerstam, la implementación

carece del nivel de abstracción que se requiere para extender el modelo fácilmente a otros tipos de simulaciones. Al ser una transformación directa del código de Wang, no propone un modelamiento abstracto del universo de simulación ni un modelo estándar para aplicar procesos físicos a las partículas. Además las optimizaciones están totalmente focalizadas en mejorar los tiempos de instrucciones atómicas lo cual las acopla directamente a la simulación particular que implementan.

# CAPÍTULO 3. DISEÑO DEL MODELO DE SIMULACIÓN

En este capítulo se describen las reglas físicas que gobiernan la simulación de partículas en bioluminiscencia. Posteriormente, se detalla el trabajo realizado en la propuesta de (Lagos, 2012) para finalmente presentar una estrategia de paralelización en base a lo descrito.

## 3.1 MODELO PARA BIOLUMINISCENCIA

Como se mencionó en el Capítulo 1, la bioluminiscencia es una técnica utilizada en imagenología molecular para el estudio no invasivo de procesos biológicos. Debido a las aplicaciones posibles, resulta interesante la capacidad de predecir el comportamiento de las partículas en diferentes tejidos sin la necesidad de utilizar seres vivos.

Para realizar una simulación de partículas para bioluminiscencia a través del método Monte Carlo es necesario detallar las reglas físicas del fenómeno, las cuales se presentan a continuación.

### 3.1.1 Generación de un fotón

El lanzamiento de un fotón es el proceso a través del cual una *fuente* produce un nuevo evento el cual entra al ciclo de simulación.

Las fuentes son objetos definidos por el usuario y están encargados de producir fotones de luz. Estos objetos pueden estar dispersos en el universo o ser puntuales. Además las fuentes se caracterizan por ser direccionadas o isotrópicas, es decir, pueden emitir fotones restringidos a un rango de direcciones o emitir fotones con igual probabilidad en todas las direcciones. Una típica

simulación puede involucrar una o muchas fuentes distribuidas de forma arbitraria en el universo. Así mismo, cada una de las fuentes puede realizar cientos de millones de lanzamientos.

### 3.1.2 Cálculo del paso del fotón

El proceso de step o paso del fotón, es aquel en el cual se determina el largo del desplazamiento (*step size*) de una partícula. Para obtener el largo del desplazamiento, se realiza el muestreo de una variable aleatoria que se distribuye de acuerdo a la distribución de probabilidad del desplazamiento libre del fotón según la ecuación 3.1:

$$s = \frac{-ln(\xi)}{\mu_t} \tag{3.1}$$

donde $\mu_t$ corresponde al coeficiente de interacción del tejido en el que se encuentra la partícula y $\xi$ corresponde a un número aleatorio uniformemente distribuido en $[0, 1]$. El coeficiente de interacción es la suma del coeficiente de absorción $\mu_a$ y el coeficiente de dispersión $\mu_s$.

El detalle del proceso de step será retomado en el capítulo 4.

### 3.1.3 Movimiento de un fotón

Una vez que el largo de desplazamiento $s$ ha sido calculado, es posible realizar el movimiento del fotón en el tejido. Para ello se actualiza su posición según la ecuación 3.2:

$$\begin{aligned} x &\leftarrow x + \mu_x s \\ y &\leftarrow y + \mu_y s \\ z &\leftarrow z + \mu_z s \end{aligned} \tag{3.2}$$

El vector unitario $[\mu_x\ \mu_y\ \mu_z]^t$ corresponde a la dirección de desplazamiento que posee la partícula. En consecuencia, los valores en el lado izquierdo son sustituidos por la nueva posición, la cual es obtenida añadiendo la ponderación del *step size* y la dirección de la partícula.

### 3.1.4 Procesos de reflexión y refracción

Producto del desplazamiento, un fotón puede llegar al límite del cuerpo que lo alberga, en cuyo caso podría reflectarse o transmitirse. Para tomar una decisión se debe calcular las probabilidades para cada evento.

La probabilidad de que el fotón sea internamente reflejado depende del ángulo de incidencia $\alpha_i$ entre la dirección del fotón y la normal a la superficie de reflexión, y de las propiedades del material. Claramente, $\alpha_i = 0$ significa que la incidencia es ortogonal.

Para una simulación de capas de tejidos, donde cada una de las capas es perpendicular al eje $z$, el ángulo de incidencia es calculado a través de la ecuación 3.3.

$$\alpha_i = \cos^{-1}(|\mu_z|) \tag{3.3}$$

Por otro lado, la ecuación de Snell describe la relación entre el ángulo de incidencia $\alpha_i$ de una partícula, el ángulo de transmisión $\alpha_t$ y los índices refractivos del medio de incidencia y transmisión, $n_i$ y $n_t$ respectivamente, de acuerdo a la siguiente igualdad:

$$n_i \sin \alpha_i = n_t \sin \alpha_t \tag{3.4}$$

Con los ángulos de incidencia y transmisión se procede a determinar la reflectancia interna, la cual es un promedio de las reflectancias para las dos direcciones de polarización ortogonales. La reflectancia interna es calculada a través de la fórmula de Fresnel como muestra la ecuación 3.5.

$$R(\alpha_i) = \frac{1}{2}\left[\frac{\sin^2(a_i - a_t)}{\sin^2(a_i + a_t)} + \frac{\tan^2(a_i - a_t)}{\tan^2(a_i + a_t)}\right] \tag{3.5}$$

Por último, se determina si el fotón es reflectado o transmitido. Para ello, se genera un número aleatorio $\xi$ distribuido uniformemente, y se compara con la reflectancia interna. La decisión queda expresada de la siguiente manera:

- Si $\xi \leq R(\alpha_i)$, entonces el fotón se reflecta.
- Si $\xi > R(\alpha_i)$, entonces el fotón se transmite al siguiente tejido.

Cuando la partícula es reflejada su nueva dirección es obtenida cambiando el signo de la componente $\mu_z$. En caso de ser transmitida, la nueva dirección está dada por el ángulo de transmisión de la ecuación de Snell.

### 3.1.5 Proceso de Absorción

Una vez que un fotón ha terminado de desplazarse, su energía es disminuida lo cual es el resultado de la interacción que ha tenido con el medio. La energía que pierde la partícula es depositada en la posición final que alcanza. La cantidad de energía que pierde la partícula, $\Delta W$, se obtiene de acuerdo a la siguiente ecuación:

$$\Delta W = W \frac{\mu_a}{\mu_t} \tag{3.6}$$

donde $W$ es la energía que trae el fotón. La energía perdida es depositada en la posición final del fotón, por ejemplo, para un universo discretizado de manera cilíndrica en función del eje $z$ y el radio $r$ de las coordenadas $x$ e $y$, la absorción será registrada en una matriz según la ecuación 3.7.

$$A(r, z) \leftarrow A(r, z) + \Delta W \tag{3.7}$$

Finalmente, el peso del fotón es actualizado disminuyendo su energía según:

$$W \leftarrow W - \Delta W \tag{3.8}$$

### 3.1.6 Proceso de Dispersión

La dispersión es el cambio de dirección del desplazamiento de la partícula debido a la interacción que posee con los tejidos. Este proceso se lleva a cabo una vez que ha sido disminuido el peso de la partícula. El cálculo de la nueva dirección requiere del muestreo de dos variables aleatorias:

1. El ángulo de deflección $\theta \in [0, \pi)$.

2. El ángulo azimutal $\psi \in [0, 2\pi)$.

El coseno del ángulo de deflección puede ser expresado por medio de una variable aleatoria $\xi$, tal como muestra la ecuación 3.9, en la cual $g$ es el factor de anisotropía.

$$\cos\theta = \begin{cases} \frac{1}{2g}\left(1+g^2-\left[\frac{1-g^2}{1-g+2g\xi}\right]^2\right) & \text{si } g > 1 \\ 2\xi - 1 & \text{si } g = 0 \end{cases} \tag{3.9}$$

Por otro lado, el ángulo azimutal está uniformemente distribuido en el intervalo $[0, 2\pi)$, debido a lo cual se determina según la ecuación 3.10.

$$\psi = 2\pi\xi \tag{3.10}$$

Cuando han sido obtenidos los ángulos de dispersión, es posible calcular la nueva dirección del fotón:

$$\begin{aligned} \mu'_x &= \frac{\sin\theta}{\sqrt{1-\mu_z^2}}(\mu_x\mu_z\cos\psi - \mu_y\sin\psi) + \mu_x\cos\theta \\ \mu'_y &= \frac{\sin\theta}{\sqrt{1-\mu_z^2}}(\mu_y\mu_z\cos\psi + \mu_x\sin\psi) + \mu_y\cos\theta \\ \mu'_z &= -\sin\theta\cos\psi\sqrt{1-\mu_z^2} + \mu_z\cos\theta \end{aligned} \tag{3.11}$$

En caso de que el ángulo de la partícula sea muy cercano a una incidencia normal en la superficie, la fórmula a usar debería ser:

$$\begin{aligned} \mu'_x &= \sin\theta\cos\psi \\ \mu'_y &= \sin\theta sin\psi \\ \mu'_z &= SIGN(\mu_z)\cos\theta \end{aligned} \tag{3.12}$$

### 3.1.7 Fin de la vida de un fotón

Existen dos maneras en las cuales se finaliza la simulación de un fotón. La primera es cuando éste es transmitido fuera del universo de simulación, independientemente de la energía que posea. La segunda es cuando su energía es menor que un umbral establecido. Como se puede observar en las ecuaciones 3.6 y 3.8, la pérdida de energía siempre es una fracción de la energía actual del fotón, y por lo tanto, la simulación podría continuar indefinidamente. Cuando la energía traspasa

el umbral mencionado, la partícula debe ser finalizada, pues sus aportes de energía no constituyen información útil para la simulación.

No obstante, es necesario asegurar la conservación de energía de la partícula sin distorsionar la distribución de energía depositada en los tejidos. Para ésto se utiliza una técnica de ruleta a través del muestreo de una variable aleatoria. La técnica otorga al fotón una probabilidad de sobrevivir, lo cual incrementa en cierta medida $m$ (por ejemplo $m = 10$) su energía. Si la partícula no sobrevive a la ruleta entonces el peso de la partícula es reducido a cero, tal como muestra la ecuación 3.13.

$$W = \begin{cases} mW & \text{si } \xi \leq 1/m \\ 0 & \text{si } \xi > 1/m \end{cases} \tag{3.13}$$

## 3.2 DISEÑO BASADO EN OBJETOS CONCURRENTES

El proceso de bioluminiscencia descrito anteriormente fue modelado por (Lagos, 2012). El objetivo de dicho trabajo fue incorporar en la etapa temprana de diseño las oportunidades de concurrencia inherentes al problema. Para esto se utilizó una metodología orientada al objeto que además identificaba los objetos activos y pasivos del sistema. En diseño orientado al objeto, los objetos activos son aquellos que presentan un comportamiento autónomo y se comunican con otros objetos proactivamente. La identificación de estos objetos es altamente relevante, pues de ellos depende el nivel de concurrencia del sistema.

Para que el diseño resultante fuese transformado en forma directa en una implementación fue necesario contar con un lenguaje de programación que soportara concurrencia a nivel de clases. $\mu$C++ es un lenguaje de programación que provee a nivel sintáctico la capacidad de concurrencia (Buhr et al., 1992), y por lo tanto no es necesaria la utilización de bibliotecas externas para incluir ejecución multi-hebra.

Una de las limitaciones del trabajo de Lagos es que su implementación para bioluminiscencia permite sólo la ejecución de experimentos en dos dimensiones. No obstante, dada la prolijidad del modelo obtenido se ha decidido estudiar cómo extenderlo a tres dimensiones y posteriormente analizar si es posible recoger sus principales ideas para una estrategia de paralelización

en GPU.

### 3.2.1 Extensión a tres dimensiones

La estrategia de (Lagos, 2012) realiza una clara distinción entre el manejo de eventos, cuerpos y procesos físicos. La principal característica de su modelo es que, por medio de las super clases definidas, las implementaciones que sean realizadas son nativamente concurrentes, ya que $\mu$C++ permite de manera simple y elegante definir objetos que poseen exclusión mutua de manera nativa.

La figura 3.2 presenta el diagrama de clases asociado a su diseño. En este diagrama se muestra una arquitectura genérica, sin hacer referencia a ninguna implementación en particular. Las dos clases más importantes del diagrama son `TrackingHandler` y `EventDispatcher`, ambas son *tareas*. Las instancias de `TrackingHandler` son las encargadas de realizar toda la simulación de eventos, los cuales son obtenidos desde un *monitor* que es instancia de la clase `EventBuffer`. Por otro lado, las instancias de `EventDispatcher` tienen como propósito obtener nuevos eventos y hacer ingreso de ellos en instancias de `EventBuffer`.

El diagrama de secuencia de la figura 3.1 permite entender cómo se lleva a cabo el proceso de transporte de fotones. En dicho diagrama, es posible apreciar que los objetos de la clase `TrackingHandler` se encargan de comandar la simulación de cada partícula. En primer lugar se determinan los cambios que han de ser aplicados al fotón. Posteriormente se lleva a cabo el movimiento del fotón hasta su siguiente posición. En caso de que intente escapar del cuerpo, se aplican los procesos de reflexión o refración según corresponda. Luego son aplicados los cambios de dispersión y absorción. Finalmente, se determina si la partícula sobrevive y se guardan los resultados.

La extensión de este modelo a bioluminiscencia agrega los tipos de cuerpo: *círculo* y *rectángulo*. Del mismo modo, se ha definido la clase *Point2D* a través de la cual se realiza el manejo de puntos en el universo.

A grandes rasgos, los pasos que deben completarse para extender el software a tres dimensiones son:

1. Creación de una clase *Point3D* y una *PointFactory* para abstraer la utilización de puntos en

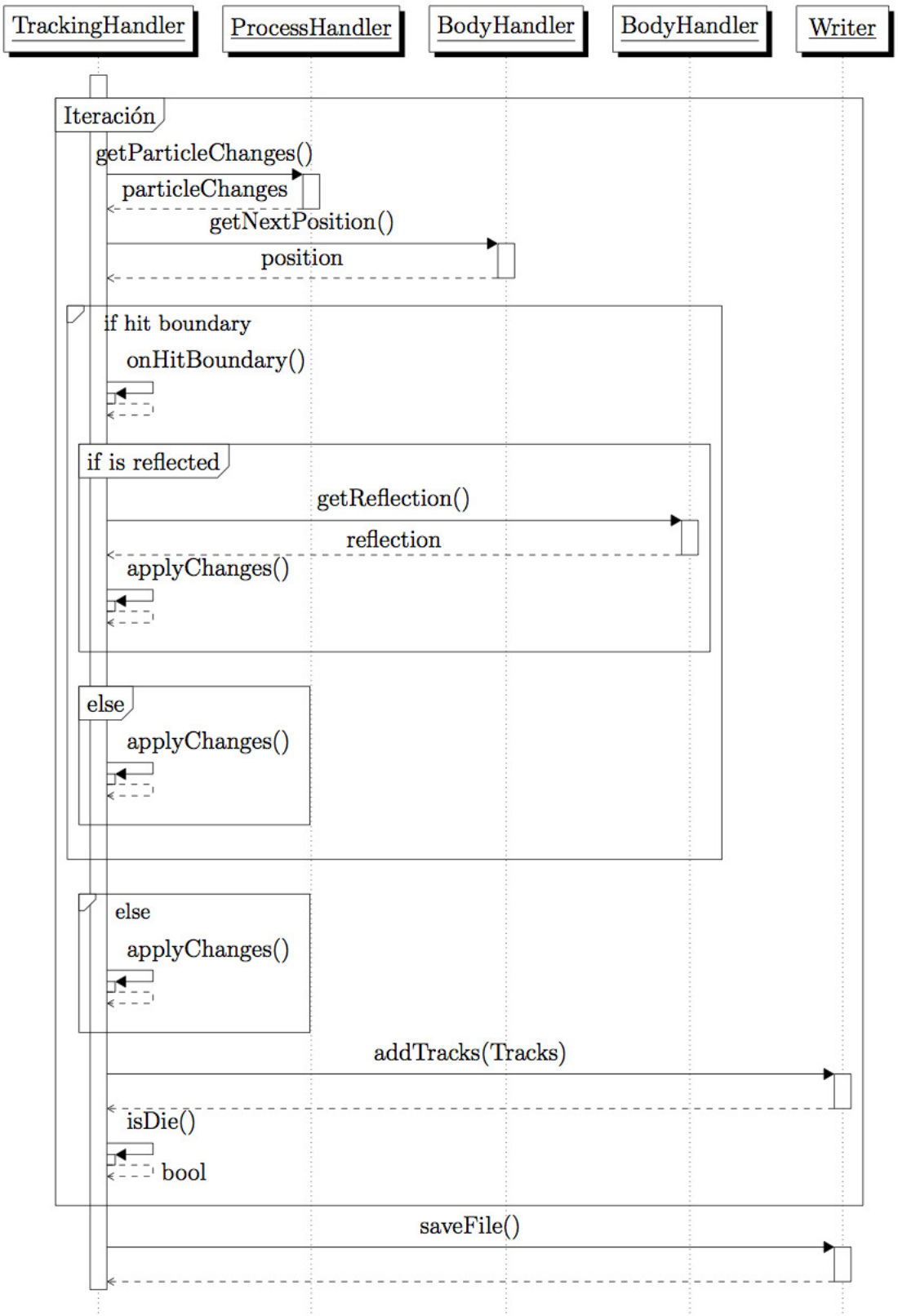



FIGURA 3.1: *Diagrama de secuencia relacionado al tranporte de fotones.*

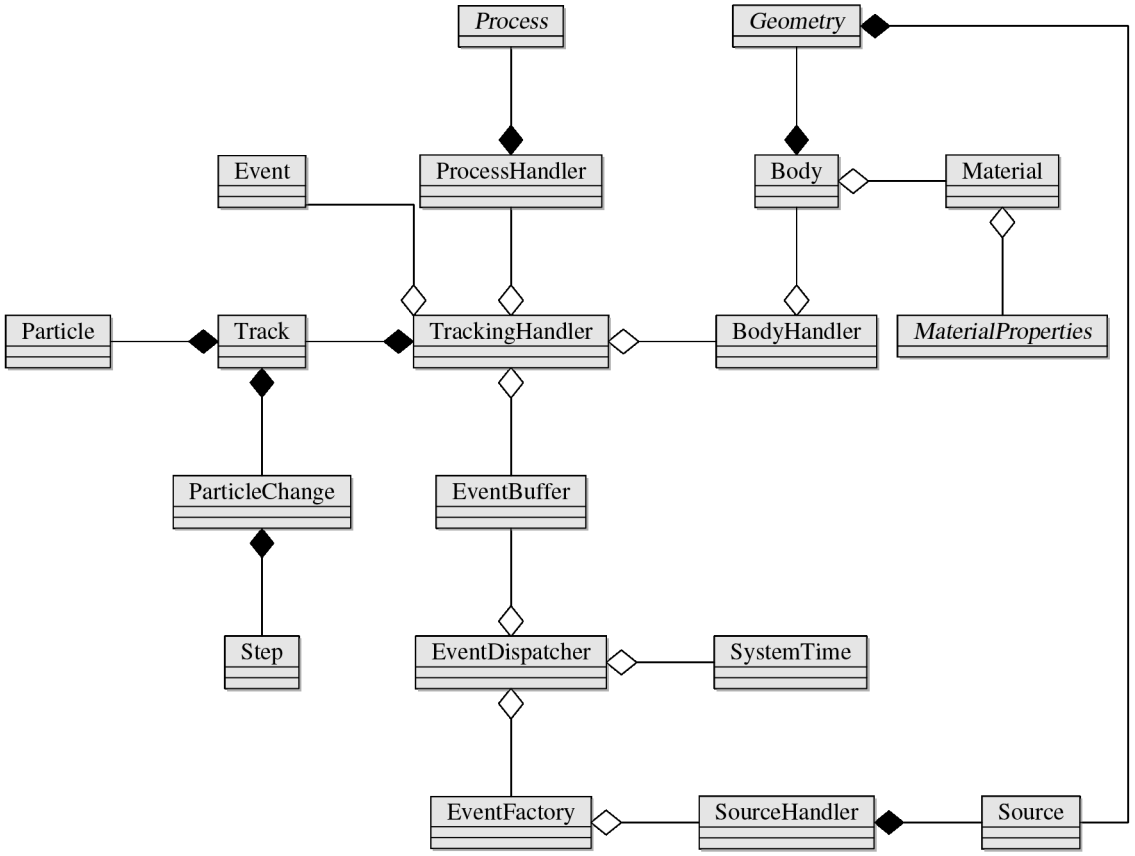



FIGURA 3.2: *Diagrama de clases de modelo de simulación Monte Carlo para objetos concurrentes.*

diferentes dimensiones.

2. Creación de una clase *Box* la cual extenderá a *Geometry*, permitiendo crear universos en tres dimensiones.

3. Extensión de fuentes de eventos a tres dimensiones, agregando además fuentes direccionadas.

4. Correcciones en los procesos físicos.

5. Correcciones menores de código escrito para dos dimensiones.

### 3.2.2 Adaptación a GPU

Una vez indicadas las reglas de simulación para bioluminiscencia y extendido a tres dimensiones el modelo de objetos concurrentes, es posible definir la manera en que éste será adaptado para su utilización.

En programación GPU el paralelismo ocurre fundamentalmente a través de *kernels*. Un *kernel* es una función que se ejecuta en paralelo en el dispositivo de procesamiento gráfico pero su llamado depende de instrucciones definidas en el código secuencial. La figura 3.3 muestra un ejemplo simple de *kernel* escrito en lenguaje CUDA. El código tiene como propósito realizar la multiplicación de dos vectores.

```
__global__ void Multiplicacion(int* vector1, int* vector2, int* resultado) \{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    resultado[i] = vector1[i] * vector2[i];
}
```

FIGURA 3.3: *Ejemplo de Kernel. El código es ejecutado en paralelo al unísono para obtener la multiplicación de dos vectores.*

De acuerdo con esto, la manera natural en que el modelo de objetos concurrentes puede ser adaptado, es ejecutando en paralelo a través de *kernels* los procesos físicos que afectan a las partículas.

## 3.3 ARQUITECTURA DE LA SIMULACIÓN

Las nuevas capacidades provistas por CUDA permiten la construcción de software orientado a objetos. Si bien existen características fundamentales que aún no están soportadas, las especificaciones actuales son suficientes para su utilización, y en particular, para adaptar el modelo de objetos concurrentes a una nueva arquitectura.

La idea central de la arquitectura propuesta en el presente trabajo, es desacoplar las entidades de los procesos físicos que pueden afectarlas.

Una simulación de partículas puede ser descrita en términos genéricos como un conjunto de procesos físicos que son aplicados a un objeto hasta que se cumple algún tipo de condición. Bajo esta premisa es posible reutilizar modelos orientados a objetos propuestos en otros trabajos e implementar los procesos físicos en una tecnología en particular, en este caso GPU.

La Figura 3.4 muestra un esquema de cómo se organizaría una simulación. Tanto los objetos como los procesos físicos deberán ser alojados en memoria global de la GPU, no obstante, la secuencia de pasos a seguir para completar el proceso de bioluminiscencia es determinada desde el *host*.

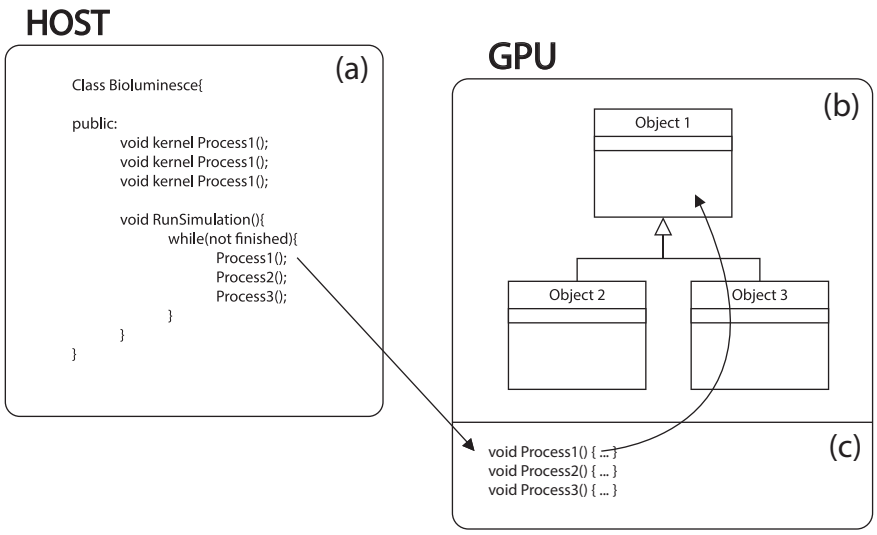



FIGURA 3.4: *Esquema de la nueva arquitectura propuesta para simulación.*

Esta arquitectura permite definir simulaciones de manera genérica, sin pensar en ninguna tecnología en particular. Las simulaciones deben escribirse como una secuencia de pasos comandados desde el *host*. Es decir, en vez de escribir un algoritmo específico para GPU, se escribe uno consistente en una secuencia de procesos físicos, cada uno de los cuales opera sobre el conjunto de objetos que ya residen en GPU.

Este modelo difiere de los modelos propuestos hasta ahora en los siguientes aspectos:

1. El esquema es genérico para simulación Monte Carlo de paso de partículas a través de la materia. La vida de un fotón y su viaje a través de los tejidos se modela como una repetición de una secuencia de procesos físicos, los cuales pueden incluso aplicarse en base a una cierta probabilidad. Por ejemplo, en PET un rayo gama tiene asociado varios procesos físicos en cada paso, pero dependiendo de una cierta probabilidad, sólo uno de ellos actúa cada vez. En el caso de bioluminiscencia, la refracción y reflexión sólo se aplican cuando un fotón intenta escapar de un tejido.

2. La decisión de qué proceso aplicar queda fuera de la GPU. Esto permite someter masivamente fotones que están realizando el mismo proceso. Esto es diferente a otros modelos donde la decisión se toma dentro de la GPU, provocando divergencia entre las hebras y por ende una pérdida de rendimiento.

3. No existe transferencia de particulas entre el dispositivo y el *host*. Una vez que una partícula muere, el objeto asociado es reutilizado por una nueva partícula. Luego, un objeto C++ que

representa a un fotón no es destruido cuando el fotón pierde toda su energía, sino que es reciclado como un nuevo fotón, evitando así la sobrecarga de creación y destrucción de objetos, y también la transferencias de objetos entre GPU y *host*.

### 3.3.1 Restricciones de clases

Un software para GPU se divide en dos partes fundamentales: código huesped (*host code*) y código de dispositivo (*device code*). El código huesped puede ser totalmente orientado a objetos puesto que es compilado a través de herramientas comunes y corrientes. No obstante, el código de dispositivo no puede ser totalmente orientado a objetos, pues la tecnología actual de GPU no lo permite. Considerando las limitaciones tecnológicas actuales de GPU, el diseño del problema no debe:

1. Utilizar recursión
2. Definir clases abstractas
3. Realizar llamados a funciones no definidas en el mismo archivo fuente (esta es una limitación del compilador).
4. Definir métodos de objetos con el calificador `__global__`.

De acuerdo a estas limitaciones, es aún posible adaptar casi cualquier modelo existente orientado a objetos para que pueda ser ejecutado en un dispositivo gráfico.

### 3.3.2 Definición de procesos

Los procesos físicos serán escritos como funciones que aplican algún cambio al conjunto de partículas de la simulación, en paralelo. La Figura 3.5 representa un proceso físico actuando en paralelo sobre una multitud de fotones. Para que este modelo sea eficiente en GPU, el proceso físico debe ser escrito tal que evite divergencias en las hebras.

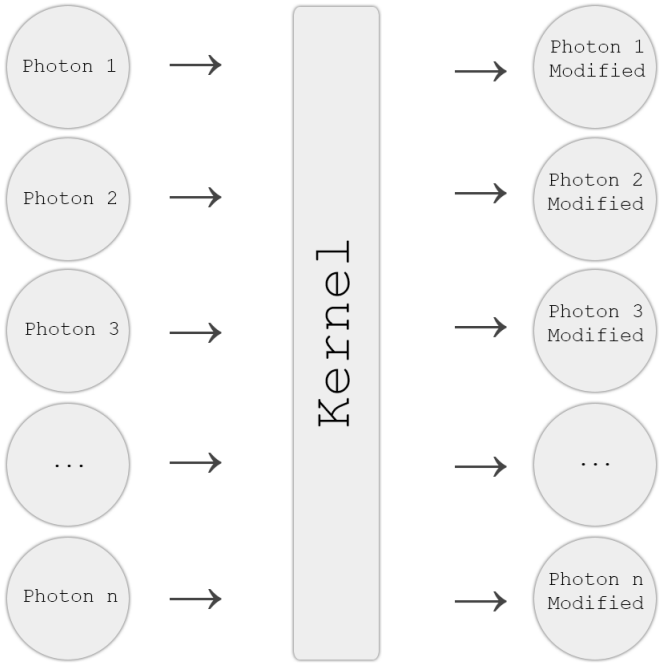



FIGURA 3.5: *Esquema de aplicación de procesos físicos a través de kernels.*

La estandarización de los parámetros de entrada para los kernels permitirá desacoplar totalmente el código huesped del código de dispositivo, liberando al programador de la responsabilidad de escoger los parámetros adecuados para un proceso en específico.

Atendiendo las limitaciones tecnológicas, en este trabajo todos los *kernels* seguirán el patrón presentado en la figura 3.6

```
__global__ void ProcesoFisico(Particle* particles, Simulation* simulation, Random* random) \{
    // Code...
\}
```

FIGURA 3.6: *Ejemplo de definición de un proceso físico a través de un kernel.*

Con respecto a los parámetros del kernel, `particles` corresponde a un arreglo con la agrupación total de partículas. Durante la simulación existen tantas partículas como hilos definidos para la ejecución de *kernels*. Una vez que la simulación de una partícula finaliza, la posición del arreglo es reutilizada, reiniciando sus datos. El parámetro `simulation` corresponde a un objeto con los datos de la simulación, tanto del universo como aquellos asociados a las mediciones. Finalmente el parámetro `random` es un puntero a un objeto con $n$ máquinas generadoras de números pseudo-aleatorios, donde $n$ es el número de hilos con que se ha de ejecutar el proceso.

# CAPÍTULO 4. IMPLEMENTACIÓN DEL SISTEMA DE SIMULACIÓN

## 4.1 EXTENSIÓN DE OBJETOS CONCURRENTES

Como fue mencionado en el Capítulo 3, se pretende utilizar el trabajo presentado por (Lagos, 2012) como base para una nueva arquitectura. Como parte del estudio a realizar y con el objetivo de analizar las posibilidades que dicho trabajo establece, se ha decidido extenderlo a tres dimensiones. A continuación se presenta el detalle de la extensión.

### 4.1.1 Puntos en tres dimensiones

Fue necesaria la creación de una nueva clase $Point3D$, la cual es bastante similar a la ya existente $Point2D$. La diferencia fundamental radica en la adición de la componente $z$, para la cual se crearon sus métodos *getters* y *setters* asociados.

El alcance del trabajo original está limitado a la creación de experimentos en dos dimensiones. En consecuencia existen vacíos y en algunos casos código *escrito en duro*, es decir, que no hace una adecuada utilización de interfaces y clases abstractas.

El caso más frecuente encontrado fue la utilización directa de la clase $Point2D$, lo cual dificulta la extensión a tres dimensiones. La manera en que el problema fue solucionado es a través de la creación de una *Fábrica de Puntos* (*PointFactory*), la cual está definida en la figura 4.1.

```
1
2 class PointFactory {
3     private:
4         static int dimensions;
5     public:
6         PointFactory();
7         static Point* getPoint();
8         static Point* getPoint(int dim);
9         static Point* getPoint(double n_x, double n_y);
10        static Point* getPoint(double n_x, double n_y, double n_z);
11        static Point* getPoint(Point* p);
12 }
```

FIGURA 4.1: *Definición de la clase PointFactory.*

La clase *PointFactory* permite la fácil creación o clonación de puntos, liberando al programador de la necesidad de indicar o conocer la cantidad de dimensiones del experimento. Dichas creaciones o clonaciones se realizan a través de las distintas sobrecargas del método *getPoint*. Es necesario indicar a la clase la cantidad de dimensiones que posee la simulación, para lo cual se debe asignar correctamente el valor de la variable estática *dimensions* una vez que el archivo de entrada de la simulación es procesado.

### 4.1.2 Cuerpos en tres dimensiones

De acuerdo con el diagrama de clases para objetos concurrentes, para la implementación de cuerpos en tres dimensiones sólo es necesaria la creación de una clase que implemente a *Geometry*. Se ha decidido crear la clase *Box*, la cual permite la definición de cajas en tres dimensiones.

Básicamente, se han implementados los siguientes métodos:

1. `bool insideGeometry(Point * point)`: permite determinar si un punto está dentro o fuera de la caja.

2. `Point* getIntersectionPoint(Point * pointOne, Point * pointTwo)`: permite determinar el punto de intersección de una recta y la caja.

3. `Point* getRandomPoint()`: utilizado para obtener un punto interior aleatorio.

4. `Point* getRandomDirection()`: utilizado para obtener una dirección al azar.

5. `Point* getRandomDirection(Point * direction, double angularAperture)`: utilizado para obtener una dirección aleatoria considerando una dirección de foco y un ángulo de apertura.

6. `Point* getReflection(Point* pointOne, Point* pointTwo, Point* direction)`: permite obtener la dirección de un punto al reflectarse en alguna superficie.

7. `double getIncidenceAngle(Point * point, Point * direction)`: permite obtener el ángulo de incidencia cuando una particula intenta escapar del cuerpo.

### 4.1.3 Fuentes de luz

Una vez que existe la posibilidad de defininir cuerpos en tres dimensiones, la definición de fuentes de luz en tres dimensiones es inmediata. Esto es debido a que tanto los cuerpos (`Body`) como las fuentes (`Source`) hacen uso de alguna clase que implemente a `Geometry`.

No obstante se ha planteado la necesidad de crear fuentes de luz direccionadas, permitiendo escoger una dirección en el espacio y un ángulo de apertura. Para agregar dicha característica, se han agregado los atributos `direction` y `angularAperture` a la clase `Source`. Posteriormente se adapta el método `getRandomDirection()`, el cual ahora emitirá partículas en el espacio definido por la dirección y apertura angular.

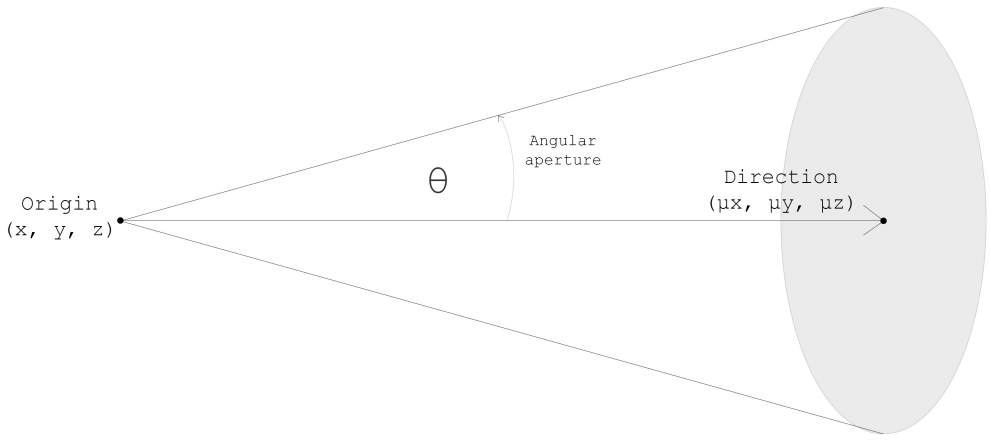



FIGURA 4.2: *Fuente direccionada. Las fuentes están compuestas de un origen, una dirección y una apertura angular.*

La figura 4.2 muestra la representación de una fuente direccionada de fotones. Los objetos se componen de un origen y una dirección, además de una apertura angular con el propósito de ampliar el rango de emisión de partículas. En la figura, cada punto en la base del cono es una posible dirección en la cual la fuente puede emitir un evento. A medida que la apertura angular $\theta$ se amplía, la dirección de emisión se vuelve más isotrópica. La variable $\theta$ debe pertenecer al intervalo $[0, \pi)$.

## 4.2 REUTILIZACIÓN DEL MODELO EN GPU

Una vez que el modelo descrito anteriormente ha sido extendido a tres dimensiones, se decide cuáles y de qué manera sus componentes serán reutilizadas para una implementación en tarjetas gráficas.

En consecuencia, tomando las ideas del modelo de objetos concurrentes, la implementación de la nueva arquitectura se dividirá en tres partes:

1. Definición del universo.
2. Definición de los procesos físicos.
3. Control de la simulación.

En las siguientes secciones se detallan cada uno de los puntos mencionados.

### 4.2.1 Implementación de clases

Para realizar la implementación de las clases se han tomado las componentes más relevantes que permiten construir un modelo simple para la simulación.

Considerando a su vez la limitaciones actuales de GPU, no será posible construir las clases abstractas que permiten una ideal definición orientada a objetos. En su lugar, a continuación se muestran cuáles son las clases más representativas y de qué manera han sido implementadas o adaptadas.

#### *4.2.1.1 Implementación de clase Point*

La clase `Point` se ha mantenido tal cual ha sido definida en la clase `Point3D` extendida desde el modelo de objetos concurrentes. Los únicos cambios que ha sufrido corresponden a aquellos necesarios para permitir su correcto funcionamiento en GPU.

```
class Point {

    private:
        float x;
        float y;
        float z;

    public:
        __host__ __device__ Point();
        __host__ __device__ Point(float x, float y, float z);

        __device__ float getX();
        __device__ float getY();
        __device__ float getZ();

        __device__ void setX(float x);
        __device__ void setY(float y);
        __device__ void setZ(float z);
};
```

FIGURA 4.3: *Definición de la clase Point.*

En la figura 4.3 puede verse que los constructores permiten la definición de objetos tanto en el *host* como en el dispositivo. Sin embargo, los métodos para obtener y asignar las componentes de un punto sólo son accesibles cuando los objetos están albergados en memoria GPU.

#### *4.2.1.2 Implementación de clase Particle*

La clase `Particle` es aquella que representa un fotón o partícula que debe ser simulado. Según la arquitectura propuesta las partículas se crean y destruyen sólo una vez durante la simulación. Cuando una partícula pierde toda su energía o escapa completamente del universo de simulación, ésta es reciclada, es decir, es inicializada nuevamente evitando de esta manera cualquier tipo de sobrecarga debido a la creación y destrucción de objetos.

Por otro lado, estos objetos permanecen siempre en memoria del dispositivo ahorrando de esta manera el costo de transferencia.

A partir de lo anterior la nueva clase `Particle` provee los métodos para re-inicializar,

```
class Particle {

private:
    Point position;
    Point direction;
    float energy;
    Step step;
    int body_id;
    bool disposed;

public:
    __host__ __device__ Particle();
    __device__ Particle(Point position, Point direction, float energy);

    __device__ Point getPosition();
    __device__ void setPosition(Point position);

    __device__ Point getDirection();
    __device__ void setDirection(Point direction);
    __device__ void setDirection(float x, float y, float z);

    __device__ float getEnergy();
    __device__ void setEnergy(float energy);
    __device__ void dropEnergy(float energy);

    __device__ Step getStep();
    __device__ void setStep(Step step);

    __device__ int getBodyId();
    __device__ void setBodyId(int body_id);

    __device__ void dispose();
    __device__ bool isDisposed();
    __device__ bool isAlive();

    __device__ void restart();

    __device__ void move();
    __device__ void updateStep(float step_size);
    __device__ void updateStep(float step_size, float remaining_step_size);
    __device__ void updateStep(float step_size, float remaining_step_size, bool hit);
};
```

FIGURA 4.4: *Definición de la clase Particle.*

desechar o consultar si una partícula está siendo utilizada. Todos sus métodos son compilados para ser utilizados en dispositivo (excepto el constructor, el cual es también un método del *host*).

La figura 4.4 muestra la definición de la clase `Particle`. Han sido implementados los métodos para el manejo de posición, dirección y energía. Además ha sido necesario que las partículas contengan una referencia del cuerpo en cual están, para facilitar la búsqueda de ellas en el espacio y evitar divergencias. Finalmente, también se han implementado los mecanismos necesarios para mover la partícula en el espacio, por medio de los métodos `move()` el cual la desplaza según su `step` y el método `updateStepSize()`, el cual actualiza el objeto de `Step`.

#### *4.2.1.3 Implementación de clase Step*

*Step* es un tipo de clase auxiliar utilizada junto con *Particle*. La clase en cuestión permite manejar la información asociada al desplazamiento de una partícula según las reglas definidas para la simulación de bioluminiscencia. A continuación se muestra su definición.

```
class Step {

private:
    float step_size;
    float remaining_step_size;
    bool hit;
    float interaction_ratio;
public:
    __host__ __device__ Step();

    __device__ void setStepSize(float step_size);
    __device__ float getStepSize();

    __device__ void setHit(bool hit);
    __device__ bool getHit();

    __device__ void setInteractionRatio(float ratio);
    __device__ float getInteractionRatio();

    __device__ void setRemainingStepSize(float remaining_step_size);
    __device__ float getRemainingStepSize();
};
```

FIGURA 4.5: *Definición de la clase Step.*

Durante cada iteración de la simulación es necesario calcular el largo de desplazamiento de una partícula. Pero dicho valor por si solo no permite conocer la posición final del fotón. Es necesario además conocer el coeficiente de interacción del medio en el cual se produce el desplazamiento, y más importante aún, cuando la partícula se encuentra en el límite del tejido y ha de ser transmitida, es necesario re-calcular el "resto del largo de desplazamiento". Dicho valor es almacenado, y se expone por medio de los métodos `setRemainingStepSize()` y `getRemainingStepSize()`.

Para determinar la distancia que le resta a una partícula por desplazarse es necesario multiplicar el *step size* original por el *radio de interacción*, el cual está formado por los coeficientes de interacción del medio de incidencia y el medio de transmisión. El radio de interacción es manejado por medio del atributo `interaction_ratio`, el cual resume los dos coeficientes que lo componen.

$$interaction\_ratio = \frac{u_{t_{incidencia}}}{u_{t_{transmision}}} \tag{4.1}$$

Por lo tanto, la clase *Step* facilita manejar el desplazamiento, otorgando un mayor grado de abstracción y modularización al proceso de Step.

#### *4.2.1.4 Implementación de clase Layer*

El presente trabajo utiliza el tipo de geometría *Layer* para representar planos infinitos. Los planos infinitos han sido utilizados por otros invesigadores (Wang & Jacques, 1992) para realizar simulaciones Monte Carlo de bioluminiscencia. En particular dichos trabajos simulan capas de piel humana.

Los cálculos entre partículas y un plano infinito (*Layer*) son significativamente más simples que en un cuerpo como una caja. Consultar si una partícula está en una capa, calcular una interacción o determinar un ángulo de incidencia, se reducen a ecuaciones de menor complejidad.

Considerando entonces la simplicidad y la posibilidad de contrastar resultados, se ha decidido utilizar capas infinitas. Continuando con la misma idea, tampoco se realizará una separación entre una capa y sus propiedades físicas, como los coeficientes de absorción, dispersión o índice refractivo.

#### *4.2.1.5 Implementación de clase Random*

La clase `Random` provee los mecanismos para crear objetos con mútiples máquinas de estado. La idea central es que cada instancia de la clase debe poseer un estado por cada hilo de ejecución CUDA. La figura 4.6 muestra cómo ha sido definida.

```
1  class Random {
2
3  private:
4      curandState* d_states;
5
6  public:
7      __host__ __device__ Random();
8      __host__ Random(int num_states);
9
10     __host__ static Random* getNewDeviceRandom(int num_states);
11
12     __device__ void init(int thread_idx, unsigned long seed);
13     __device__ float getNext(int thread_idx);
14 };
15
16 /*
17  * Kernel definition (__global__ function members are not allowed yet)
18  */
19
20 __global__ void InitRandom(Random* d_random, unsigned long seed);
```

FIGURA 4.6: *Definición de la clase Random.*

Debido a que las máquinas de estado deben estar dentro del dispositivo y que la inicialización es llevada a cabo desde el *host*, es necesario crear el pseudo-constructor `getNewDeviceRandom()` el cual puede ser llamado desde el *host*. Este método retorna un puntero a una máquina de generación de números aleatorios, procurada en memoria global del dispositivo,

Adicionalmente se definen los métodos, `init()` para inicializar las semillas de cada estado y `getNext()` que permite obtener el siguiente número aleatorio desde una máquina de estados.

Otro aspecto importante a considerar es que la inicialización debe ser realizada en paralelo por cada hilo de la tarjeta. Por lo tanto es necesario la creación de un *kernel* para dicho propósito. Debido a que no es posible definir métodos con el calificador `__global__`, se define adicionalmente una función para cumplir dicho cometido.

La generación de números aleatorios es un aspecto fundamental en las simulaciones Monte Carlo. Los generadores de números aleatorios están basados en algoritmos matemáticos que permiten obtener secuencias repetibles. No obstante, dichos números son en realidad pseudo-aleatorios, puesto que luego de cierta cantidad de generaciones es posible observar patrones repetitivos.

Para la generación de los números aleatorios necesarios en este trabajo, se ha utilizado una biblioteca provista por CUDA llamada cuRAND.

La biblioteca cuRAND permite la generación de alto rendimiento de números aleatorios en GPU. Existen dos interfaces en la biblioteca para generar números en lote. La primera permite generación números invocando *kernels* desde el *host* y la segunda a través de funciones específicas desde el dispositivo GPU. Además provee una amplia variedad de algoritmos y ofrece de igual modo un gran rando de distribuciones de probabilidad.

#### *4.2.1.6 Implementación de clase Simulation*

La clase `Simulation` tiene como propósito mantener registro de los datos de una simulación y también organizarla.

```
class Simulation {

public:

    // Matriz de abosorci\'on
    unsigned long long int* d_Arz;

    // Numero de capas del universo
    int num_layers;

    // Capas del universo
    Layer* layers;

    // Nro de fuentes del universo
    int num_sources;

    // Fuente actual
    int current_source_index;

    // Fuentes del universo
    Source* sources;

    // Definici\'on de la Grilla para la matriz de absorci\'on
    float dz;
    float dr;
    int ndz;
    int ndr;

    // Numero de ventos a simular
    int num_events;
    int events_left;

    // Destino de salida para la simulacion
    char* output_file;

    __host__ Simulation();
    __host__ Simulation(unsigned long long int* d_Arz, int num_layers, Layer* layers, int
        num_sources, Source* sources, float dz, float dr, int ndz, int ndr, char* output_file);

    __device__ Layer getLayer(int index);
    __device__ Layer* getLayers();

    __host__ __device__ int getNumLayers();
    __host__ __device__ int getNumSources();
    __host__ __device__ int getCurrentSourceIndex();
    __host__ __device__ float getDZ();
    __host__ __device__ float getDR();
    __host__ __device__ int getNDZ();
    __host__ __device__ int getNDR();
    __host__ __device__ int getNumEvents();
    __host__ __device__ int getEventsLeft();

    __device__ void setLayers(Layer* layers);
    __device__ void setNumLayers(int num_layers);
    __device__ void setNumSources(int num_sources);
    __device__ void setCurrentSourceIndex(int source_index);
    __device__ void setDZ(float dz);
    __device__ void setDR(float dr);
    __device__ void setNDZ(int ndz);
    __device__ void setNDR(int ndr);
    __device__ void setNumEvents(int num_events);
    __device__ void setEventsLeft(int events_left);


    __device__ int subtractEvent();
    __device__ void recordAbsorption(Particle p, float weight_dropped);
    __host__ void updateSource(int source_index);
    __device__ Source getCurrentSource();
};
```

FIGURA 4.7: *Definición de la clase Simulation.*

La definición de la clase en la figura 4.7 muestra los métodos y atributos necesarios para llevar a cabo una simulación de bioluminiscencia.

A través del arreglo de capas y el arreglo de fuentes es posible definir el universo de simulación que se utilizará. Las mediciones que se realicen serán almacenadas en la matriz de absorción `d_Azr` la cual es un puntero a memoria global del dispositivo. En este mismo contexto, es el método `recordAbsorption()` el encargado de hacer el registro de la absorción, para ello es necesario que acceda de manera atómica a la posición que desea modificar.

La discretización del universo está dada por los atributos `dz` y `dr` los cuales indican la distancia de separación entre cada posición de la grilla. Los atributos `ndz` y `ndr` indican la cantidad de posiciones que la grilla tendrá en el eje $X$ y en el eje $R$, respectivamente.

Finalmente, la clase provee los métodos necesarios para el manejo adecuado de las fuentes y la cantidad de eventos restantes de la simulación.

#### *4.2.1.7 Input/Output*

Se han implementado las clases adecuadas para la lectura de experimentos y escritura de resultados.

La definición de experimentos ha sido tomada desde el trabajo realizado por Wang. La clase *InputHandler* permite leer experimentos que definen:

1. Número de simulaciones
2. Nombre de archivo y tipo de salida
3. Cantidad de eventos a simular
4. Dimensión de la grilla que discretiza el universo
5. Número de capas infinitas
6. Tipo y propiedades físicas de las capas

La clase `OutputHandler`, permite escribir resultados de distintos tipos de simulación. Para bioluminiscencia se ha implementado el método `writeBiolumSimulation()`, el cual registra en un archivo la matriz de absorción obtenida.

### 4.2.2 Implementación de procesos

Los procesos físicos son la segunda componente de la arquitectura propuesta. A través de éstos se implementará el paralelismo y de dicha manera se pretende obtener una mejora en el rendimiento computacional.

El modelo de objetos concurrentes define las clases `ProcessHandler` y `Process`, las cuales están destinadas a abstraer el conjunto de procesos físicos que son aplicados a una partícula y el proceso en cuestión, respectivamente. Este enfoque no ha sido reutilizado, puesto que en realidad no es posible calcular a priori todos los procesos que afectarán a una partícula. La principal razón de esto es que existen procesos que dependen del resultado de un proceso anterior.

En consecuencia se ha tomado la decisión de descartar la utilización de un `ProcessHandler` en medro de la implementación de un proceso de control de la simulación el cual será detallado en la Sección 4.3.

A continuación, se describen los procesos físicos que han sido implementados por medio de *kernels*.

#### *4.2.2.1 Proceso `ComputeStepSize()`*

El proceso `ComputeStepSize()` es el encargado de realizar el cálculo en paralelo del largo de desplazamiento del *pool* de partículas.

**Algoritmo 4.1:** Cálculo del nuevo largo de desplazamiento.

```
1  int i = threadIdx.x + blockIdx.x * blockDim.x;
2  Particle& p = particles[i];
3  if p.getStep().getRemainingStepSize() == ZERO then
4  |   float random_num = random();
5  |   float step_size = -log(random_num) / simulation.getLayer(p).getInteractionCoef();
6  |   p.updateStep(new_step_size);
7  else
8  |   float new_step_size = p.getStep().getRemainingStepSize() /
   |   simulation.getLayer(p).getInteractionCoef();
9  |   p.updateStep(new_step_size);
10 end
```

El algoritmo 4.1 muestra los dos casos fundamentales que pueden suceder. El primero es cuando la partícula no posee desplazamiento faltante del *step* anterior. En dicha situación se calcula el nuevo *step size* muestreando un número aleatorio y obteniendo el coeficiente de interacción del cuerpo actual. En caso de que exista una porción de step que no ha sido utilizada para desplazar a la partícula, se actualiza el *step* con dicha cantidad.

#### *4.2.2.2 Proceso* `OnHitBoundary()`

El proceso `OnHitBoundary()` es el encargado de cambiar el estado de *hit* del step de una partícula.

**Algoritmo 4.2:** Proceso de Hit, el cual se lleva a cabo cuando una partícula llega al límite de un cuerpo.

```
1  int i = threadIdx.x + blockIdx.x * blockDim.x;
2  Particle& p = particles[i];
3  float uz = p.getDirection().getZ();
4  float z_bound = (uz > ZERO)? simulation.getLayer(p).getZ1() : simulation.getLayer(p).getZ0();
5  float diff = (z_bound - p.getPosition().getZ()) / uz;
6  bool hit_boundary = (uz != ZERO) && (p.getStep().getStepSize() > diff);
7  if hit_boundary then
8  |   float remaining_step = (p.getStep().getStepSize() - diff) *
   |   simulation.getLayer(p).getInteractionCoef();
9  |   p.updateStep(diff, remaining_step);
10 end
```

El algoritmo presentado evalúa si la posición final dada por el largo de desplazamiento calculado en el proceso anterior está fuera de la capa actual. En dicho caso se desplaza la partícula

hasta el límite del tejido y se actualiza la variable que indica cuánto queda por desplazarse. A través de este proceso, el fotón es modificado y en los posteriores procesos físicos se podrá conocer si la partícula ha hecho colisión con el límite o no.

#### *4.2.2.3 Proceso* `Move()`

Es el más simple de todos y corresponde al proceso que mueve una partícula en su dirección de desplazamiento.

Para mover una partícula es necesario sumar a su posición actual el largo del desplazamiento multiplicado por la dirección.

**Algoritmo 4.3:** Proceso Move, utilizado para desplazar la partícula a una nueva posición.

```
1 int i = threadIdx.x + blockIdx.x * blockDim.x;
2 Particle& p = particles[i];
3 Point position = p.getPosition();
4 Point direction = p.getDirection();
5 float step_size = p.step.getStepSize();
6 float x = position.getX() + (direction.getX() * step_size);
7 float y = position.getY() + (direction.getY() * step_size);
8 float z = position.getZ() + (direction.getZ() * step_size);
9 p.updatePosition(x, y, z);
```

El algoritmo 4.3 muestra cómo las componentes de la posición son actualizadas añadiendo los valores calculados a través de la magnitud y dirección de desplazamiento.

#### *4.2.2.4 Proceso* `ReflectTransmit()`

El proceso físico `ReflectTransmit()` es aquel que posee mayor grado de divergencia. A través de este proceso se determina si una partícula que ha llegado al límite de un tejido se refleja o transmite.

El proceso sigue las reglas definidas en el Capítulo 3. Cuando una partícula es reflejada sólo es necesario invertir la componente $z$ del atributo de dirección. Cuando la partícula es transmitida es necesario calcular la nueva dirección mediante el cociente de los índices refractivos.

El algoritmo 4.4 describe los pasos para llevar a cabo el proceso. En primer lugar se

analiza la dirección de la partícula para determinar si se encuentra en el límite superior o inferior de la capa. Posteriormente se asume que la partícula es reflejada. En caso que el ángulo de incidencia sea mayor al ángulo crítico de reflexión, se procede a determinar si el fotón es transmitido o se finaliza el proceso. El fotón es transmitido cuando la reflectancia interna es menor que una variable aleatoria muestreada.

**Algoritmo 4.4:** Proceso ReflectTransmit, utilizado para realizar reflexión o transmisión de una partícula que ha llegado al límite de un cuerpo.

```
1  int i = threadIdx.x + blockIdx.x * blockDim.x;
2  Particle& p = particles[i];
3  float dir_x;
4  float dir_y;
5  float dir_z;
6  float cos_crit;
7  int next_layer;
8  if p.getDirection().getZ() > ZERO then
9  |   cos_crit = simulation.getLayer(p).getCosCrit1();
10 |   next_layer = p.getBodyId() + 1;
11 else
12 |   cos_crit = simulation.getLayer(p).getCosCrit0();
13 |   next_layer = p.getBodyId() - 1;
14 end
15 float ca1 = abs(p.getDirection().getZ());
16 dir_x = p.getDirection().getX();
17 dir_y = p.getDirection().getY();
18 dir_z = -1 * p.getDirection().getZ();
19 p.setDirection(dir_x, dir_y, dir_z);
20 if ca1 > cos_crit then
21 |   float ni = simulation.getLayer(p).getRefractiveIndex();
22 |   float nt = simulation.getLayer(next_layer).getRefractiveIndex();
23 |   float ni_nt = ni / nt;
24 |   float sa1 = sqrt(ONE - ca1 * ca1);
25 |   float sa2 = min(ni_nt * sa1, ONE);
26 |   float uz1 = sqrt(ONE - sa2 * sa2);
27 |   float ca1ca2 = ca1 * uz1;
28 |   float sa1sa2 = sa1 * sa2;
29 |   float sa1ca2 = sa1 * uz1;
30 |   float ca1sa2 = ca1 * sa2;
31 |   float rFresnel = fresnel(ca1, ca2, sa1, sa2);
32 |   float rand = random();
33 |   if rFresnel < rand then
34 |   |   p.setBodyId(next_layer);
35 |   |   dir_x = p.getDirection().getX() * ni_nt;
36 |   |   dir_y = p.getDirection().getY() * ni_nt;
37 |   |   dir_z = -copysignf(uz1, p.getDirection().getZ());
38 |   |   p.setDirection(dir_x, dir_y, dir_z);
39 |   |   if p.getBodyId() == 0 —— p.getBodyId() == simulation.getNumLayers() - 1 then
40 |   |   |   p.setEnergy(ZERO);
41 |   |   end
42 |   end
43 end
```

#### *4.2.2.5 Proceso* `Absorption()`

El proceso de absorción es el encargado de disminuir la energía de la partícula. El algoritmo 4.5 muestra cómo es obtenido el peso a disminuir. De manera adicional, el objeto de la clase `Simulation`, realiza un registro de la energía disminuida, agregándola a la matriz de absorción. Dicho procedimiento se lleva a cabo por el método `recordAbsorption()`.

**Algoritmo 4.5:** Proceso Absorption, el cual disminuye la energía de una partícula.

```
1 int i = threadIdx.x + blockIdx.x * blockDim.x;
2 Particle& p = particles[i];
3 float energy = p.getEnergy();
4 float abs_coef = simulation.getLayer(p).getAbsorptionCoef();
5 float int_coef = simulation.getLayer(p).getInteractionCoef();
6 float weight_dropped = energy * abs_coef / int_coef;
7 p.dropEnergy(weight_dropped);
8 simulation.recordAbsorption(p, weight_dropped);
```

#### *4.2.2.6 Proceso* `Scattering()`

El proceso de dispersión involucra el muestreo de dos variables: el ángulo de deflección y el ángulo azimutal. El algoritmo 4.6 muestra la manera en que ha sido implementado el muestreo de los ángulos para finalmente actualizar la dirección del fotón. Es necesario además acceder a las propiedades del material, puesto que el ángulo de deflección depende del factor de anisotropía.

**Algoritmo 4.6:** Proceso Scattering, utilizado para simular dispersión en las partículas.

```
1  int i = threadIdx.x + blockIdx.x * blockDim.x;
2  Particle& p = particles[i];
3  // cos theta, sin theta
4  float cost, sint;
5  // cos psi, sin psi
6  float cosp, sinp;
7  float psi;
8  float SIGN;
9  float temp;
10 float last_ux, last_uy, last_uz;
11 float rand;
12 float g = sim.getLayer(p).getAnisotropyCoef();
13 rand = random();
14 cost = TWO * rand - ONE;
15 if g > ZERO then
16 |  temp = (ONE - g * g) / (ONE + g*cost);
17 |  cost = (ONE + g * g - temp*temp) / (TWO * g);
18 |  cost = max(cost, -ONE);
19 |  cost = min(cost, ONE);
20 end
21 sint = sqrt(ONE - cost * cost);
22 rand = random();
23 psi = TWO * PI * rand;
24 sincos(psi, &sinp, &cosp);
25 float stcp = sint * cosp;
26 float stsp = sint * sinp;
27 last_ux = p.getDirection().getX();
28 last_uy = p.getDirection().getY();
29 last_uz = p.getDirection().getZ();
30 if fabsf(last_uz) > cos(ZERO) then
31 |  SIGN = ((last_uz) >= ZERO ? ONE : -ONE);
32 |  p.setDirection(stcp, stsp, cost * SIGN);
33 else
34 |  temp = 1 / sqrt(ONE - last_uz * last_uz);
35 |  float new_ux = (stcp * last_ux * last_uz - stsp * last_uy) * temp + last_ux * cost;
36 |  float new_uy = (stcp * last_uy * last_uz + stsp * last_ux) * temp + last_uy * cost;
37 |  float new_uz = (-stcp / temp) + last_uz * cost;
38 |  p.setDirection(new_ux, new_uy, new_uz);
39 end
```

#### *4.2.2.7 Proceso* `Roulette()`

La finalización de la simulación de una partícula se lleva a cabo a través del proceso de ruleta. El *kernel* sólo tiene efecto sobre un fotón cuando su peso es menor a la constante de *peso crítico*, el cual está definido como $10^{-4}$. El proceso compara un número aleatorio con la oportunidad que tiene la partícula de sobrevivir. En caso de hacerlo, se aumenta la energía a través de la división del valor por la probabilidad (generalmente 0.1). En caso de que la partícula no sobreviva y la cantidad de eventos que quede por simular sea mayor a la cantidad de hilos que utiliza el *kernel*, se realiza la sustracción de un evento desde la simulación y se reinicia el fotón. En caso de que la cantidad de hilos del *kernel* sea mayor a los eventos restantes, el hilo es inhabilitado desechando la partícula totalmente.

**Algoritmo 4.7:** Proceso Roulette, en el cual se decide si se finaliza la simulación de una partícula.

```
1  int i = threadIdx.x + blockIdx.x * blockDim.x;
2  Particle& p = particles[i];
3  if p.getEnergy() < CRITICAL_WEIGHT then
4  |   float rand = random();
5  |   if p.getEnergy() != ZERO && rand < CHANCE then
6  |   |   p.setEnergy(p.getEnergy() * ONE / CHANCE);
7  |   else if simulation.subtractEvent() >= NUM_THREADS then
8  |   |   Source source = simulation.getCurrentSource();
9  |   |   Point position = source.getPosition();
10 |   |   Point direction = source.getRandomDirection();
11 |   |   p.restart(position, direction);
12 |   else
13 |   |   p.dispose();
14 |   end
15 end
```

## 4.3 MODELO DE CONTROL DE LA SIMULACIÓN

En el modelo propuesto, el control de la simulación permanece totalmente en el *host*. Esto se logra escribiendo la simulación como una serie de pasos o procesos, que son aplicados a la totalidad de las partículas hasta que finaliza la simulación de cada una.

El algoritmo 4.8 muestra la manera en que ha sido implementada la simulación de

bioluminiscencia. Se ha secuencializado la simulación de cada fuente. Es importante notar que dicha decisión ha sido tomada desde el host y no en la GPU. Cada uno de los procesos físicos es aplicado a las partículas, hasta agotar todas los eventos de la fuente

**Algoritmo 4.8:** Control de la simulación.

```
1  foreach source in sources do
2      simulation.setSource(source);
3      int events_left = source.getEventsLeft();
4      Particle * d_particles;
5      KernelParticlesConstructor(d_particles, d_simulation, d_random);
6      while events_left ≥ 1 do
7          KernelComputeStepSizes(d_particles, d_simulation, d_random);
8          KernelHitBoundary(d_particles, d_simulation, d_random);
9          KernelMove(d_particles, d_simulation, d_random);
10         KernelReflectTransmit(d_particles, d_simulation, d_random);
11         KernelAbsorption(d_particles, d_simulation, d_random);
12         KernelScattering(d_particles, d_simulation, d_random);
13         KernelRoulette(d_particles, d_simulation, d_random);
14         cudaMemcpy(&events_left, &d_simulations[i].events_left, sizeof(int),
           cudaMemcpyDeviceToHost);
15     end
16 end
```

El modelo propuesto resulta útil cuando es necesario tomar decisiones que provocan divergencia. Aunque la mayor parte de las decisiones pueden ser tomadas en el *host* es posible notar que para esta implementación en particular existe la posibilidad de unificar los *kernels* y analizar el costo de la latencia involucrada en llamar a cada uno. Esta tarea será llevada a cabo y analizada en el Capítulo 5.

# CAPÍTULO 5. EXPERIMENTOS

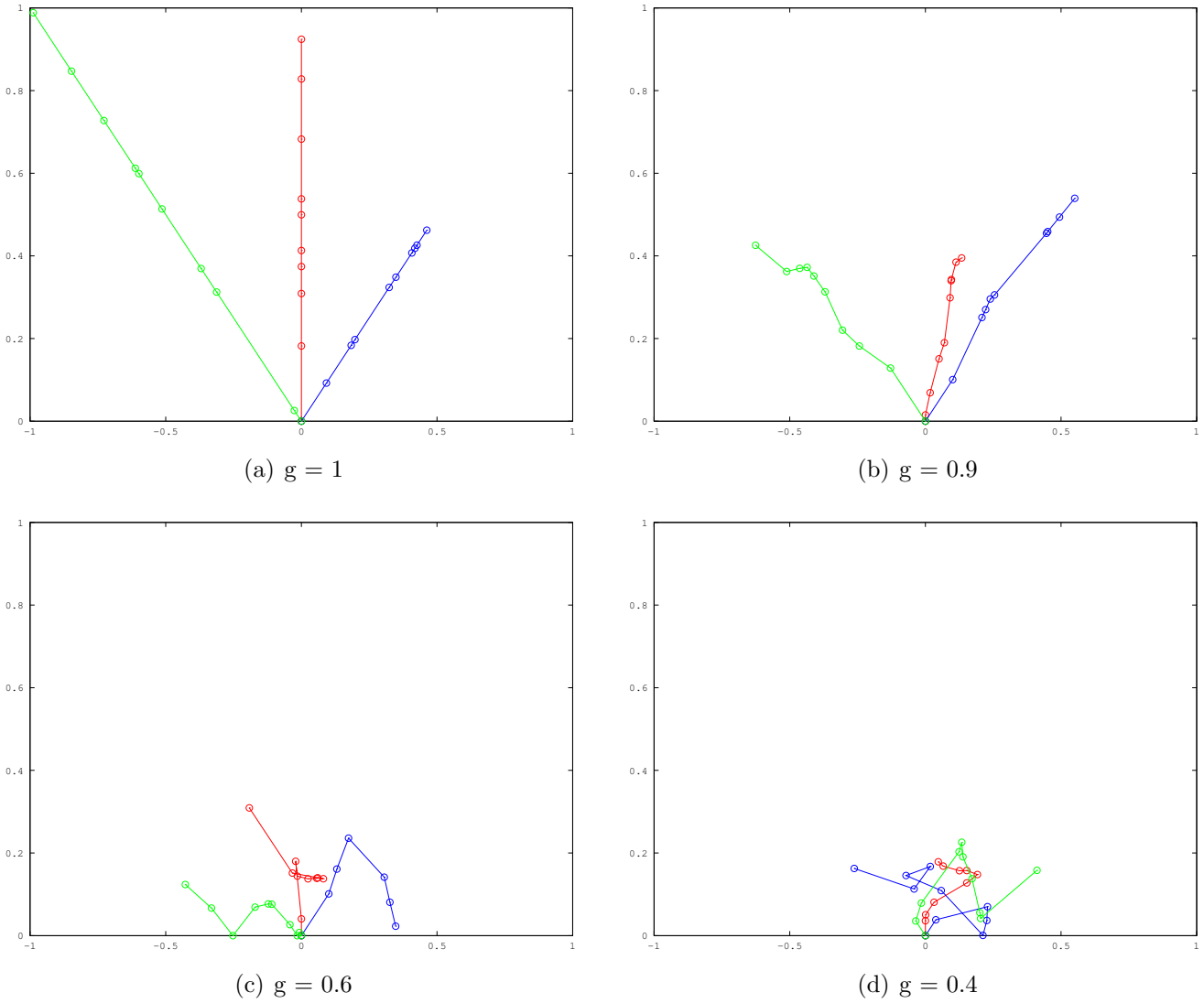
A continuación se detallan los experimentos realizados sobre la implementación para bioluminiscencia. Las pruebas hechas comprenden la funcionalidad del sistema por medio de pruebas unitarias, la validación por medio de la comparación con resultados de otros estudios y finalmente el rendimiento computacional.

Durante el Capítulo 4, fue descrito el modelo de control de la simulación, el cual desglosa el fenómeno en una lista de procesos físicos aplicados a un lote de partículas. En dicho capítulo además se propuso realizar la implementación por medio de un *kernel* único, con el objetivo de minimizar la latencia involucrada en el llamado a código de GPU. Dicha implementación obtuvo los mejores resultados y en las tablas de rendimiento se hace referencia a ella como una *versión optimizada*.

## 5.1 FUNCIONALIDADES

### 5.1.1 Proceso de dispersión y factor de anisotropía

El proceso de dispersión de una partícula está directamente relacionado con el factor de anisotropía de la capa donde es transportada. Como fue descrito en el Capítulo 3, el coseno del ángulo de deflección está dado principalmente por el factor de anisotropía y una variable aleatoria. El rango en que el ángulo de deflección puede variar aumentará en la medida que $g$ disminuya, es decir, el movimiento del fotón tenderá ser más isotrópico.



(a) g = 1 (b) g = 0.9

(c) g = 0.6 (d) g = 0.4

FIGURA 5.1: *Dispersión de fotones para factores de anisotropía 1, 0.9, 0.6 y 0.4.*

La figura anterior muestra un experimento realizado, en el cual para un medio en particular se ha disminuido el factor de anisotropía gradualmente. Se han utilizado tres fuentes de partículas ubicadas en la posición $(0, 0, 0)$. La fuente Nº 1 en color rojo, ha sido inicializada para emitir partículas en dirección $(0, 0, 1)$, la fuente Nº 2 en color azul emite partículas en la dirección $(0, 1, 1)$ y por último la fuente Nº 3 emite partículas en dirección $(0, -1, 1)$. Cada fuente de partículas emite sólo un fotón y el plano de visualización es perpendicular al eje $X$.

En la figura 5.1(a) es posible notar el movimiento lineal de las partículas. Cuando $g = 1$ el coseno del ángulo de deflección toma valor 1, por lo tanto el ángulo de deflección es 0 y la partícula no sufre un cambio en su trayectoria.

La figura 5.1(b), el cual posee $g = 0.9$, muestra un movimiento no lineal. Sin embargo

hay una clara tendencia en la dirección de la partícula. Al disminuir el factor de anisostropía ha aumentado el rango en que el ángulo de deflección puede variar. En los siguientes gráficos 5.1(c) y 5.1(d) esto es más claro, el movimiento no parece seguir una tendencia establecida

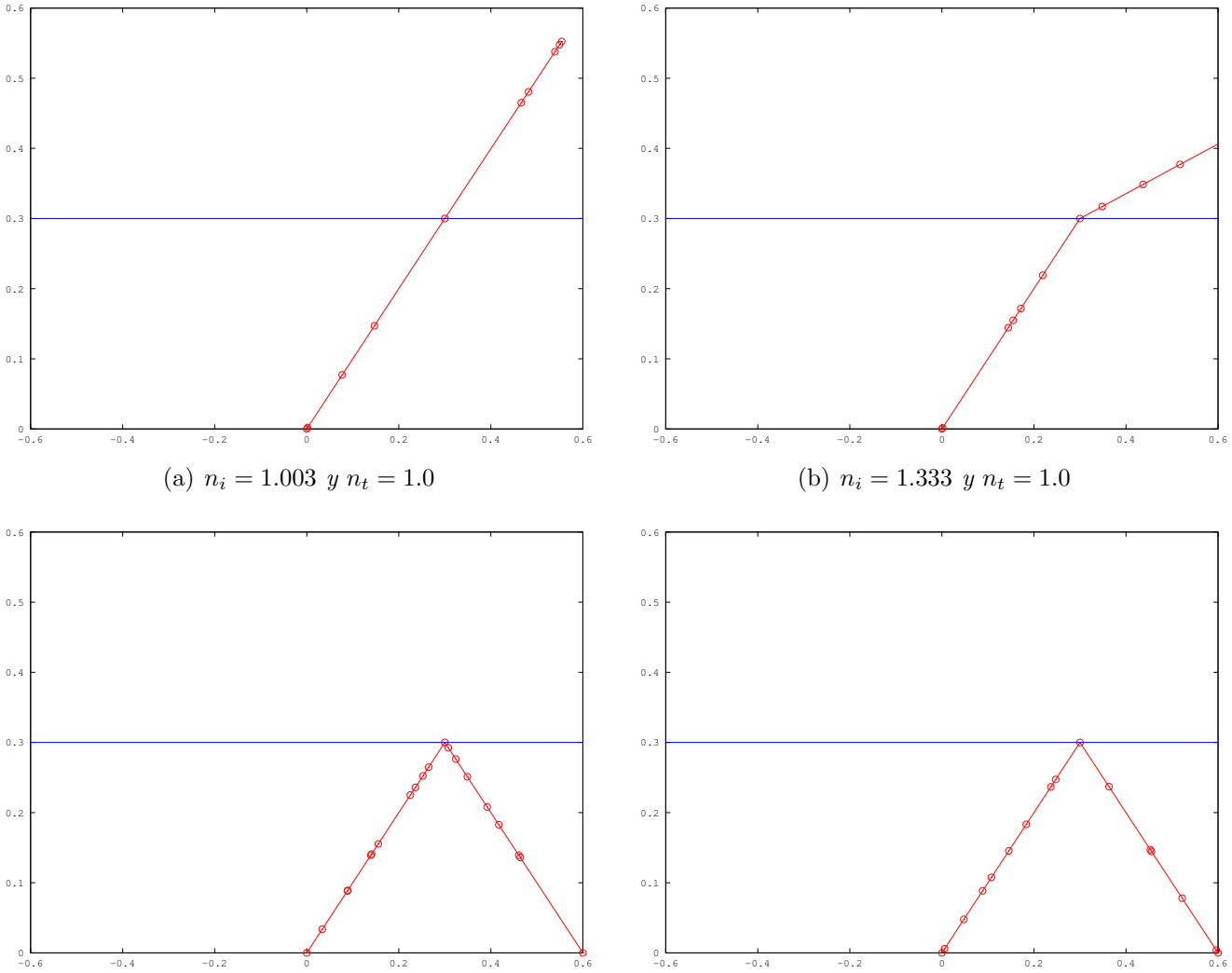
### 5.1.2 Reflexión y refracción

Cuando una partícula llega al límite del cuerpo que la alberga, ésta puede ser reflectada o transmitida hacia un nuevo medio. La probabilidad de que ocurra una u otra alternativa está condicionada al índice refractivo del medio de incidencia $n_i$ y del medio de transmisión $n_t$.

A medida que el índice refractivo del medio de incidencia es más alto la probabilidad de que la partícula sea reflectada aumenta. A continuación, se presenta un experimento para el cual se ha incrementado gradualmente el índice refractivo $n_i$ del medio de incidencia, manteniendo constante el índice refractivo $n_t$ del medio de transmisión, el cual corresponde a vacío. Se ha utilizado sólo una fuente de luz en la posición $(0, 0, 0)$, la cual emite una partícula en dirección $(0, 1, 1)$ en cada uno de los casos.

La figura 5.2(a) muestra un índice refractivo en el medio de incidencia ligeramente superior a vacío: aire. Dado la poca diferencia entre los índices se obtiene una refracción despreciable. Sin embargo, la figura 5.2(b) muestra una refracción pronunciada, dado que el medio de incidencia posee un índice refractivo mayor, similar al de agua.

Las figuras 5.2(c) y 5.2(d) muestran el resultado de utilizar índices refractivos altos, similares al de cuarzo y diamante, respectivamente. En ambos gráficos se ha obtenido una reflexión de las partículas, lo cual resulta predecible debido al alto valor de sus índices. Por último, es posible apreciar también que el ángulo de incidencia es exactamente igual al ángulo de reflexión.



(a) $n_i = 1.003$ *y* $n_t = 1.0$

(b) $n_i = 1.333$ *y* $n_t = 1.0$

(c) $n_i = 1.544$ *y* $n_t = 1.0$

(d) $n_i = 2.42$ *y* $n_t = 1.0$

FIGURA 5.2: *Reflexión y refracción al aumentar índice refractivo del medio de incidencia.*

### 5.1.3 Coeficiente de absorción, coeficiente de dispersión y Step

El largo de desplazamiento de un fotón es inversamente proporcional al coeficiente de interacción $\mu_t$ del cuerpo en el que se encuentra la partícula. Como fue descrito anteriormente, $\mu_t$ es la suma del coeficiente de absorción $\mu_a$ y el coeficiente de dispersión $\mu_s$. En consecuencia, se espera que para cuerpos con mayores coeficientes de interacción el desplazamiento de las partículas sea menor.

La figuras listadas a continuación muestran un experimento en el cual se ha aumentado gradualmente el coeficiente de interacción. Los valores son los descritos en la tabla 5.1 la cual ha sido

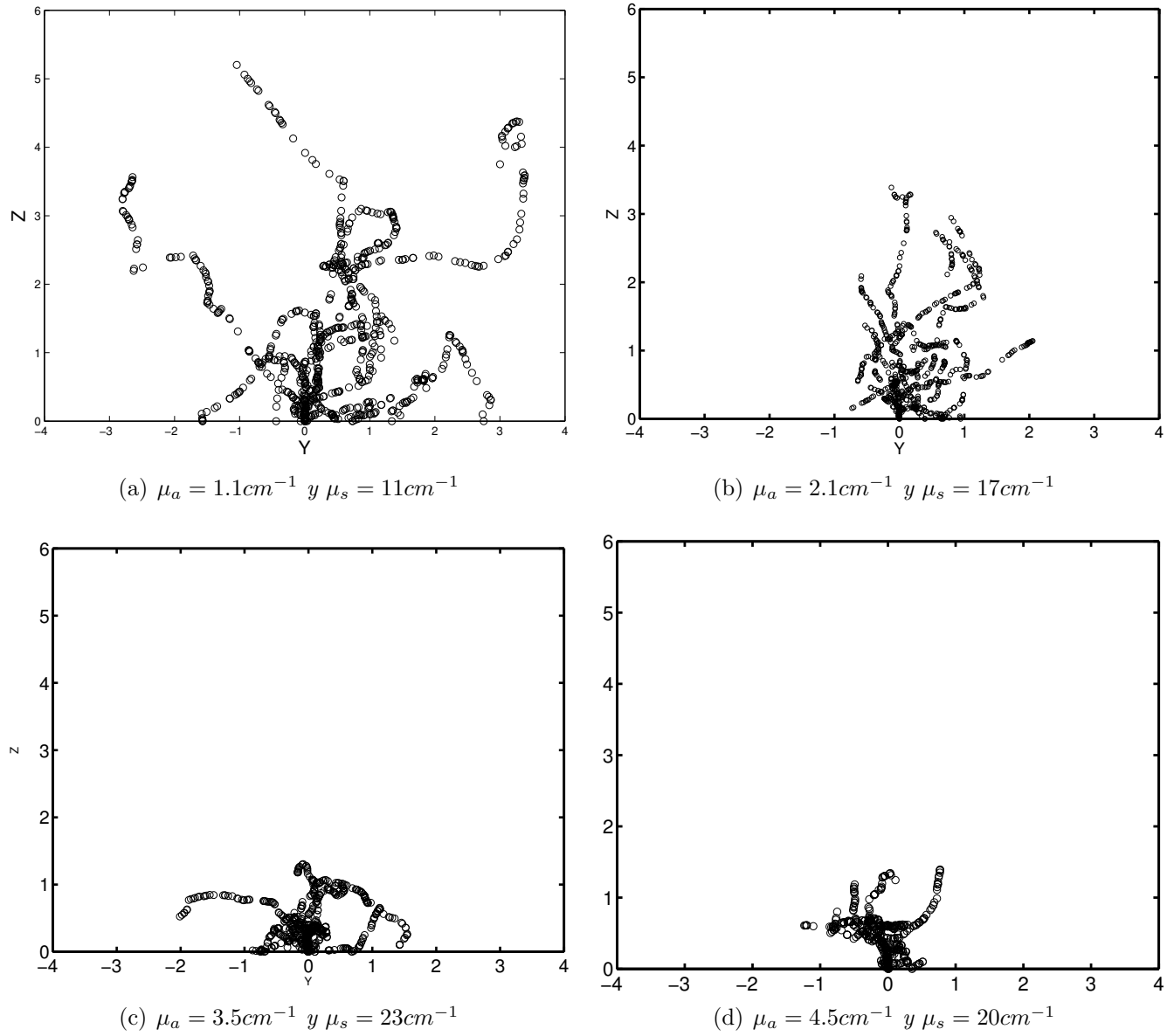
obtenida desde (Wang et al., 2006). En cada caso se ha utilizado una fuente en la posición $(0, 0, 0)$ que emite diez partículas en dirección $(0, 0, 1)$.

TABLA 5.1: Coeficientes de absorción y dispersión para diferentes órganos de un cuerpo.

| Material | $\mu_a[cm^{-1}]$ | $\mu_s[cm^{-1}]$ |
|---|---|---|
| Corazón | 1.1 | 11 |
| Estómago | 2.1 | 17 |
| Pulmón | 3.5 | 23 |
| Hígado | 4.5 | 20 |

La figura 5.3(a) muestra el comportamiento de las partículas en un medio con coeficientes de un corazón. Se puede apreciar que los fotones realizan extensos steps antes de que su simulación sea finalizada. La figura 5.3(b) correspondiente a un estómago, muestra partículas con menor desplazamiento, siendo éstas finalizadas mucho antes que en la figura anterior.

En las figuras 5.3(c) y 5.3(d) las partículas permanecen cerca de la fuente y su desplazamiento es notoriamente bajo. Esto es debido a los altos coeficientes de interacción que poseen un pulmón e hígado, respectivamente.




(a) $\mu_a = 1.1cm^{-1}$ *y* $\mu_s = 11cm^{-1}$

(b) $\mu_a = 2.1cm^{-1}$ *y* $\mu_s = 17cm^{-1}$

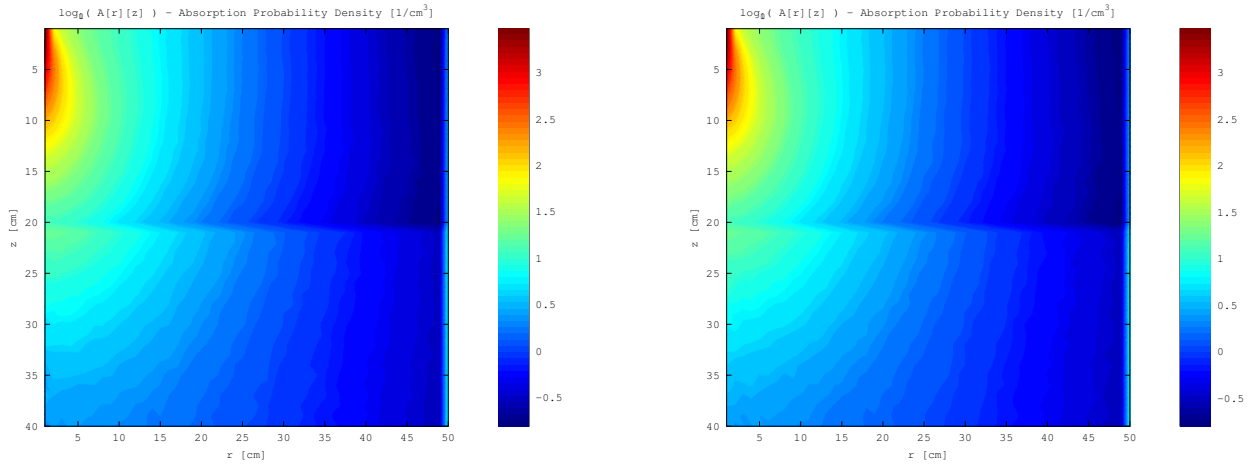
(c) $\mu_a = 3.5cm^{-1}$ *y* $\mu_s = 23cm^{-1}$

(d) $\mu_a = 4.5cm^{-1}$ *y* $\mu_s = 20cm^{-1}$

FIGURA 5.3: *Decaímiento de steps al aumentar coeficiente de interacción.*

### 5.1.4 Absorción

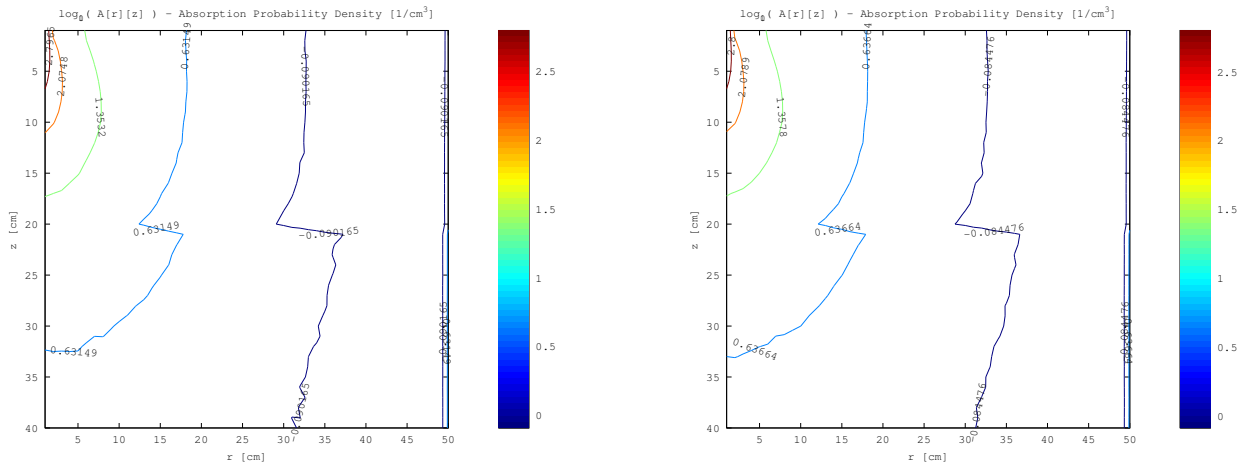
Uno de los principales resultados de una simulación de bioluminiscencia es la matriz de absorción de energía. A medida que las partículas se desplazan por los tejidos parte de su energía es absorbida por el medio, tal como ha sido descrito en el Capítulo 3. La absorción es registrada en un arreglo biodimensional llamado $A[r][z]$, el cual almacena la densidad de probabilidad de la absorción en función del radio $r$ y la profundidad $z$ para el haz simulado.




(a) GPU (b) CPU

FIGURA 5.4: *Densidad de probabilidad de la absorción obtenida para implementación GPU y CPU en escala logarítmica.*

Las figuras 5.4(a) y 5.4(b) muestran una comparación de los resultados obtenidos por la implementación para GPU en bioluminiscencia y de una implementación de en CPU secuencial. En ambos casos se ha utilizado el mismo experimento, el cual consta de tres capas y diez millones de fotones.




(a) GPU (b) CPU

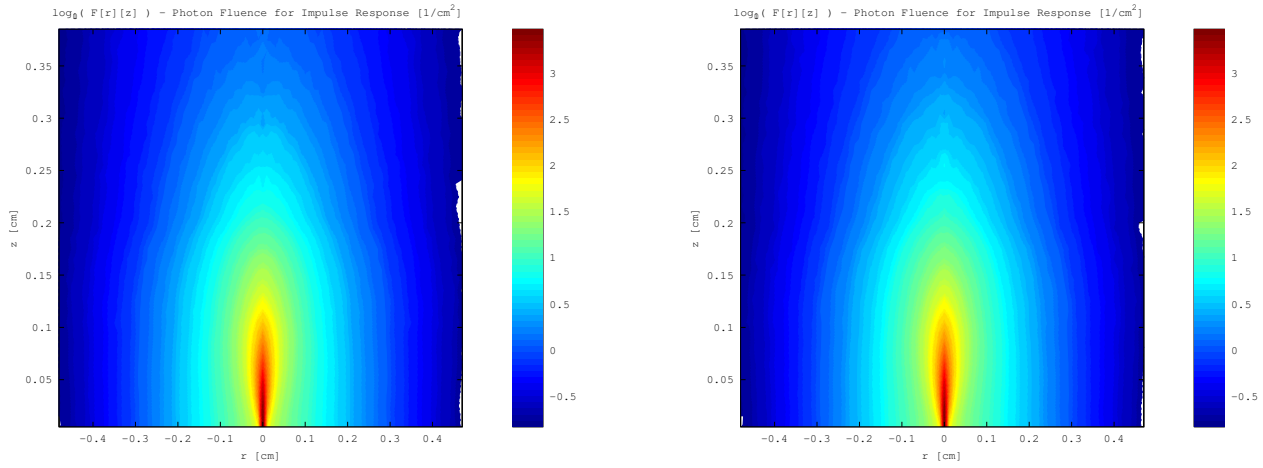
FIGURA 5.5: *Contornos de densidad de probabilidad de la absorción obtenida para implementación GPU y CPU en escala logarítmica.*

En las figuras 5.5(a) y 5.5(b) es posible apreciar las leves diferencias en los valores de las matrices. Cabe destacar que en la implementación GPU el nivel de precisión es menor debido a que

la tarjeta utilizada no permite el uso del tipo de dato *double*. No obstante, queda en evidencia la similitud de los contornos de ambas imágenes.

### 5.1.5 Fluencia

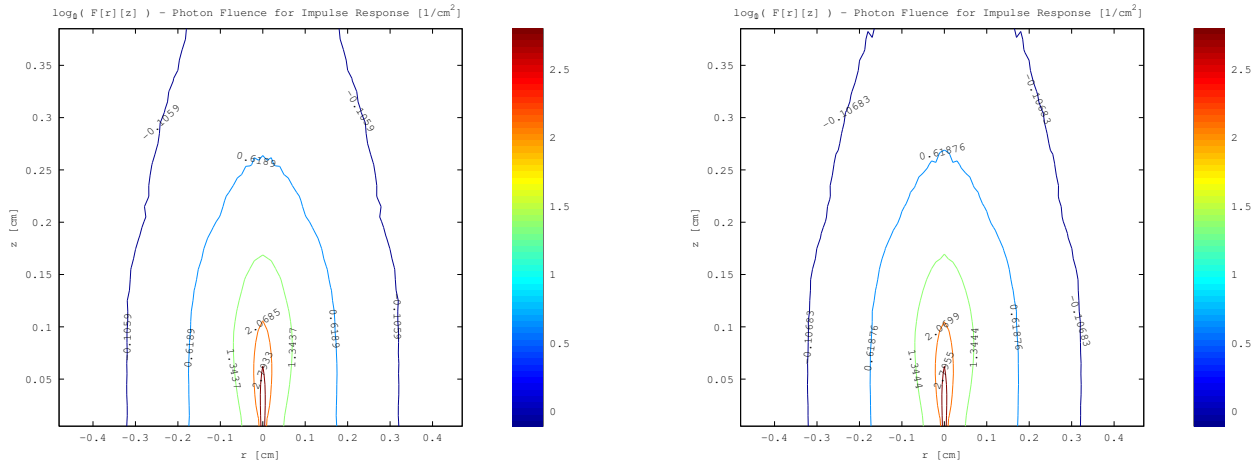
La matriz de absorción generada por medio de la simulación puede ser convertida en fluencia y viceversa. La fluencia de una posición de la grilla corresponde al valor de la energía acumulada dividido por el coeficiente local de absorción de la capa. A continuación las figuras 5.6(a) y 5.6(b) muestran la comparación de la fluencia obtenida para la implementación en GPU y una implementación CPU secuencial.




(a) GPU (b) CPU

FIGURA 5.6: *Comparación de fluencia obtenida en implementación GPU y CPU*

Al igual que en el experimento anterior, las figuras 5.7(a) y 5.7(b) muestran diferencias leves en los valores de los contornos, obteniendo ambas implementaciones resultados similares para las mediciones aplicadas.




(a) GPU (b) CPU

FIGURA 5.7: *Contornos de fluencia obtenida en implementación GPU y CPU*

## 5.2 RENDIMIENTO COMPUTACIONAL

Uno de los propósitos fundamentales del presente trabajo es acelerar los tiempos de simulaciones Monte Carlo. Por ello es necesario analizar el rendimiento computacional de la solución obtenida.

A continuación se detallan los experimentos para analizar rendimiento computacional.

### 5.2.1 Cantidad de eventos

Dependiendo del tipo de simulación, una fuente de luz podría emitir cientos de millones de eventos. Por lo tanto, resulta interesante analizar el comportamiento de los resultados aplicando variaciones en la cantidad de eventos involucrados en la simulación para conocer de qué manera su magnitud influye en el tiempo de ejecución.

En consecuencia se han realizado dos tipos de experimentos relacionados a cantidad de eventos. Las tablas 5.2 y 5.3 muestran los resultados obtenidos al aumentar gradualmente de manera lineal la cantidad de eventos de una simulación, manteniendo todos los demás parámetros constantes.

TABLA 5.2: Análisis de tiempo de ejecución y speedup para una variación lineal en la cantidad de eventos.

| Nº Photons | GPU (s) | CPU (s) | Speedup |
|---|---|---|---|
| 10,000,000 | 130.75 | 1222.4 | 9x |
| 20,000,000 | 260.07 | 2462.6 | 9x |
| 30,000,000 | 390.20 | 3697.2 | 9x |
| 40,000,000 | 519.60 | 4911.6 | 9x |
| 50,000,000 | 651.70 | 6132.0 | 9x |

TABLA 5.3: Análisis de tiempo de ejecución y speedup para una variación lineal en la cantidad de eventos utilizando la versión optimizada del software.

| Nº Photons | GPU Opt (s) | CPU (s) | Speedup |
|---|---|---|---|
| 10,000,000 | 36.82 | 1222.4 | 33x |
| 20,000,000 | 73.24 | 2462.6 | 34x |
| 30,000,000 | 109.75 | 3697.2 | 34x |
| 40,000,000 | 146.17 | 4911.6 | 34x |
| 50,000,000 | 182.6 | 6132.0 | 34x |

Los tiempos de cada una de las implementaciones, CPU, GPU normal y GPU optimizada aumentan en la misma proporción que lo hacen los eventos a simular, es decir, de manera lineal.

Por otro lado, la tablas 5.4 y 5.5 muestran los resultados obtenidos al aumentar gradualmente de manera exponencial la cantidad de eventos de una simulación, manteniendo todos los demás parámetros constantes.

TABLA 5.4: Análisis de tiempo de ejecución y speedup al aumentar de manera exponencial la cantidad de eventos de la simulación.

| Nº Photons | GPU (s) | CPU (s) | Speedup |
|---|---|---|---|
| 10,000 | 0.22 | 1.30 | 6x |
| 100,000 | 2.27 | 12.22 | 5x |
| 1,000,000 | 13.88 | 124.81 | 9x |
| 10,000,000 | 130.75 | 1222.40 | 9x |
| 100,000,000 | 1297.7 | 12002.30 | 9x |

TABLA 5.5: Análisis de tiempo de ejecución y speedup al aumentar de manera exponencial la cantidad de eventos de la simulación para la versión optimizada del software.

| Nº Photons | GPU Opt (s) | CPU (s) | Speedup |
|---|---|---|---|
| 10,000 | 0.50 | 1.21 | 2x |
| 100,000 | 0.74 | 11.98 | 16x |
| 1,000,000 | 3.98 | 127.41 | 32x |
| 10,000,000 | 36.82 | 1222.40 | 33x |
| 100,000,000 | 365.04 | 12002.30 | 33x |

Al igual que el experimento anterior, los tiempos de ejecución aumentan del mismo modo que la cantidad de eventos. Un aumento exponencial en la cantidad de fotones ha tenido como consecuencia un aumento exponencial en los tiempos de simulación.

### 5.2.2 Discretización del universo

Para realizar el registro de la matriz de absorción se requiere discretizar el universo en una grilla. El número de posiciones y las dimensiones de cada una pueden variar dependiendo del experimento.

Para probar el rendimiento, se ha utilizado el mismo experimento pero en este caso se han variado los parámetros $dr$ y $dz$ los cuales representan la medida de cada posición y también los parámetros $ndr$ y $ndz$, los que hacen referencia a la cantidad de posiciones en el eje $r$ y el eje $z$ respectivamente.

La tabla 5.6 no muestra grandes diferencias en las distintas configuraciones de universo. No obstante, la tabla 5.7 la cual corresponde a la versión optimizada del software deja en evidencia que cuando el universo es de mayor tamaño el *speedup* alcanzado es más elevado. La principal razón de esto es debido a que cuando existen más posiciones en el universo disminuye la posibilidad de que dos hebras intenten realizar escrituras atómicas en la misma posición de la matriz.

TABLA 5.6: Speedup y tiempo de ejecución para distintas discretizaciones del universo de simulación.

| dr and dz | nr | nz | GPU (s) | CPU (s) | Speedup |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 201.56 | 1264.56 | 6x |
| 0.1 | 10 | 10 | 130.21 | 1310.62 | 10x |
| 0.01 | 100 | 100 | 134.74 | 1317.80 | 10x |
| 0.001 | 1000 | 1000 | 127.91 | 1323.36 | 10x |

TABLA 5.7: Speedup y tiempo de ejecución para distintas discretizaciones del universo de simulación utilizando la versión optimizada del software.

| dr and dz | nr | nz | GPU Opt (s) | CPU (s) | Speedup |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 94.34 | 1264.56 | 13x |
| 0.1 | 10 | 10 | 42.79 | 1310.62 | 31x |
| 0.01 | 100 | 100 | 36.84 | 1317.80 | 36x |
| 0.001 | 1000 | 1000 | 44.46 | 1323.36 | 30x |

### 5.2.3 Bloques y hebras por bloque

CUDA permite realizar llamados a *kernels* con diferentes bloques e hilos por bloque. Un bloque es un conjunto de hilos. Cada multiprocesador de la tarjeta de video ejecuta lotes de bloques uno detrás otro.

Cada una de las generaciones de CUDA posee diferentes restricciones para la manera en que se puede organizar una grilla (conjunto de bloques) y utilizarla en *kernels*. Dichas restricciones pueden ser categorizadas en tres tipos:

1. Cantidad de bloques concurrentes en cada multiprocesador.

2. Cantidad de hilos por bloque.

3. Cantidad total de hilos por multiprocesador.

Atendiendo la gran diversidad de posibilidades se han realizado varios experimentos aplicando cambios a la cantidad de bloques e hilos por bloque, tal como muestran las tablas 5.8 y 5.9.

TABLA 5.8: Tiempo de ejecución y speedup obtenido para 25 configuraciones distintas de grillas sobre un mismo experimento.

| Blocks | Threads per block | GPU (s) | CPU (s) | Speedup |
|---|---|---|---|---|
| 14 | 32 | 944.64 | 1222.4 | 1x |
| 14 | 64 | 508.41 | 1222.4 | 2x |
| 14 | 128 | 306.31 | 1222.4 | 4x |
| 14 | 256 | 210.32 | 1222.4 | 6x |
| 14 | 512 | 151.64 | 1222.4 | 8x |
| 28 | 32 | 507.34 | 1222.4 | 2x |
| 28 | 64 | 287.36 | 1222.4 | 4x |
| 28 | 128 | 192.82 | 1222.4 | 6x |
| 28 | 256 | 156.61 | 1222.4 | 8x |
| 28 | 512 | 138.75 | 1222.4 | 9x |
| 42 | 32 | 355.39 | 1222.4 | 3x |
| 42 | 64 | 215.80 | 1222.4 | 6x |
| 42 | 128 | 161.79 | 1222.4 | 8x |
| 42 | 256 | 146.59 | 1222.4 | 8x |
| 42 | 512 | 145.56 | 1222.4 | 8x |
| 56 | 32 | 279.6 | 1222.4 | 4x |
| 56 | 64 | 181.9 | 1222.4 | 7x |
| 56 | 128 | 148.12 | 1222.4 | 8x |
| 56 | 256 | 143.13 | 1222.4 | 9x |
| 56 | 512 | 135.30 | 1222.4 | 9x |
| 70 | 32 | 238 | 1222.4 | 5x |
| 70 | 64 | 166.26 | 1222.4 | 7x |
| 70 | 128 | 144.34 | 1222.4 | 8x |
| 70 | 256 | 149.42 | 1222.4 | 8x |
| 70 | 512 | 130.70 | 1222.4 | 9x |

TABLA 5.9: Tiempo de ejecución y speedup obtenido para 25 configuraciones distintas de grillas sobre un mismo experimento utilizando la versión optimizada del software.

| Blocks | Threads per block | GPU (s) | CPU (s) | Speedup |
|---|---|---|---|---|
| 14 | 32 | 268.62 | 1222.4 | 5x |
| 14 | 64 | 136.78 | 1222.4 | 9x |
| 14 | 128 | 72.04 | 1222.4 | 17x |
| 14 | 256 | 37.41 | 1222.4 | 33x |
| 14 | 512 | 38.15 | 1222.4 | 32x |
| 28 | 32 | 135.76 | 1222.4 | 9x |
| 28 | 64 | 69.98 | 1222.4 | 17x |
| 28 | 128 | 44.38 | 1222.4 | 28x |
| 28 | 256 | 38.77 | 1222.4 | 32x |
| 28 | 512 | 38.28 | 1222.4 | 32x |
| 42 | 32 | 91.22 | 1222.4 | 13x |
| 42 | 64 | 48.36 | 1222.4 | 25x |
| 42 | 128 | 46.63 | 1222.4 | 26x |
| 42 | 256 | 40.30 | 1222.4 | 30x |
| 42 | 512 | 38 | 1222.4 | 32x |
| 56 | 32 | 69.35 | 1222.4 | 18x |
| 56 | 64 | 40.33 | 1222.4 | 30x |
| 56 | 128 | 39.26 | 1222.4 | 31x |
| 56 | 256 | 38.64 | 1222.4 | 32x |
| 56 | 512 | 36.22 | 1222.4 | 34x |
| 70 | 32 | 55.83 | 1222.4 | 22x |
| 70 | 64 | 41.70 | 1222.4 | 29x |
| 70 | 128 | 40.80 | 1222.4 | 30x |
| 70 | 256 | 37.53 | 1222.4 | 33x |
| 70 | 512 | 36.80 | 1222.4 | 33x |

A partir de los experimentos realizados es posible notar que un aumento en la cantidad de bloques produce una disminución en los tiempos de ejecución. Esto es totalmente esperable debido a que al existir más bloques por procesar al mismo tiempo se disminuyen los tiempos asociados a la latencia de lanzar un *kernel* a GPU.

Por otro lado queda en claro que los mayores *speedups* se producen para 512 bloques por hebra, lo cual corresponde a lo que indica NVIDIA para la generación Fermi.

A partir de estos experimentos, además es posible afirmar que la latencia se constituye como un factor determinante en el performance obtenido por este tipo de dispositivos. Este factor es mucho más evidente si se analiza la gran diferencia existente entre la versión normal del software la cual utiliza varios procesos físicos separados en distintos *kernels* y la versión optimizada, que unifica

todos los *kernels* en uno, utilizando este último varias veces antes de retornar el hilo de ejecución al *host*.

# CAPÍTULO 6. CONCLUSIONES

Por medio del presente trabajo se ha obtenido una estrategia de paralelización de simulaciones Monte Carlo, la cual ha sido implementada para el fenómeno de bioluminiscencia. Como base del trabajo se ha utilizado un modelo de simulación existente, el cual hace uso de objetos concurrentes por medio de $\mu$C++. No obstante, la implementación actual se ha llevado a cabo sobre tarjetas gráficas utilizando CUDA.

El modelo que se ha propuesto para GPU difiere de otros fundamentalmente en su capacidad de plantear simulaciones de manera genérica sin pensar en una implementación particular. Además permite modelar las entidades del universo de simulación por medio de orientación al objeto y los procesos físicos a través de *kernels*, unificando todo mediante un modelo de control de la simulación.

Las validaciones de los resultados se han realizado a través de pruebas unitarias. Los experimentos a los cuales fue sometido el software muestran que las partículas presentan un comportamiento ante los procesos físicos de acuerdo a las reglas señaladas por la teoría. Además los experimentos de absorción de energía y fluencia presentan resultados equivalentes a trabajos realizados por otros investigadores.

Debido a lo anteriormente expuesto, es posible afirmar que se han cumplido los objetivos planteados al comienzo de este documento. Se han encontrado oportunidades de paralelización, se ha adaptapdo un modelo de simulación para utilizarlo GPGPU, el cual fue implementado para bioluminiscencia y finalmente se diseñaron y realizaron experimentos para evaluar y validar los resultados obtenidos.

En cuanto a rendimiento computacional, al comparar los tiempos de ejecución con una implementación secuencial de otro software que posee el mismo propósito, se obtuvo un *speedup* máximo de 33x. Si bien en la literatura es posible encontrar implementaciones con *speedups* superiores, el software generado en este trabajo no posee ningún tipo de optimización para memoria,

lo cual permite mejorar el rendimiento dramáticamente (Alerstam et al., 2010; Fang & Boas, 2009). Cabe destacar que el mayor beneficio de la solución generada está dado por el nivel de abstracción con el que la simulación es definida. Gracias al uso de la orientación al objeto se puede extender la solución a otro tipo de simulaciones de eventos discretos e independientes. En el futuro resultaría interesante estudiar de qué manera es posible optimizar el uso de memoria, transferencia de datos y la latencia involucrada en la utilización de *kernels*. Por ejemplo, es posible almacenar en memoria cache las posiciones de la matriz de absorción cercanas a las fuentes de fotones, puesto que estas localidades son las más accedidas durante las mediciones. Otra posibilidad es unir las funcionalidades de ciertos *kernels* de tal modo de reducir la latencia involucrada en su invocación. Finalmente, se podría construir un algoritmo de invocación de kernels, por medio del cual éstos no sean invocados hasta que existan suficientes datos para optimizar su funcionamiento. En caso de que no se cuente con suficientes datos, se volvería a invocar a los kernels anteriores, hasta que se produjeran datos suficientes para avanzar a la siguiente etapa.

Aún está por demostrarse cuán fácil es extender el software a otras áreas de física médica. No obstante, se sospecha que no será una tarea sencilla, ya que se requiere un importante esfuerzo para implementar de manera más elaborada las interacciones físicas y el modelamiento de cuerpos complejos.

Aunque ha sido desarrollado un modelo para simulaciones en GPU, cabe enfatizar que los problemas a los cuales es aplicable deben cumplir ciertas condiciones que fueron mencionadas en el capítulo 1. En primer lugar, el problema debe involucrar una gran cantidad de datos, ya que el costo de invocación de *kernels* es alto. Y segundo, debe ser posible ajustar los pasos de resolución del problema al modelo SIMT, puesto que de lo contrario las divergencias existentes no permitirán obtener un gran provecho de la tecnología.

A pesar de todas las ventajas que provee este nuevo modelo de cómputo masivo y paralelo, es necesario mencionar que estas tecnologías requieren de un mayor grado de conocimento del programador. Dicho conocimiento abarca desde el modelo de computación hasta cómo está organizado el hardware, todo con el fin de explotar al máximo las capacidades de las tarjetas gráficas. Aún estas tecnologías se encuentran en desarrollo y continúan avanzando en medro del rendimiento y la facilidad de uso.

Finalmente se concluye que la utilización de GPU es una estrategia prometedora para la

simulación de eventos discretos, otorgando a un bajo costo un alto poder de procesamiento.

# REFERENCIAS


Agostinelli, S., & et al (2003). GEANT4 - A Simulation Toolkit. *Nuclear Instruments and Methods in Physics Research*, (pp. 250–303).

Alerstam, E., Lo, W. C. Y., Han, T. D., Rose, J., Andersson-Engels, S., & Lilge, L. (2010). Next-generation acceleration and code optimization for light transport in turbid media using GPUs. *Biomedical Optics Express*.

Alexandrakis, G., Rannou, F. R., & Chatziioannou, A. F. (2005). Tomographic bioluminescence imaging by use of a combined optical-pet (opet) system: a computer simulation feasibility study. *Physics in Medicine and Biology*, *50*(17), 4225.

Alexandrakis, G., Rannou, F. R., & Chatziioannou, A. F. (2006). Effect of optical property estimation accuracy on tomographic bioluminescence imaging: simulation of a combined optical–pet (opet) system. *Physics in Medicine and Biology*, *51*(8), 2045.

Beenhouwer, J. D., andD. Kruecker, S. S., Ferrer, L., D'Asseler, Y., Lemahieu, I., & Rannou, F. (2007). Cluster computing software for GATE simulations. *Medical Physics*, *34*, 1926–1933.

Binder, K., & Heermann, D. (2002). *Monte Carlo Simulation in Statistical Physics. An Introduction*. Springer, 4th ed.

Buhr, P. A., Ditchfield, G., Stroobosscher, R. A., Younger, B. M., & Zarnke, C. R. (1992). $\mu$c++: Concurrency in the object-oriented language c++. *SOFTWARE-PRACTICE AND EXPERIENCE*, (pp. 137–172).

Bushberg, J. T., Seibert, J. A., Leidholdt, E. M., & Boone, J. M. (2002). *The essential of physics of medical imaging*. Lippincott Williams, Wilkins, 2nd ed.

Cooperman, G., Nguyen, V., & Malioutov, I. (2006). Parallelization of geant4 using top-c and marshalgen. In *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, (pp. 48–55).

Doll, T., & Freeman, D. L. (1994). Monte Carlo Methods in Chemistry. *IEEE Computational Science & Engineering*, (pp. 22–32).

Dong, X., Cooperman, G., & Apostolakis, J. (2012). GEANT4: Semi-automatic transformation into scalable thread-parallel software. In *16th International Euro-Par Conference (Euro-Par 2010)*, (pp. 287–303).

Fang, Q., & Boas, D. A. (2009). Monte CArlo simulation of photon migration in 3D turbid media accelerated by graphics processing units. *Optics Express*.

Glasserman, P. (2004). *Monte Carlo Methods in Financial Engineering*. Applications of mathematics : stochastic modelling and applied probability. Springer.

James, F. (1980). Monte carlo theory and practice. *Reports on Progress in Physics*, *43*, 1145–1189.

Jan, S., & et al (2004). Gate: a simulation toolkit for pet and spect. *Physics in Medicine and Biology*, *49*, 4543–4561.

Kalos, M., & Whitlock, P. (2009). *Monte Carlo Methods*. Wiley.

Lagos, H. (2012). Modelamiento de simulaciones Monte Carlo de eventos discretos e indepentientes e implementación orientada a objetos concurrentes. *Trabajo de título, Ingeniería civil informática. Universidad de Santiago de Chile*.

Maigne, L., Hill, D., Calvat, P., Breton, V., Reuillon, R., Lazaro, D., Legre, Y., & Donnarieix, D. (2004). Parallelization of Monte Carlo simulations and submission to a grid environment. *Parallel Processing Letters*, *14*(02), 177–196.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU Computing. *Proceedings of IEEE*, (pp. 879–899).

Park, H., & Fishwick, P. A. (2010). A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation. *Journal Simulation*, *86*, 613–628.

Rannou, F., Vega-Acevedo, N., & Bitar, Z. E. (2013). A parallel computational model for GATE simulations. *Computer Methods and Programs in Biomedicine*.

Rehemtulla, A., Hall, D. E., Stegman, L. D., Prasad, U., Chen, G., Bhojani, M. S., Chenevert, T. L., & Ross, B. D. (2002). Molecular Imaging of Gene Expression and Efficacy following Adenoviral-Mediated Brain Tumor Gene Therapy. *Molecular Imaging*, *1*, 43–55.

Rehemtulla, A., Stegman, L. D., Cardozo, S. J., Gupta, S., Hall, D. E., Contag, C. H., & Ross, B. D. (2000). Rapid and Quantitative Assessment of Cancer Treatment Response Using In Vivo Bioluminescence Imaging. *Neoplasia*, *2*, 491–495.

Rogers, D. W. O. (2006). Fifty years of monte carlo simulations for medical physics. *Physics in Medicine and Biology*, *51*(13), R287.
URL `http://stacks.iop.org/0031-9155/51/i=13/a=R17`

Schellenberger, J. (2010). *Monte Carlo Simulation in Systems Biology*. Ph.D. thesis, La Jolla, CA, USA. AAI3422922.

Staelens, S., Beenhouwer, J. D., Kruecker, D., Maigne, L., Rannou, F., Ferrer, L., D'Asseler, Y., Buvat, I., & Lemahieu, I. (2006). GATE: improving the computational efficiency. *Nuclear Instruments and Methods in Physics Research Section A, A569*, (pp. 341–345).

Torres-Tramon, P., Vega-Acevedo, N., & Rannou, F. R. (2010). Multithreading GATE. In *IEEE Nuclear Science Symposium and Medical Imaging Conference Knoxville*.

Wang, G., Cong, W., Durairaj, K., Qian, X., Shen, H., Sinn, P., Hoffman, E., McLennan, G., & Henry, M. (2006). In vivo mouse studies with bioluminescence tomography. *Opt. Express*, (pp. 7801–7809).

Wang, L., & Jacques, S. L. (1992). Monte Carlo Modeling of Light Transport in Multi-layered Tissues in Standard C. *University of Texas M. D. Anderson Cancer Center*.

Xu, Y., Yu, M., Zhang, C., & Yang, B. (2013). The analysis of generic SIMT scheduling model extracted from GPU. In W. Xu, L. Xiao, C. Zhang, J. Li, & L. Yu (Eds.) *Computer Engineering and Technology*, vol. 396 of *Communications in Computer and Information Science*, (pp. 9–18). Springer Berlin Heidelberg.

# APÉNDICE A. ARCHIVO DE ENTRADA

```
# Archivo de entrada para la simulacion
########################################
1                                # numero de simulaciones

### Archivo de salida
output/stdexp.out   A            # nombre de archivo, ASCII/Binary

.01 .01                          # dz, dr

40  50  1                        # No. de dz, dr & da.
5                                # No. de capas
# n mua mus g   d                # Una linea por cada capa
1.0                              # n para el medio superior
1.37    1   100 0.9     0.1      # capa 1
1.37    1   10  0       0.1      # capa 2
1.37    2   10  0.7     0.2      # capa 3
1.0                              # n para el medio inferior

1                                # No. de fuentes

# No. fotones x y z ux uy uz apertura_angular (radianes)
1000000 0   0   0   0   0   1   0
```