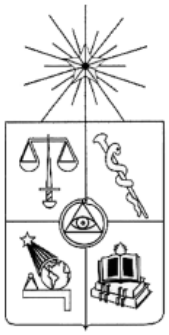


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

VISUALIZADOR DE SISTEMAS DE PARTÍCULAS EN 2D Y 3D

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

LISSETTE PAULINA CABRERA DÍAZ

PROFESORES GUÍA:
RODRIGO SOTO BERTRÁN
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA RIVARA ZÚÑIGA

SANTIAGO DE CHILE
JUNIO 2012

Resumen

El trabajo desarrollado durante este tema de memoria se enmarcó en el ámbito de experimentación física sobre sistemas de partículas. El objetivo fue diseñar y desarrollar una herramienta de visualización de sistemas de partículas en 3D, que permita manejar grandes volúmenes de datos, visualizarlos rápidamente y proveer controles de visualización.

El Departamento de Física de esta Facultad realiza diversos experimentos y simulaciones computacionales de materiales granulares compuestos por miles o millones de partículas esféricas. Una buena herramienta de visualización de estas simulaciones permite a los investigadores realizar un análisis más riguroso y concluir correctamente sobre los experimentos. Las herramientas con que se contaba, previo a este trabajo, poseían falencias importantes que no permitían obtener el máximo provecho de las simulaciones.

El trabajo realizado se dividió en cuatro etapas. La primera etapa consistió en recoger los requerimientos que debía satisfacer la aplicación para cumplir con los objetivos planteados. En base a los requerimientos tomados se comenzó la segunda etapa, diseñar la arquitectura del sistema. La arquitectura debió considerar un software creado previamente para monitorear las simulaciones, el cual establece la conexión con éstas y obtiene su información y datos relevantes. Un hito desafiante en esta etapa fue adaptar la herramienta disponible a los requerimientos pedidos. Como resultado de esta segunda etapa se obtuvo un diseño de la estructura del software utilizando el paradigma de programación orientada a objetos y utilizando diversos patrones de diseño a fin de mejorar la calidad del software. La tercera etapa consistió en implementar la aplicación. Para ello, se creó una interfaz de usuario adecuada a los requisitos y con herramientas de control que faciliten el uso de la aplicación. Luego se implementaron las funcionalidades disponibles en la interfaz, dentro de las cuales se encuentran: Rotar libremente la escena, distinguir partículas en base a su tipo y velocidad, graficar partículas de distinto radio, conectarse a distintas simulaciones, generar grabaciones de video y capturar imágenes de la visualización, entre otras. La última etapa consistió en *testing* y optimización. Se hicieron pruebas para medir la calidad de la aplicación, optimizando el software de acuerdo a los resultados obtenidos.

Como resultado se obtuvo una aplicación robusta, extensible y fácil de usar, la cual permite visualizar simulaciones físicas de sistemas de alrededor de 500.000 partículas de manera eficiente y facilita el análisis de éstas.

Agradecimientos

Primeramente a Dios, quien es mi Padre y Salvador.
"Porque de Él, y por Él, y para Él, son todas las cosas. A Él sea la gloria por los siglos. Amén."
Rom.11:36

A mi abuelo, Oscar Cabrera, fuente de inagotable inspiración. Un hombre autodidacta e ingeniero por naturaleza. Lleno de bondad y generosidad. Lamentablemente dejó este mundo a mitad de este trabajo, pero sé que verá su sueño concluido desde el lugar celestial.

A mi familia, que siempre me ha apoyado y alentado, en las buenas y en las malas. Especialmente a mis padres, que se han sacrificado y esforzado por darme siempre lo mejor, incluso más allá de sus fuerzas. A mis hermanos, por soportar mi mal humor cuando las cosas se ponían difíciles y por confiar siempre en mí.

A Guillermo por la infinidad de detalles que siempre alegran mis días.

A mis compañeros de carrera Carlos y Roberto, con quienes compartí y seguiré compartiendo hermosos momentos más allá de la computación.

A Iván Pablo, por su apoyo incondicional y compañía durante toda la carrera. Por no dejar que me rindiera y alentarme siempre a continuar. Por creer en mí y hacerme crecer.

A mis compañeros de Universidad, Moisés y Nicolás, que gracias a su amistad y cariño hicieron más llevadera la carga.

Tabla de contenidos

Índice de figuras

1 Introducción

1.1 Aspectos generales

El tema a desarrollar durante la memoria se enmarcó en el ámbito de experimentación física sobre sistemas de partículas. Un sistema de partículas es un conjunto de partículas cuyas propiedades globales son objeto de estudio. Para ello se consideran las fuerzas exteriores que afectan al sistema, y las fuerzas interiores provenientes de la interacción entre las propias partículas del sistema. Los sistemas de partículas son útiles para obtener conclusiones sobre cuerpos continuos, considerando los resultados sobre un sistema discreto de muchas partículas.

El Departamento de Física de esta Facultad realiza diversos experimentos y simulaciones computacionales de materiales granulares compuestos por miles o millones de partículas esféricas. Esto genera una gran cantidad de datos que requieren ser visualizados en tiempo real o a posteriori. En los experimentos la cantidad de partículas es del orden de 10.000 y en las simulaciones la cifra crece a 500.000. Esto hace necesario un visualizador eficiente capaz de manejar estas grandes cantidades.

Usualmente los sistemas de partículas que simulan escenarios naturales tales como fuego, nubes, fluidos, etc. realizan las simulaciones sobre la CPU. Sin embargo, si el número de partículas sobrepasa las 10.000, dicho sistema será muy difícil de correr sobre una CPU en tiempo real [1]. Por convención, una técnica de *rendering* se considera de ejecución en tiempo real, si es capaz de graficar en pantalla al menos 15 veces en un segundo [2]. La cantidad de veces que se genera una imagen por segundo es llamada *frames per second*, *fps*. Hoy en día, con el desarrollo de las GPUs, es posible tratar estos problemas de cálculos complejos en tiempo real de manera eficiente y rápida.

1.2 Motivación

Uno de los experimentos realizados consiste en confinar miles de partículas en una caja, que puede ser de muy poca altura (cuasi 2D) o de cualquier otra altura arbitraria (3D), la cual es sometida a vibración mediante movimientos verticales. A través de este experimento, se ha observado que el medio granular desarrolla una fase en la que se comporta como un fluido que puede tener una transición de fase a un sólido cristalino (las partículas se ordenan como una cuadrícula).

Para realizar un análisis más riguroso y concluir correctamente sobre los experimentos es necesaria una herramienta que permita visualizar el experimento desde diferentes ángulos. Esta herramienta debe ofrecer opciones de manipulación de las imágenes capturadas.

Además de los experimentos mencionados, se realizan simulaciones que permiten manejar una cantidad aún mayor de datos, de alrededor de 500.000 partículas. Actualmente, las simulaciones son realizadas por un software que predice en qué momento ocurren choques entre las partículas. La simulación es dirigida por eventos, donde cada evento es el momento en el que

ocurre un choque entre partículas. Previo a este trabajo, dentro de la línea de eventos se introducían eventos de visualización, es decir, se generaban imágenes del sistema de partículas para momentos dados de la simulación. Esto limitaba el análisis, ya que sólo se obtenían imágenes de la simulación en momentos establecidos previamente, pudiendo perderse la observación de fenómenos interesantes que no se registraban entre los eventos de visualización. Además, como el mismo software de simulación era el encargado de ejecutar las llamadas a las rutinas de visualización, se tenía un sistema muy acoplado, lo que dificultaba su mantención y extensibilidad.

Debido a que las simulaciones deben manejar grandes cantidades de datos, éstas son ejecutadas en un *cluster*. En el *cluster* no es posible realizar la visualización con el software actual, ya que el *cluster* no permite visualizar los datos.

Para dar solución al problema expuesto, es necesario desarrollar un visualizador que se adapte a la arquitectura cliente-servidor, para así separar la simulación de la visualización. De esta manera la simulación del sistema de partículas se realizaría en un servidor, y la visualización en un cliente. Este cliente podría estar en cualquier otro computador recibiendo los datos de la simulación y llamando a la rutina de visualización. Así se lograría desacoplar totalmente el sistema actual.

Actualmente, ya se cuenta con un programa cliente, el cual fue desarrollado por un estudiante de la Universidad del Bío Bío [3]. Sin embargo, este cliente sólo establece la comunicación con el servidor y recibe los datos de la simulación, pero no hace nada con ellos.

1.3 Objetivos

1.3.1 Objetivo General

Desarrollar una herramienta de visualización de sistemas de partículas, en 2D y 3D, que permita visualizar de manera eficiente alrededor de 500.000 partículas y mostrar sus propiedades físicas de manera precisa y clara. Además, la herramienta debe ser capaz de producir videos y fotos con calidad de publicación.

1.3.2 Objetivos Específicos

- Proveer vistas ortogonales. Esto es, el sistema visualizado debe quedar representado por su vista frontal (proyección ortogonal en el plano vertical XY), su vista superior (proyección ortogonal en el plano horizontal XZ) y su vista lateral (proyección ortogonal en el plano lateral YZ).
- Permitir la rotación libre de la vista.
- Obtener los datos de entrada mediante comunicación con un programa cliente que recibe los datos enviados por red desde un servidor donde se realiza la simulación.
- Proveer controles que permitan cambiar los parámetros de la simulación (ej: gravedad,

densidad, etc.) o realizar mediciones de interés (ej: fuerzas, temperatura, etc.).

- Visualizar en tiempo real (15 *fps*.) [2] para al menos una carga de 500.000 partículas y permitir la generación de videos de la visualización, así como imágenes en alta calidad.
- Ser portable a plataformas Linux y MacOSX.
- Permitir la distinción de las partículas de acuerdo a propiedades que se deseen estudiar, tales como, energía cinética, material de la partícula, velocidad vectorial, etc.
- Optar entre dos modos de visualización: considerando la dirección del vector de aceleración de la gravedad (g) fija o variable. El rol del vector de gravedad se explica más en detalle en la sección 2.5.2 de este informe.
- Diseñar la aplicación de forma que sea fácilmente extensible a nuevas funcionalidades, como por ejemplo la visualización de otras figuras geométricas, y que sea escalable, manteniendo la calidad de visualización para grandes volúmenes de datos.

1.4 Contenido

Esta memoria está agrupada como sigue:

1. **Introducción:** Se presenta una introducción al tema, en sus aspectos generales. Se expone la motivación y justificación del presente trabajo. Se enuncian también los objetivos del proyecto.
2. **Antecedentes:** Se presentan conceptos necesarios para la comprensión del trabajo realizado, explicados en forma detallada. Se presenta también revisión de trabajos relacionados.
3. **Diseño:** Se presenta un análisis de la solución desarrollada y se listan los requerimientos que ésta debe satisface. Se evalúan las distintas alternativas tecnológicas y se justifica la elección de unas por sobre otras. También se describe la arquitectura de la aplicación desarrollada.
4. **Implementación:** Se presentan detalles del proceso de implementación del proyecto. Se explican los diferentes desafíos de implementación enfrentados. También se muestran resultados obtenidos y se comparan las distintas estrategias disponibles para resolver el problema.
5. **Conclusiones:** Se exponen los resultados obtenidos y las conclusiones del trabajo realizado. Se analiza el posible trabajo futuro, y se proponen mejoras a la aplicación desarrollada.
6. **Referencias:** Se lista las referencias y bibliografía utilizada como fuente de información durante el desarrollo de la memoria.
7. **Anexos:** Se muestra información adicional y de interés sobre el desarrollo de la memoria.

2 Antecedentes

A continuación se exponen conceptos necesarios para comprender el trabajo realizado. Se incluyen conceptos tanto del área de simulaciones físicas, como de ingeniería de software. Además se muestra información bibliográfica de trabajos relacionados.

2.1 Conceptos básicos

2.1.1 Definiciones

GPU (acrónimo del inglés *Graphics Processing Unit*): La GPU es un dispositivo dedicado completamente al procesamiento gráfico, manipulando de manera rápida y eficiente la memoria disponible a fin de acelerar la construcción de imágenes.

Pipeline gráfico: Se refiere al método de rasterización (generar una imagen desde un modelo) basada en el soporte de hardware de gráficos (tarjeta gráfica). Generalmente un pipeline gráfico acepta como input una representación de una escena en 3D y entrega como output un mapa de bits de la escena en 2D. Diseñadores de GPU tradicionalmente han expresado este proceso de síntesis de imagen en un pipeline de hardware de etapas especializadas, de las cuales 3 son programables por el usuario y pueden recibir instrucciones en lenguaje de *shading*. Las distintas alternativas de lenguajes de *shading* se discutirán más adelante.

Shader: Set de instrucciones escritas en un lenguaje de *shading* que pueden ser ejecutadas por un procesador dentro del pipeline gráfico de una GPU.

Rendering: Se refiere al proceso realizado por una computadora destinado a generar una imagen en 2D a partir de un modelo en 3D.

2.1.2 Funcionamiento de la GPU

En el artículo [4] se describe el funcionamiento básico de la GPU y su pipeline de *rendering*. La GPU posee un pipeline de hardware de etapas especializadas, de las cuales destacan tres que en las tarjetas más modernas son programables y pueden recibir instrucciones en lenguaje de *shading*. Estas etapas son: el procesador de vértices, el procesador de geometría y el procesador de píxeles.

El procesador de vértices se encarga de las transformaciones que afectan los vértices de las figuras en la escena. Maneja todos los cálculos sobre las características de un vértice, tales como coordenadas, tamaño, color, textura, normal, etc.

El procesador de geometría se encarga de generar nuevas primitivas dinámicamente así como de modificar existentes.

El procesador de píxeles se encarga de las transformaciones sobre los píxeles, tales como cambiar su profundidad, calcular efectos de iluminación con gran precisión, etc. A fin de determinar el color que debería aplicarse sobre el píxel en caso de ser usado.

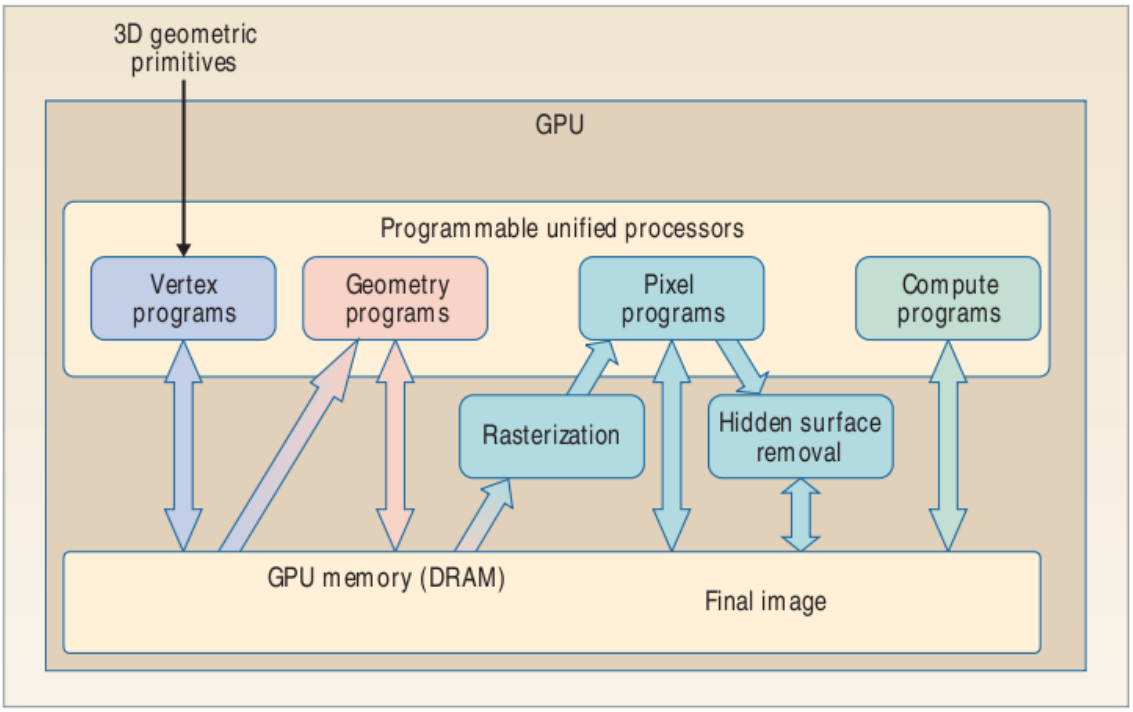


Figura 1 Pipeline de Hardware de una GPU.

2.2 Revisión bibliográfica

Como parte de la etapa de estudio del proyecto, se revisó bibliografía relacionada con simulaciones físicas en computadoras y técnicas de *rendering* de imágenes. En particular, se destacan los siguientes *papers* y artículos, los cuales se relacionaban de manera cercana al proyecto que se iba a desarrollar.

2.2.1 Simulación y visualización de sistemas de partículas

En el artículo [5] se presenta una implementación sobre la GPU de una simulación y visualización en tiempo real de un sistema compuesto por un millón de partículas. Se explican la diferencias entre los sistemas de partículas que preservan estado, de los que no. Los primeros son aquellos que para calcular la posición y velocidad actual de cada partícula se basan en sus valores previos y en la descripción de cambios en el ambiente. En cambio, los sistemas de partículas que no preservan estado calculan la posición de cada partícula basándose en fórmulas definidas por parámetros iniciales y el tiempo actual, y no reaccionan a ambientes dinámicos.

El texto introduce al lector al funcionamiento interno de una GPU y cómo se explotan sus capacidades para incrementar la eficiencia de una simulación de un sistema de partículas que preserva estado y su *rendering*.

En este trabajo existen dos puntos de interés para este tema de memoria: La asignación de espacio en la memoria interna de la GPU para la representación de las partículas aumenta el desempeño de la simulación y visualización pues reduce la comunicación con la memoria de la CPU acelerando el *rendering*. El algoritmo de ordenamiento de partículas implementado sobre la GPU, antes de su *rendering*, determina las que se encuentran más cercanas al punto de vista del observador y por lo tanto las que tapan a las de más atrás, lo cual también contribuye acelerando el *rendering*.

2.2.2 Visualización en tiempo real de grandes sistemas de partículas

El artículo [6] presenta un método para implementar efectos complejos de simulación basados en grandes sistemas de partículas ejecutados sobre la GPU. El sistema de partículas utilizado es uno que preserva estado. En este trabajo el *rendering* de las partículas se realiza de a grupos sobre la GPU. También se realiza una comparación entre sistemas de partículas implementados sobre la CPU e implementados sobre la GPU.

El texto explica que la posición de cada partícula es almacenada en una textura de punto flotante con 3 componentes de colores que serán tratadas como sus coordenadas (x,y,z). Se utilizan dos texturas y una técnica de doble buffer para calcular nuevos datos usando los valores previos. El método utilizado consiste en 5 pasos básicos:

- Procesar nacimiento y muerte de las partículas
- Actualizar atributos de las partículas
- Detección de colisiones
- Transferencia de los datos en las texturas a datos de vértices
- *Rendering* de a grupos de partículas

Para comparar el rendimiento de sistemas de partículas implementados sobre la CPU y sistemas de partículas implementados sobre la GPU se realizó el siguiente experimento: Se comparó la cantidad de cuadros por segundo (FPS) de sistemas con la misma cantidad de partículas implementados tanto en la CPU como en la GPU.

Como resultado se obtuvo que sistemas con 100.000 partículas registraron 60fps en la GPU mientras que, bajo similares condiciones, registraron 18fps en la CPU. Cuando el número de partículas creció a 200.000, se registraron 36fps en la GPU, en cambio sólo 8fps en la CPU.

2.3 Trabajo previo

2.3.1 Cliente: SiMon

En el presente tema de memoria se utiliza el trabajo realizado por un alumno de la Universidad del Bío Bío, el cual consta de un programa cliente usado para comunicarse con simulaciones que se ejecutan en un servidor externo. Como parte del periodo de investigación se estudió el funcionamiento de dicho programa, el cual se describe a continuación.

El objetivo principal del programa cliente es monitorear de forma remota simulaciones que corren en un clúster de procesadores. El programa fue denominado SiMon, por su abreviación del inglés *Simulation Monitor*, está escrito en lenguaje C y es compatible con plataformas Linux y MacOS. Dentro de sus funciones básicas SiMon permite:

- Listar las simulaciones que se encuentran en ejecución
- Modificar los parámetros de las simulaciones mientras se ejecutan
- Realizar mediciones de interés sobre las simulaciones
- Hacer llamadas a rutinas de visualización de las partículas

Previo a este trabajo, SiMon funcionaba a través de línea de comando, y no poseía interfaz gráfica. El visualizador desarrollado en este tema de memoria, mejoró sustancialmente la interacción gráfica con las simulaciones, ya que uno de los objetivos era que el visualizador interactuara con SiMon y permitiera realizar las funciones antes listadas a través de controles gráficos (botones, menús, campos de texto, etc.).

Para listar las simulaciones que se encuentran en ejecución, se utiliza en el servidor un servicio que está en permanente ejecución (demonio) esperando mensajes de SiMon. Como única tarea el demonio responde al comando "#list", ejecutado por SiMon, con la información que éste posee sobre las simulaciones en ejecución (nombre, puerto, directorio y fecha/hora de comienzo).

Para la comunicación remota entre SiMon y el servidor donde se ejecutan las simulaciones se utiliza el protocolo UDP, que tiene la característica de ser bastante rápido ya que no establece una conexión, sino que incorpora en cada paquete de datos (datagrama) la información necesaria para su direccionamiento. En el servidor se utilizan dos hilos paralelos, uno para correr la simulación y otro para recibir y responder a los mensajes enviados por SiMon.

Para la visualización de partículas, SiMon realizaba dos tipos de animaciones básicas, utilizando las librerías Libplot y OpenGL. Las visualizaciones obtenidas no permitían hacer observaciones, sin embargo, fueron implementadas como pauta para el trabajo posterior. La Figura 2 muestra dos capturas de pantalla mientras se ejecutaban las animaciones en SiMon.

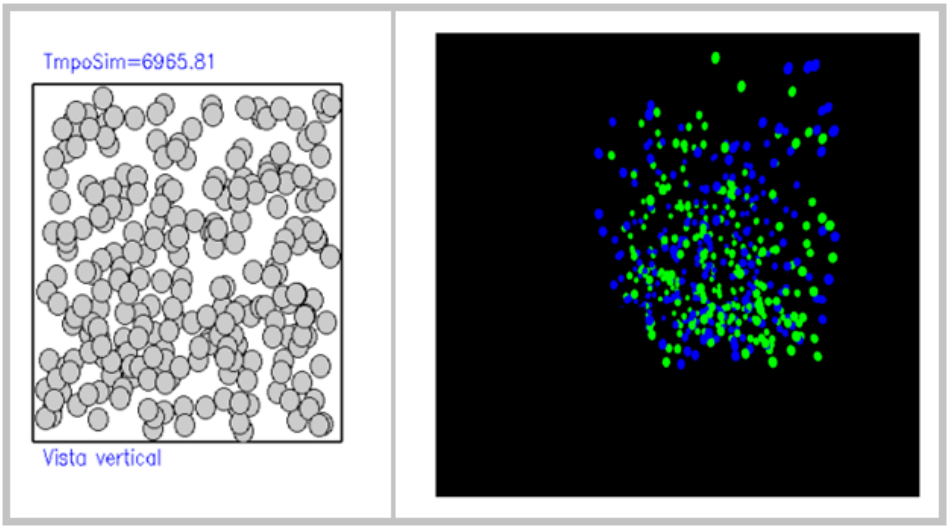


Figura 2 Animaciones desplegadas por SiMon (Izq.) Animación utilizando Libplot. (Der) Animación utilizando OpenGL.

SiMon es capaz de recibir comandos y ejecutar una acción de acuerdo a ese comando. Dentro de los principales comandos se destacan los siguientes, los cuales son de interés para la aplicación que será desarrollada en este trabajo:

connect [port]: Crea una conexión con el servidor en el puerto especificado como argumento.

send : Envía un comando al servidor.

2.3.2 Servidor: Simulador

El servidor al cual se conecta SiMon está conformado por dos componentes principales: el programa que realiza la simulación de medios granulares, en adelante 'el simulador', y el servidor propiamente tal que espera mensajes externos y reacciona ante estos.

El simulador fue diseñado inicialmente como parte del trabajo de Doctorado del profesor Dino Risso (Universidad del Bío Bío) [7] y como tema principal de la tesis de Magister del profesor Mauricio Marín (Universidad de Chile) [8]. Está escrito en lenguaje C y se mantiene como un código privativo del *Grupo de Dinámica Molecular y Teoría Cinética (GDMTC)* el cual está conformado por los académicos Patricio Cordero S. y Rodrigo Soto B. de la Universidad de Chile, y el académico Dino E. Risso de la Universidad del Bío-Bío.

El funcionamiento del simulador es complejo y no se detallará en este informe, ya que está fuera del alcance del trabajo realizado. Sin embargo, es importante describir una

característica particular del simulador, la cual se refiere al cálculo del vector de aceleración de la gravedad (*g*), ya que es un dato relevante a la hora de visualizar.

Existen simulaciones en las que se desea girar o agitar el contenedor del medio granular simulado. Para lograr este objetivo el simulador cambia la dirección del vector de gravedad a través del tiempo, lo que hace mucho más sencillos los cálculos y crea el efecto esperado en el sistema. Existen simulaciones en las que la posición del vector de gravedad se gira gradualmente en cada instante de tiempo a fin de obtener un efecto de rotación del sistema. En este caso a la hora de visualizar se desea observar el sistema considerando siempre la dirección del vector de gravedad en el sentido negativo del eje y. En cambio, existen otras simulaciones que alternan la posición del vector de gravedad entre el sentido positivo y el sentido negativo del eje y, a fin de obtener un efecto de agitación vertical del sistema. En este caso, a la hora de visualizar sólo se desea observar el efecto final en las partículas y no considerar el sentido del vector de gravedad fijo, ya que esto ocasionaría una rotación no deseada en la visualización.

En cuanto al servidor, éste está implementado dentro del mismo simulador como un hilo de ejecución paralelo (*thread*), el cual recibe peticiones a través de comandos, los cuales deben comenzar con el símbolo #, y envía como respuesta datos obtenidos de la simulación. Los datos se envían de la siguiente manera.

La simulación actualiza datos (posiciones de las partículas, velocidades de las partículas, gravedad, etc.) cada cierto intervalo de tiempo. La velocidad en que se actualizan los datos es llamada tasa de simulación. Estos datos son enviados por el servidor y contienen todo lo necesario para el *rendering* un *frame* de la animación. La velocidad en que se envían los datos de cada *frame* es llamada tasa de *snapshots*. Existen simulaciones muy rápidas (las variables cambian muy rápido) en que la tasa de simulación no es lo suficientemente alta (pasa mucho tiempo entre cada actualización de los datos) como para hacer observaciones útiles en la visualización. Y existen otros casos en que si bien la tasa de simulación es suficiente para realizar observación de los fenómenos, la tasa de *snapshots* es muy alta para poder apreciar estos fenómenos a la hora de reproducir la animación. Por lo anterior, el servidor acepta comandos capaces de controlar ambas tasas.

A continuación se listan los comandos aceptados por el servidor y que son de interés para el presente trabajo:

#init : Pide al servidor los datos que permanecen constante a lo largo de la simulación. Ej: cantidad de partículas, dimensiones de la caja que contiene al sistema de partículas, etc.

#start : Pide al servidor que comience el envío de datos de cada *frame*. Esto es, las posiciones de las partículas, el tiempo que lleva la simulación ejecutándose, etc.

#stop : Pide al servidor que detenga el envío de datos.

#snapshotrateincr : Aumenta la tasa de *snapshots* (*#snapshotratedecr* para disminuir).

#simrateon **:** Activa el control para aumentar o disminuir la tasa de simulación. (*#simrateoff* para desactivar).

#simrateincr **:** Aumenta la tasa de simulación (#simratedecr para disminuir). Se debe haber ejecutado *#simrateon* previamente.

2.3.3 Interacción entre SiMon y el servidor

Como se mencionó en la sección anterior, SiMon está implementado para recibir comandos y ejecutar una acción de respuesta usando la función cmd_execute(comando). Esta función lo único que hace es ver qué comando fue recibido y ejecutar una acción diferente de acuerdo a eso. Cuando el comando recibido es *connect*, SiMon se prepara para recibir datos del servidor y crea un nuevo hilo de ejecución paralela (*thread*) para ello. De esta manera SiMon trabaja con dos *threads*, uno principal para recibir comandos y otro para recibir los datos enviados por el servidor.

Por otro lado, el servidor implementa la misma estrategia. Usa un *thread* principal para recibir los comandos enviados por SiMon y ejecuta una acción de respuesta usando la función simon_cmd(comando). Cuando recibe el comando #init, que indica que debe comenzar el envío de datos al cliente, crea un nuevo *thread* para realizar esta acción. La muestra la interacción entre SiMon y el servidor, suponiendo que ya fue ejecutado un comando connect en SiMon y ya fue recibido un comando #init en el servidor.

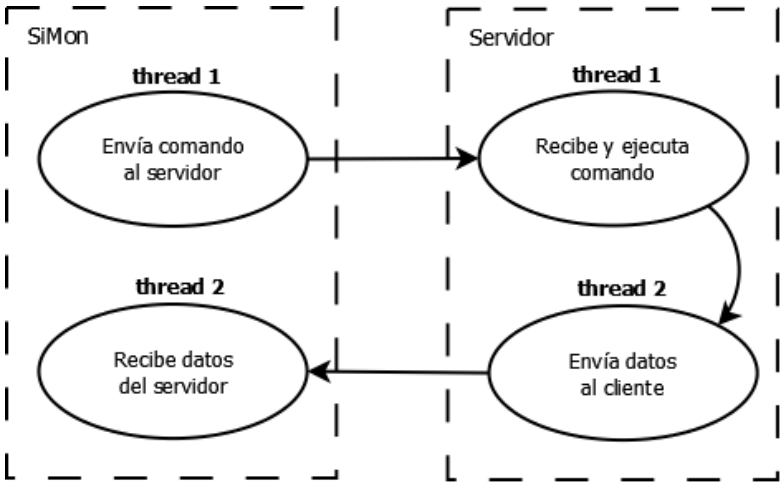


Figura 3 Comunicación entre SiMon y el servidor. SiMon utiliza un thread para enviar comandos al servidor y otro para recibir los datos del servidor. El servidor utiliza un thread para recibir los comandos de SiMon y otro para enviar la respuesta

2.4 Conceptos de ingeniería de software

2.4.1 Programación Orientada a Objetos

La Programación Orientada a Objetos es un paradigma de programación que define los programas en términos de "clases de objetos". Estos objetos son entidades que combinan estado (propiedades), comportamiento (métodos) e identidad (propiedad del objeto que lo diferencia del resto).

La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que interactúan entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

Dado que el diseño de aplicaciones basadas en este paradigma de programación es complejo, se propone abordar el problema mediante el uso de un conjunto de patrones para el diseño. Los patrones de diseño son un conjunto de soluciones genéricas reutilizables y adaptables a problemas que ocurren recurrentemente. En programación orientada a objetos, los patrones de diseño se expresan como plantillas de objetos e interacciones que resuelven un problema particular. Estas plantillas pueden ser aplicadas y reutilizadas en otros programas, con poca o nula adaptación.

2.4.2 Patrones de diseño

A continuación, se describen los patrones de diseño relevantes para la aplicación desarrollada en el proyecto:

Strategy: Define una familia de algoritmos, encapsulando cada uno y haciéndolos intercambiables. Strategy permite al algoritmo variar independientemente de los clientes que lo usan. Es útil cuando se tienen clases relacionadas que sólo difieren en su comportamiento, o se necesitan muchas variantes del mismo algoritmo. Permite además, evitar la exposición de estructuras complejas y ocultar los datos del algoritmo

Singleton: Asegura que una clase sólo posee una instancia y proporciona un punto de acceso global a ella

Iterator: Proporciona una forma de acceso a los elementos de un objeto de agregación secuencialmente sin exponer su representación. Permite proveer una interfaz uniforme para recorrer distintas estructuras agregadas (Apoya la iteración polimórfica).

Facade: Provee una interfaz unificada para un conjunto de interfaces pertenecientes a un subsistema. Se define una interfaz de alto nivel que interactúa con el subsistema de forma más fácil. Permite proporcionar una interfaz sencilla para subsistemas completos y así mantenerlos de forma independiente de los elementos que interactúan con él.

3 Análisis y diseño de la solución

3.1 Requerimientos

Para lograr los objetivos planteados anteriormente se tomaron los requerimientos que debía satisfacer el visualizador, los cuales consisten en permitir al usuario realizar las siguientes acciones:

1. Rotar la vista libremente utilizando el mouse o teclado.
2. Cambiar de vista ortogonal de manera sencilla mediante mouse o teclado.
3. Seleccionar de una lista de direcciones IP frecuentes el servidor al cual conectarse.
4. Agregar direcciones IP con un nombre asociado de manera que puedan ser seleccionadas fácilmente en próximas conexiones.
5. Seleccionar de una lista el puerto del servidor al cual el usuario desea conectarse. La lista debe contener todos los puertos del servidor en los cuales corren simulaciones al momento de la consulta.
6. Ingresar el puerto manualmente. Esto es útil en el caso que el servicio encargado de listar los puertos disponibles no esté ejecutándose en el servidor.
7. Alternar entre orientar la visualización considerando o no la dirección del vector de gravedad fija.
8. Cambiar parámetros de la simulación (ej: tasa de simulación, tasa de snapshots, etc.) mediante controles intuitivos.
9. Ser capaz de iniciar, detener y guardar una grabación de video de la animación. El formato de salida debe ser avi.
10. Tomar capturas de pantalla de la animación y guardar las imágenes en tamaños arbitrarios y en diferentes formatos (jpg, png, bmp, etc.).
11. Activar el coloreado de las partículas de acuerdo a su velocidad, variando entre los colores azul y rojo. Siendo azules las partículas más lentas y rojas las más rápidas.
12. Configurar el valor mínimo y máximo de velocidades considerados para el cálculo de los colores de cada partícula, en base a un histograma de velocidades.

3.2 Evaluación de alternativas tecnológicas

Para realizar la implementación se consideró evaluar diferentes alternativas de lenguajes de programación y librerías gráficas. Las alternativas propuestas se describen a continuación y se explica la razón de escoger una por sobre la otra.

3.2.1 Librería gráfica

Existen dos APIs que predominan el mercado gráfico: OpenGL [9] y DirectX [10]. Ambas proveen funcionalidades similares. Sin embargo, se ha descartado DirectX, dado que no es compatible con entornos Linux o MacOSX. Por lo tanto, la librería gráfica que se usará en el desarrollo de esta memoria será OpenGL, ya que uno de los objetivos es crear una herramienta portable a dichas plataformas.

3.2.2 Lenguaje de programación

En cuanto a los lenguajes de programación, se ha considerado utilizar Python o C++. En ambos lenguajes se encuentra disponible la librería gráfica OpenGL.

Dentro de las ventajas de usar Python se encuentra su alta expresividad. Con Python se pueden programar tareas complejas en pocas líneas de código, tomando mucho menos tiempo de programación, quedando más tiempo disponible para el *refactoring* y la optimización del código. Además, no presenta una curva de aprendizaje empinada, en términos de esfuerzo invertido versus tiempo y es fácil de depurar. También existe mucho material disponible en internet para trabajar con este lenguaje. Una de las principales razones por las que se consideró esta opción, es porque la alumna está familiarizada al uso de este lenguaje.

Por otro lado, la gran ventaja de C++ es su desempeño, siendo capaz de realizar tareas complejas en rangos de tiempo de ejecución muy cortos. Además, posee un buen control de manejo de memoria y de administración de recursos. Es robusto para sistemas complejos, pero difícil de depurar.

Se optó por usar ambos lenguajes de programación para distintos propósitos dentro del sistema. El visualizador se construirá principalmente en Python, pues la alumna ha trabajado antes con este lenguaje y esto permitirá realizar entregas rápidas que podrán testearse y optimizarse. Para cálculos complejos que demanden mayor uso de recursos se usará C++, ya que posee mejor desempeño. Esto es posible ya que Python permite hacer llamadas a extensiones escritas en C++ [11].

3.2.3 Lenguaje de shading

Una de las ventajas de las tarjetas gráficas modernas es que su GPU posee procesadores programables por el usuario mediante código escrito en lenguaje de *shading*. En este trabajo se explota esta característica, manipulando las propiedades de los vértices de OpenGL en la GPU, a

fin de aumentar la eficiencia de la aplicación. Dentro de los lenguajes de *shading* disponibles, se estudiaron las siguientes alternativas: GLSL, Cg y HLSL.

Dentro de estas alternativas se descartó HLSL, ya que dicho lenguaje de *shading* fue creado por Microsoft sólo para ser usado con la librería gráfica DirectX, la cual no es compatible con entornos Linux o MacOSX. Por lo que se evaluaron los lenguajes de *shading* Cg y GLSL.

3.2.3.1 GLSL [12]:

- Compatibilidad con múltiples plataformas incluyendo GNU/Linux, Mac OS X y Windows.
- Los *shaders* escritos en GLSL pueden ser usados en tarjetas gráficas de cualquier vendedor que soporte lenguaje de shading de OpenGL.
- No requiere Toolkit ni librerías adicionales.
- Cada vendedor de hardware incluye el compilador de GLSL en su dispositivo, lo que permite a cada vendedor crear código optimizado para la arquitectura de su tarjeta gráfica en particular.

3.2.3.2 Cg [13]:

- Es fácil para depurar errores.
- Es un lenguaje de *shading* creado primeramente para trabajar sobre tarjetas gráficas Nvidia.
- Permite portabilidad pero mediante la creación de perfiles que establecen capacidades mínimas y requisitos.
- Permite ser usado tanto con OpenGL como con DirectX.

Se optó por el lenguaje de *shading* GLSL, ya que éste fue creado por el mismo consorcio de industrias que gobierna la especificación OpenGL, librería gráfica escogida para el desarrollo de la aplicación. Esto entrega un enlace más natural con las variables de esta librería y menor diferencia de sintaxis [14].

3.3 Ambiente de desarrollo

La aplicación se desarrolló utilizando el sistema operativo Ubuntu Linux 10.04 y el entorno de desarrollo integrado (IDE) Eclipse, extendido con el plugin PyDev para trabajar con lenguaje Python. Además se utilizó Virtualenv, herramienta para crear un entorno independiente en Python, a fin de evitar problemas de versiones con otras aplicaciones que utilicen módulos de Python.

Como se mencionó en la sección 4.1.1, OpenGL fue la librería gráfica escogida para realizar el rendering de las partículas y GLSL el lenguaje de shading escogido para realizar las

optimizaciones utilizando la GPU. Además, se utilizaron otras librerías que facilitaron el desarrollo, a continuación se describen las más relevantes:

3.3.1 Qt

Qt es una librería que facilita el desarrollo de aplicaciones con interfaz gráfica de usuario (GUI), y también aplicaciones sin interfaz gráfica como herramientas de línea de comando y consola para servidores. Es en la primera área que ha sido ampliamente destacada, siendo utilizada en aplicaciones de escritorio de renombre como: KDE, Skype, Google Earth, VLC, entre otros [15].

Qt es distribuida bajo los términos de GNU Lesser General Public License, es software libre y de código abierto [16]. Qt utiliza el lenguaje de programación C++ de forma nativa. Funciona en todas las principales plataformas, y posee mucho material de documentación y apoyo, el cual puede ser encontrado fácilmente en la Web.

Además Qt cuenta con un editor gráfico, Qt Designer. El cual facilita la creación de interfaces, ya que ofrece edición de manera "what-you-see-is-what-you-get" (WYSIWYG).

Para el desarrollo de esta aplicación se utilizaron las características de Qt que permiten la creación de ventanas, creación y despliegue de menúes, interacción usando cuadros de diálogos, recepción de señales de teclado, manejo de hilos de ejecución, recepción de inputs del usuario mediante botones y cuadros de texto, entre otras.

3.3.2 QGLViewer

QGLViewer es una librería escrita en C++ basada en Qt que facilita la creación de visualizadores gráficos. Es una herramienta multiplataforma pudiendo ser compilada en ambientes Unix-Linux, Mac OS y Windows.

Esta librería provee algunas de las principales características de un visualizador 3D, lo cual permite al desarrollador enfocarse en qué será dibujado, en vez de cómo el software debe dibujar. Está diseñada de tal manera que es fácil de extender y personalizar. Dentro de las funcionalidades que ofrece, a continuación se mencionan las que fueron de interés para el trabajo desarrollado en esta memoria:

- Desplazamiento a través de la escena mediante la implementación de una cámara en dos modos: vuelo (semejante a la cámara de un videojuego de pilotaje de avión) y exploración (rotación libre de la escena).
- Definición de un sistema de coordenadas intuitivo.
- Toma de screenshots de la escena en tamaños arbitrarios y en distintos formatos de archivo (.png, .jpg, .ps, etc.)
- Personalización de accesos directos del teclado y comportamiento del mouse.

Cabe destacar que al ser Python el lenguaje escogido para el desarrollo de este trabajo, para utilizar las dos librerías recién mencionadas fue necesario utilizar sus bindings para Python. En el campo de la programación, un binding es una adaptación de una biblioteca para ser usada en un lenguaje de programación distinto de aquél en el que ha sido escrita. En este caso se utilizaron los bindings PyQt y PyQGLViewer respectivamente.

3.3.3 Otras librerías

NumPy: Extensión de Python que entrega mayor soporte de vectores y matrices. Incorpora un nuevo contenedor, un objeto array multidimensional. Ofrece una biblioteca muy eficiente de funciones matemáticas, ampliando las capacidades de Python al permitir hacer operaciones rápidas sobre los arrays, además de reformatearlos, hacer estadística básica, etc.

En este trabajo se utilizó NumPy para almacenar información relevante de las partículas, tales como posición, velocidad, color, etc. Utilizar esta librería permitió realizar cálculos de manera rápida y eficiente sobre los datos.

Matplotlib: Librería que facilita la generación de gráficos de forma muy rápida a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy. En este trabajo se utilizó Matplotlib para desplegar un histograma de barras simples para la observación de propiedades de las partículas (en este caso las velocidades) y su frecuencia en el sistema.

3.4 Arquitectura de la aplicación

3.4.1 Módulos del sistema completo

Dentro de la arquitectura del sistema se distinguen tres elementos principales:

a) El cliente SiMon encargado de enviar peticiones al servidor donde se ejecutan las simulaciones y de recibir la información y datos del sistema de partículas que posteriormente debe ser graficado. La descripción de esta componente ya ha sido detallada en la sección 2.5.1.
b) El servidor en que se ejecutan las distintas simulaciones y donde además se levanta el servicio encargado de listarlas. Se encarga de recibir las peticiones del cliente a través de comandos y de enviar los datos pedidos por éste. En la sección 2.5.2 se presenta una descripción más detallada.
c) El visualizador responsable de interactuar con el usuario a través de una interfaz gráfica de usuario y de graficar el sistema de partículas. Para lograr este propósito previamente debe comunicarse con el cliente SiMon y obtener los datos necesarios de la simulación que será visualizada.

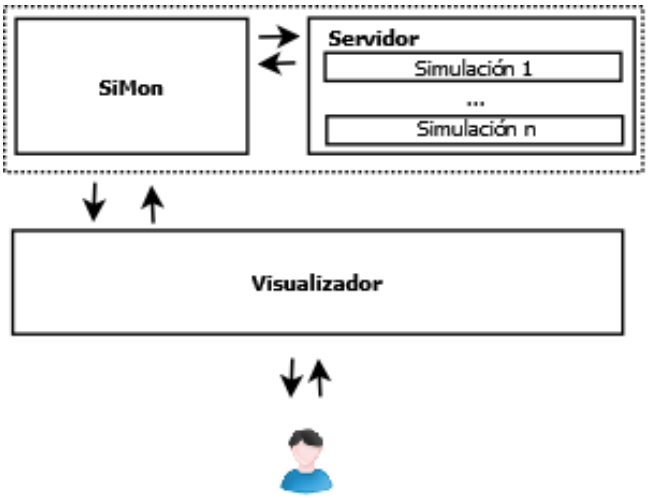


Figura 4 Arquitectura del sistema completo.

3.4.2 Módulos del visualizador

A continuación se describen las componentes principales que estructuran la aplicación, junto con detallar la motivación y principios detrás de las decisiones de diseño.

3.4.2.1 Interfaz de usuario

Esta sección del sistema es la encargada de la interacción con el usuario. Presenta la interfaz de visualización y recibe el input del usuario. Además, esta sección se comunica con el motor gráfico desplegando las imágenes generadas por éste. A continuación se describen las clases más importantes de este módulo.

ApplicationMainWindow: Esta clase es la encargada de crear la interfaz de usuario. Además, crea un objeto **Viewer**, que consiste en una ventana de visualización que está contenida en la interfaz.

Viewer: Esta clase es la responsable de crear la ventana donde se dibuja el sistema de partículas. Dicha ventana coexiste con los otros elementos de interfaz, de los cuales recibe señales originadas por las acciones del usuario y responde ante éstas. Por ejemplo, un elemento de interfaz podría ser un checkbox para activar la visualización de los ejes de coordenadas. Cuando el usuario selecciona el checkbox se emite una señal que es captada por **Viewer**, el cual delega al motor gráfico la tarea y espera una respuesta que es presentada al usuario.

Se utilizó el patrón de diseño facade, ya que **Viewer** define una interfaz de alto nivel que interactúa con el subsistema encargado de graficar y con el encargado de entrada/salida de la aplicación, los cuales poseen varias componentes para lograr su objetivo.

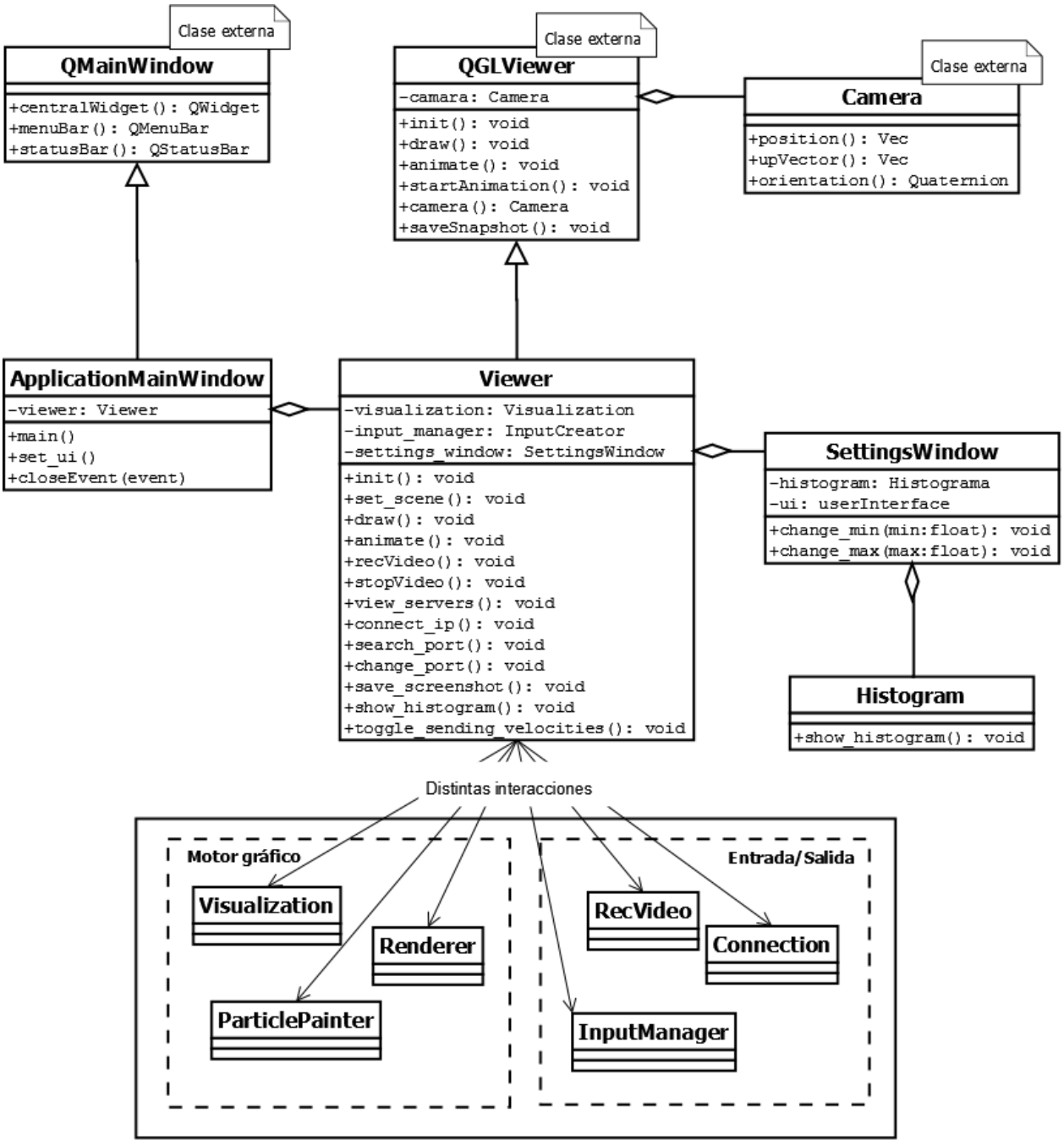


Figura 5 Diagrama de clases que componen la interfaz de usuario

3.4.2.2 Motor gráfico

El motor gráfico del sistema es el encargado de controlar, gestionar y actualizar la animación presentada en la ventana de visualización.

Visualization : Esta clase es la responsable de gestionar los elementos de la escena. Dentro de sus tareas está la de crear una instancia de **System**, que consiste en el sistema a graficar. Además, define los parámetros de visualización, tales como, ángulo de inclinación del sistema de acuerdo a la gravedad, tamaño de la caja contenedora del sistema, orientación del sistema, activación de colores, etc.

System : Esta clase representa un sistema de partículas. Almacena los datos del sistema tales como posición, velocidad y tipo de las partículas. Para efecto de este trabajo sólo se consideran partículas esféricas. Sin embargo, considerando que en el futuro pueden ser necesarios elementos de otras formas se utilizó herencia, de manera que para otros tipos de partículas se extiende esta clase. En este caso, la clase **SphericSystem** extiende a **System** representando un sistema de partículas con forma esférica.

Renderer : Esta clase, tal como lo dice su nombre, es utilizada por **System** para realizar el rendering de la escena.

ParticlePainter: Esta clase es utilizada para calcular el color de cada partícula. En simulaciones físicas el color puede representar diversos elementos, como por ejemplo, la velocidad a la que se desplazan las partículas dentro del sistema, el tipo de material de la partícula, etc., utilizando un algoritmo diferente para cada situación. Uno de los objetivos de la aplicación es que sea posible distinguir las partículas según su color dadas diferentes propiedades de interés. Debido a esto se utilizó el patrón Strategy, ya que a priori no se conoce el algoritmo utilizado para calcular el color. Este patrón de diseño, permite elegir el algoritmo de forma dinámica. En este caso, **VelocitiesPainter** extiende a **ParticlesPainter** y calcula el color según la velocidad de la partícula.

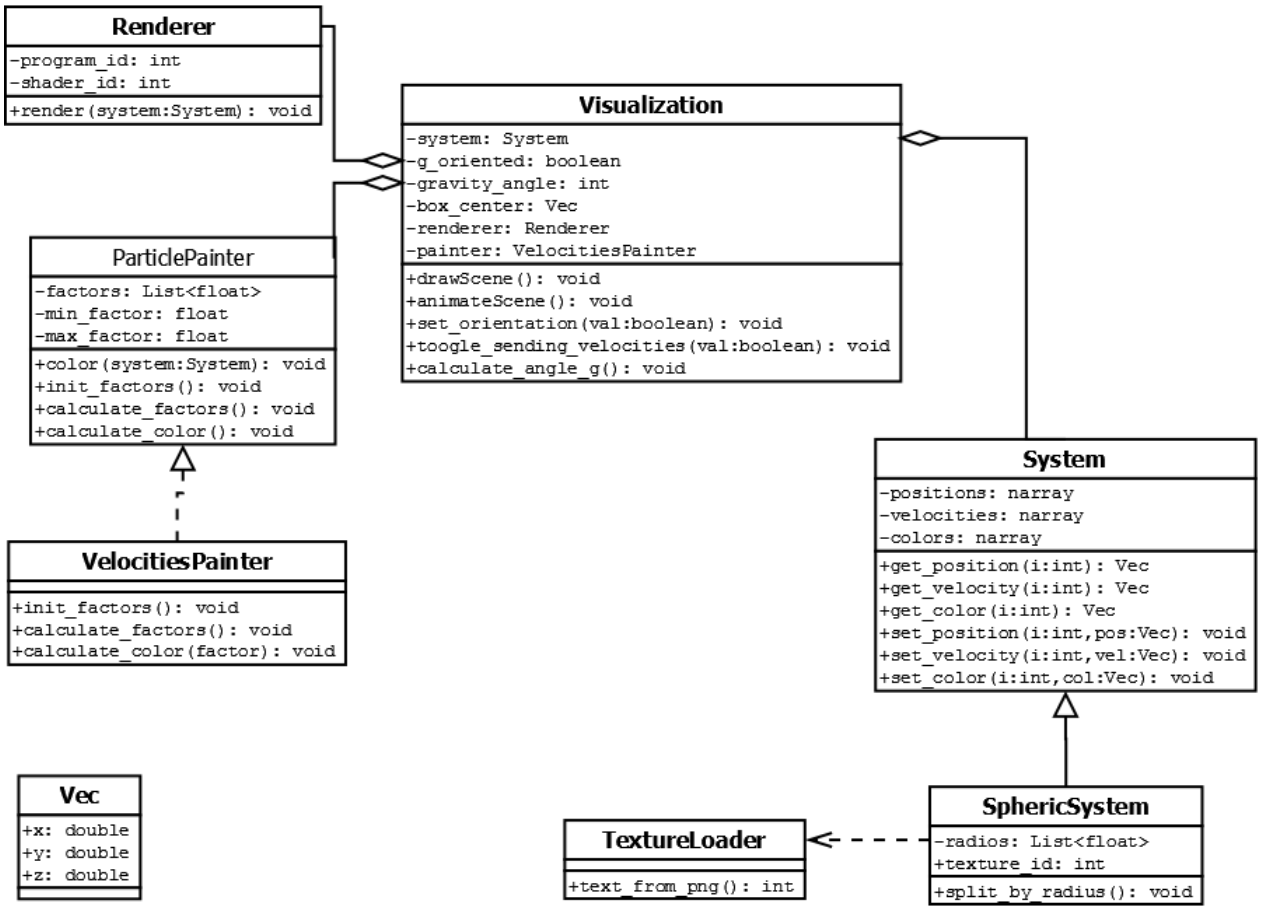


Figura 6 Diagrama de clases que componen el motor gráfico

3.4.2.3 Entrada y Salida

En este módulo del sistema se encuentran los componentes de software que se encargan de recibir los datos de entrada que posteriormente serán graficados y de generar los datos de salida tales como grabación de video, captura de imágenes, etc.

InputManager: Esta clase es la encargada de determinar la forma en que se reciben los datos que se usarán para graficar el sistema. Para ello crea un objeto de clase Input del subtipo adecuado. Dado que la aplicación no necesita más de una forma de hacer este trabajo, se decidió usar el patrón de diseño Singleton. Este patrón garantiza que exista una única instancia de esta clase.

Input: Esta clase implementa la forma en que se reciben los datos de entrada. Con el fin de garantizar la extensibilidad de la aplicación, se decidió ocupar el patrón de diseño Strategy. Este patrón permite escoger la forma en que se recibirán los datos de forma dinámica, ya que

pueden existir diferentes maneras de obtener los datos, como por ejemplo, usando un archivo, desde un servidor o a través del pipe de procesos. Dentro del alcance de este trabajo sólo se contempla la recepción de datos enviados por un servidor.

InputFromServer: Esta clase extiende a Input e implementa la recepción de datos enviados desde un servidor externo.

Connection: Esta clase es la encargada de realizar y administrar la conexión con el servidor. Está asociada a la clase **InputFromServer**.

RecVideo: Esta clase se encarga de generar grabaciones de video de la animación. Es usada por Viewer.

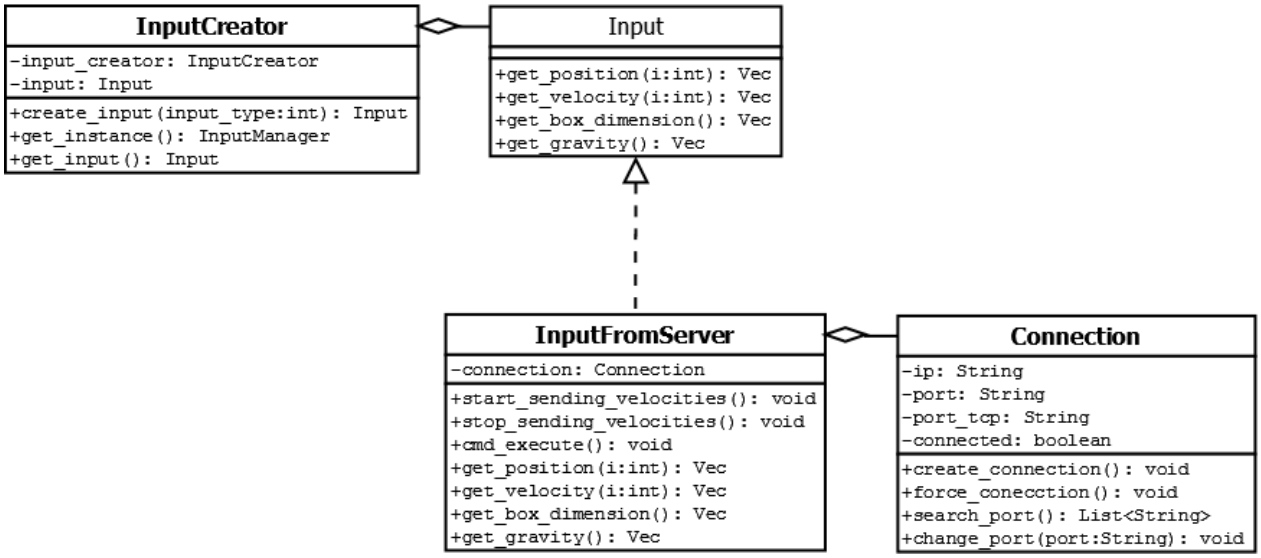


Figura 7 Diagrama de clases que componen la entrada del programa

4 Implementación

En este capítulo se describe el proceso llevado a cabo para implementar la aplicación diseñada. Se comienza por la elección de las herramientas y ambiente de desarrollo. Luego se describen las etapas en el desarrollo de la aplicación, donde se implementan las funcionalidades descritas anteriormente en la sección de requerimientos 3.1.

4.1 Conectar SiMon con Python

De acuerdo al diseño descrito en la sección 3.2, es necesario que el visualizador obtenga los datos de la simulación usando el cliente SiMon. Sin embargo, SiMon fue desarrollado en lenguaje C, y como se detalló en la sección 4.1.2 se decidió usar el lenguaje de programación Python para el desarrollo del visualizador. Debido a esto, se consideraron tres alternativas para resolver este problema:

a) Reescribir SiMon en Python: Dada la complejidad de SiMon, esta alternativa fue descartada, ya que le hubiera restado bastante tiempo a la implementación del visualizador en sí.

b) Implementar el visualizador en C: Dado al análisis realizado en la sección 4.1.2 del presente informe, se inclinó por aprovechar las ventajas de Python para el desarrollo del visualizador, por lo que esta alternativa también se descartó.

c) Usar SiMon como una librería: En esta alternativa se plantea la posibilidad de hacer llamadas a las funciones de SiMon escritas en C, desde código escrito en Python, es decir, usar SiMon como una librería externa. Dentro de las herramientas disponibles para lograr este objetivo, se encuentra SWIG (Simplified Wrapper and Interface Generator/ Generador simplificado de Interfaces y envoltorios) [17], software que permite integrar código escrito en C/C++ con distintos lenguajes de script, entre ellos Python. Esta alternativa fue la escogida pues aprovecha el trabajo previo sin restar esfuerzos a la implementación del visualizador, permitiendo que el enfoque se centre en el desarrollo rápido de la aplicación. La Figura 8 muestra la arquitectura de la solución propuesta usando SWIG.

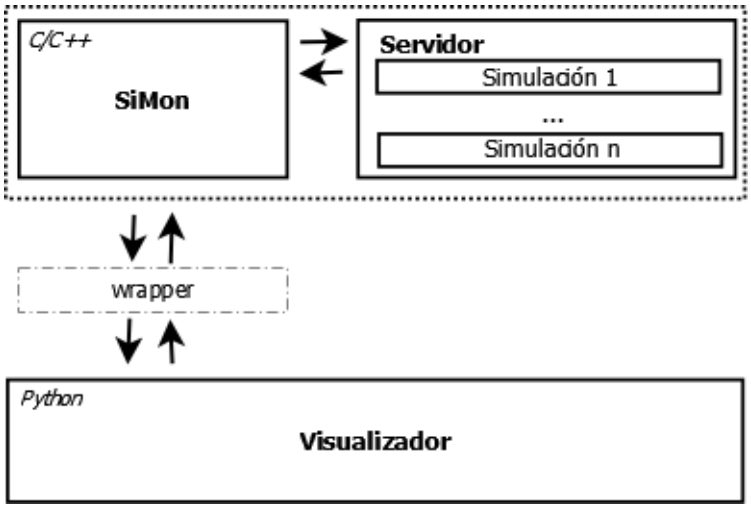


Figura 8 Arquitectura de la aplicación usando SWIG.

SWIG crea un wrapper (o envoltorio) de las interfaces que se desean tener disponibles en el lenguaje objetivo. La función del wrapper es básicamente convertir valores desde Python a una representación de bajo nivel con la cual pueda trabajar C y convertir los resultados desde C de vuelta a Python [18].

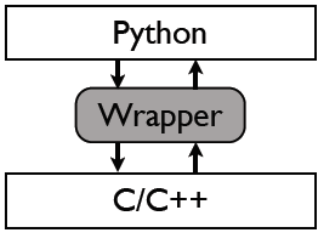


Figura 9 Extensión de Python a través de un wrapper

Para ello se debe especificar en un fichero lo que se desea exportar (funciones, estructuras, etc.), el cual debe tener la extensión .i, por convenio. Este fichero, llamado archivo de interfaces, contiene además las directivas para SWIG.

De SiMon se exportaron las siguientes funciones y variables globales que fueron necesarias en el desarrollo de la aplicación:

4.1.1 Funciones

- **void create_socket (char *ip, int puerto):** Crea un socket para recibir los mensajes del servidor.

- **float getposx(int i) :** Retorna la coordenada x de la posición de la partícula i del sistema. De manera análoga para las coordenadas y y z.

- **float getvelx(int particle):** Retorna la coordenada x de la velocidad de la partícula i del sistema. De manera análoga para las coordenadas y y z.

- **void cmd_execute(char *cmd):** Ejecuta un comando de Simon. Existen varios comandos pertimidos, pero sólo dos son de interés para este trabajo:

4.1.2 Variables globales

- **int ndiscos :** Almacena la cantidad de partículas que posee el sistema.

- **float LargoX :** Almacena el largo de la caja que contiene al sistema de partículas. LargoY para el alto y LargoZ para el ancho.

SWIG es un programa de línea de comando. Una vez preparado el archivo de interfaces (.i) SWIG lo usa como entrada para generar el wrapper para Python. Esto se hace a través de la siguiente instrucción:

swig -python simon.i

La opción -python indica que el lenguaje objetivo es Python. Adicionalmente se puede añadir la opción -globals [nombre] para asignar un nombre al objeto en Python que contendrá las variables globales. Si se omite se usa el nombre cvar para acceder a las variables.

Luego de ejecutar esta instrucción SWIG genera dos archivos simon_wrap.c y simon.py. El archivo _wrap.c es el código en C que debe ser compilado en una librería dinámica. El archivo .py es el código de apoyo que sirve como front-end para el módulo C de bajo nivel. El usuario debe importar el archivo .py.

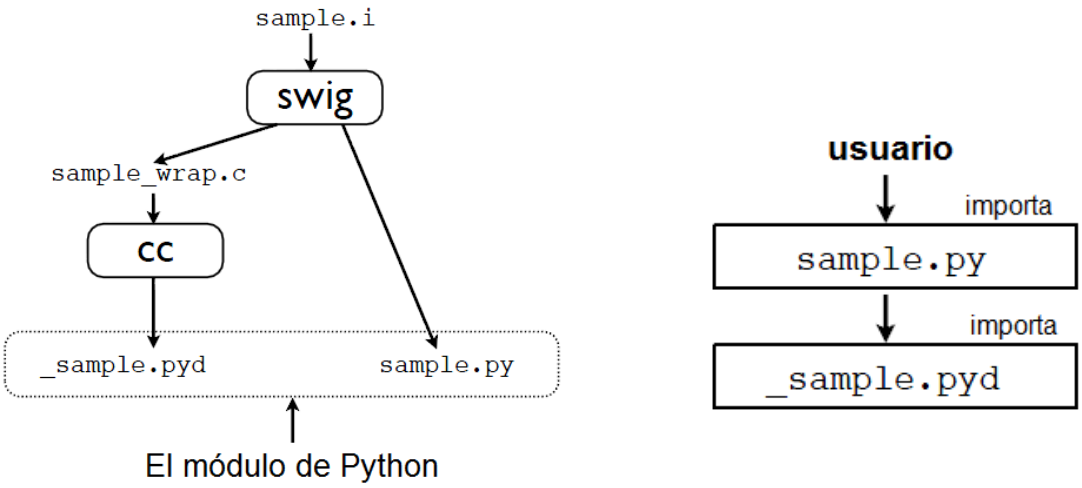


Figura 10 (Izq.) Construcción de módulos con SWIG. (Der.) Arquitectura dual en módulos generados por SWIG

Para usar el módulo basta con importarlo usando la instrucción `import simon`. Para que el intérprete de Python pueda encontrar el módulo, su ubicación debe estar incluida en la variable `path` del módulo `sys` de Python. Una vez importado las funciones pueden ser utilizadas de manera usual: simon.nombre_funcion(parámetros).

Para proveer acceso a las variables globales en C, SWIG crea un objeto especial en Python llamado "cvar" el cual es añadido en cada módulo generado. Para acceder a una variable basta con añadir a su nombre el prefijo "cvar.". Como ya se mencionó, es posible cambiar el nombre de este objeto especial añadiendo la opción `-globals [nombre]` al momento de generar el wrapper. Esto es útil cuando se importa más de un módulo generado por SWIG, pues se produce un choque de nombres y sólo se cargan las variables del último módulo importado.

4.2 Implementación de funcionalidades

Una vez creado el módulo de SiMon para ser usado en Python, el siguiente paso fue comenzar con la implementación de las funcionalidades pedidas.

4.2.1 Interfaz de usuario

Como se detalló en la sección 4.2.1 se utilizó Qt para crear la interfaz gráfica de usuario (GUI). Una de las características principales de Qt es su mecanismo para reconocer las acciones del usuario en la interfaz y reaccionar ante éstas. Cada elemento de la interfaz se conecta con un método que ejecuta una funcionalidad. De esta manera cada vez que el usuario realiza una acción, ya sea de teclado o mouse, se envía una señal al método encargado de ejecutar la acción. Del mismo modo, si la interfaz espera una respuesta tras realizar la acción, es posible enviar señales

de vuelta para indicar que la acción se completó, junto con los parámetros necesarios para actualizar la interfaz.

En Python la conexión de señales y métodos se realiza mediante la siguiente instrucción:

sender.signal.connect(receiver.slot)

Donde sender es el objeto que emite la señal, por ejemplo un elemento de la interfaz; signal es el nombre de la señal; receiver es el objeto que recibe la señal y slot es el método encargado de ejecutar la acción.

Para crear los elementos de interfaz se utilizó el editor gráfico Qt Designer, el cual genera un archivo de salida de extensión .ui que posee la especificación de la interfaz. Este archivo puede ser usado en objetos de tipo **QMainWindow** para cargar de manera rápida la interfaz de usuario en una sola instrucción. Esto genera un código mucho más limpio y un sistema que desacopla los elementos gráficos de la lógica.

En la aplicación desarrollada se implementó la clase **ApplicationMainWindow** que extiende a **QMainWindow**, perteneciente a la librería externa Qt, y utiliza el fichero viewerInterface.ui para cargar la interfaz gráfica. Además, crea una instancia de **Viewer**, que posee todos los métodos que se conectan con las señales emitidas por las acciones del usuario.

La interfaz que utiliza la aplicación posee tres componentes principales: un menú de acciones, una ventana de visualización y un panel de control en la parte inferior. El menú de acciones se ubica en la parte superior de la ventana principal, como muestra la Figura 11. Las principales acciones de este menú tienen relación con la conexión al servidor. La ventana de visualización se encuentra al centro de la ventana principal y en ella se despliega la animación del sistema de partículas. Finalmente, en la parte inferior se encuentra el panel de control que posee los elementos que controlan la animación desplegada. Siguiendo el principio de consistencia funcional en diseño de interfaces gráficas [19], se agruparon los elementos con funcionalidades similares como se indica en la Figura 12.

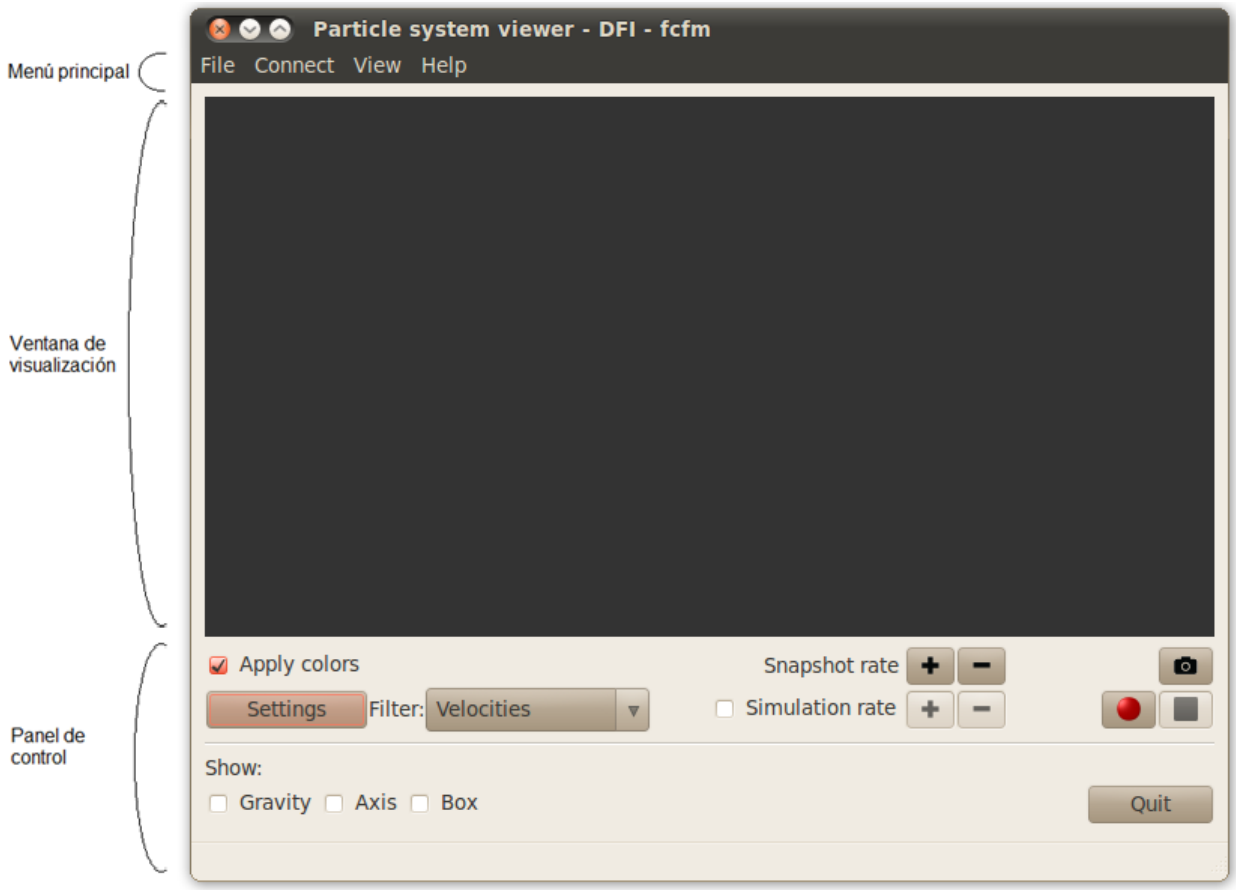


Figura 11 Ventana principal de la aplicación.

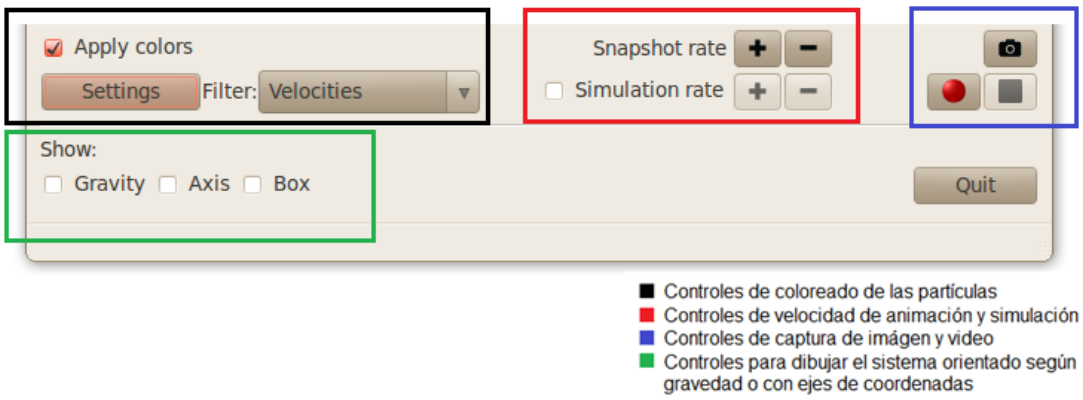


Figura 12 Panel de control de la aplicación.

4.2.2 Recepción de datos

Las clases que participan en la recepción de datos son **Viewer**, **InputManager**, **InputFromServer** y **Connection**. **Viewer** recibe las señales de la interfaz cuando un usuario quiere conectarse a un servidor para empezar a graficar. **InputManager** crea un objeto de tipo Input adecuado dependiendo de la forma en que se deben recibir los datos de entrada. Dentro del alcance de este trabajo sólo se contempla la recepción de datos enviados por un servidor, por lo tanto **InputManager** creará un objeto de tipo **InputFromServer**. Este objeto es el encargado de pedir los datos al servidor usando el módulo de SiMon creado con SWIG como interfaz. **Connection** se encarga de crear una nueva conexión entre SiMon y el servidor.

Para lograr la comunicación entre las tres componentes del sistema: el servidor, SiMon y el visualizador, se sigue el siguiente orden de acciones:

1. El usuario selecciona del menú principal la acción "Connect to server…"->"View history" dentro de la pestaña "Connect".

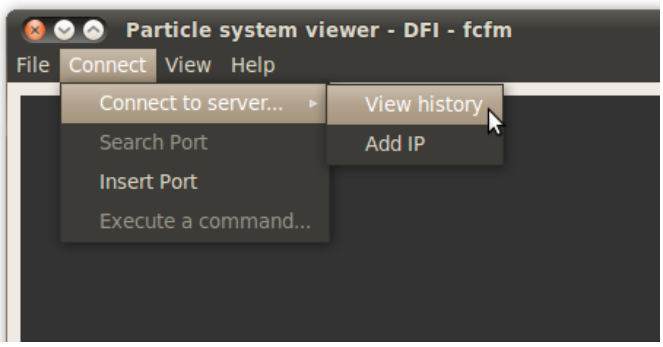


Figura 13 Menú para desplegar lista de servidores guardada.

Esto envía una señal a **Viewer** para ejecutar el método view_servers. Este método lo único que hace es leer de un archivo los nombres y direcciones IP de servidores disponibles para intentar una conexión y desplegar la información en una ventana nueva, tal como se muestra en la Figura 14. Estos datos han sido guardados previamente por el usuario.

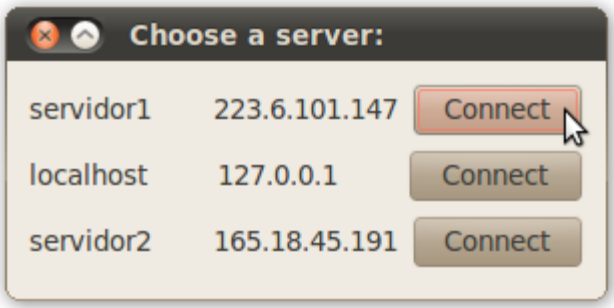


Figura 14 Ventana para escoger un servidor al cual conectarse

Para guardar los datos de nuevos servidores el usuario utiliza la acción "Add IP" desde la misma pestaña. Esto despliega una ventana nueva que consta sólo de dos campos de texto para recibir el nombre del servidor y la dirección IP. Una vez que el usuario ingresa esta información, se emite una señal para que se ejecute un método de **Viewer** que guarda los datos en el archivo de servidores e intenta establecer una conexión.

2. Cuando el usuario escoge un servidor al cual conectarse, nuevamente se emite una señal a **Viewer**, esta vez para que ejecute el método connect_ip(ip). Este método usará la instancia de **InputManager** para crear un objeto de tipo **InputFromServer**, el cual usa a **Connection** para conectarse con la ip dada.

3. Una vez establecida la conexión con el servidor, el usuario puede usar la acción "Search port" para listar los puertos en los que corren simulaciones al momento de la consulta. Al hacer click en "Search port" se emite una señal a **Viewer**, para que ejecute el método search_port(). Lo que hace este método es enviar al servidor el comando "#list" a través de SiMon, quien recibe la respuesta y la guarda en variables globales destinadas para ello. El visualizador lee estas variables y despliega una ventana nueva con todos los puertos disponibles y la opción de conectarse a alguno, tal como se muestra en la Figura 15.

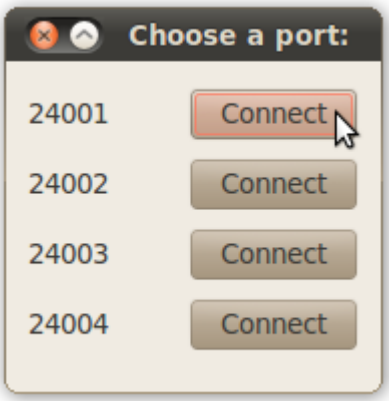


Figura 15 Ventana para escoger el puerto del servidor de la simulación que se desea visualizar.

En ocasiones, puede suceder que en el servidor no esté ejecutándose el servicio que lista los puertos. En ese caso, el usuario puede ingresar manualmente el puerto al cual desea conectarse mediante la acción "Insert port". Esto despliega una ventana que recibe el puerto del usuario y ejecuta un método de **Viewer** que intenta la conexión directamente.

4. Finalmente el usuario escoge un puerto al cual conectarse, **Viewer** recibe la señal y ejecuta el método encargado de la conexión a un puerto. Una vez establecida la conexión el motor gráfico usa la única instancia de **InputFromServer** para pedir los datos de la simulación cuando los necesita. Esta clase utiliza las funciones del módulo de SiMon creado con SWIG para hacer peticiones al servidor.

Una vez establecida la comunicación entre las componentes del sistema, el visualizador pide los datos para graficar a través de SiMon al servidor. Sin embargo, la implementación de estos dos últimos no contemplaba el intercambio de algunos datos necesarios para realizar la animación, como por ejemplo, el tipo de partícula, los radios, las velocidades, etc. Por esto fue necesario modificar ambos para lograr enviar y recibir algunas de estas variables.

Las modificaciones hechas a Simon son las siguientes:

- Se crearon variables globales que almacenan los datos nuevos.
- Se modificó el *thread* que recibe los datos del servidor para que considere el caso de recibir las variables nuevas y se encargue de almacenarlas en sus respectivas variables globales.

Las modificaciones hechas al servidor son las siguientes:

- Se agregó la validación de un comando nuevo que indica que una nueva variable debe ser enviada. Lo cual consiste en agregar un nuevo caso en la función del servidor simon_cmd(comando).

- Se modificó el *thread* que envía los datos para que envíe las nuevas variables.

Como se explicó en la sección 2.5.3 SiMon recibe los datos en un *thread* diferente del que recibe y ejecuta los comandos. Esto ocasionó el siguiente problema: el visualizador ejecutaba la función cmd_execute(comando) y ésta retornaba antes que las variables globales fueran escritas en el *thread* que recibe los datos. La ejecución del visualizador continuaba e intentaba leer una variable global que aún no había sido escrita, entonces ocurría un error. Para solucionar este problema se implementó en SiMon un mecanismo de sincronización de threads. Se agregó una condición que la función cmd_execute debía esperar que se cumpliera para continuar su ejecución. Esta condición es cambiada en el *thread* que recibe los datos, de manera que cuando las variables globales ya fueron escritas, la condición cambia a verdadera y cmd_execute puede retornar. Esto se implementó usando *mutex* y variables de condición.

4.2.3 Despliegue de la animación

La clase **Viewer** representa a la ventana de visualización donde se presenta al usuario la animación. Esta clase extiende a **QGLViewer**, que pertenece a la librería externa del mismo nombre. Para dibujar la escena en la ventana de visualización, **Viewer** crea una instancia de **Visualization**, clase perteneciente al motor gráfico que posee todos los parámetros y elementos de la escena. Para dibujar la escena se reimplementó en **Viewer** dos métodos de **QGLViewer**, los cuales son el centro de la librería externa: draw() y animate().

draw(): Es el método encargado de dibujar la escena. En este método se deben hacer las llamadas a las instrucciones de OpenGL que grafican los datos del sistema de partículas. Una llamada a draw() dibujará un frame de la animación. En la aplicación draw() llama al método de **Visualization** encargado de dibujar la escena. **Visualization** posee un objeto de la clase **Renderer** a quien le entrega los datos del sistema y delega la responsabilidad de pintar la escena.

animate(): Este método se encarga de actualizar los datos que se dibujan en draw(). Indica cómo la escena evoluciona a través del tiempo. Sólo tiene sentido cuando se ha llamado a startAnimation() previamente, ya que eso indica que se trata de una animación y que los datos deben actualizarse de manera periódica y recalcular draw().

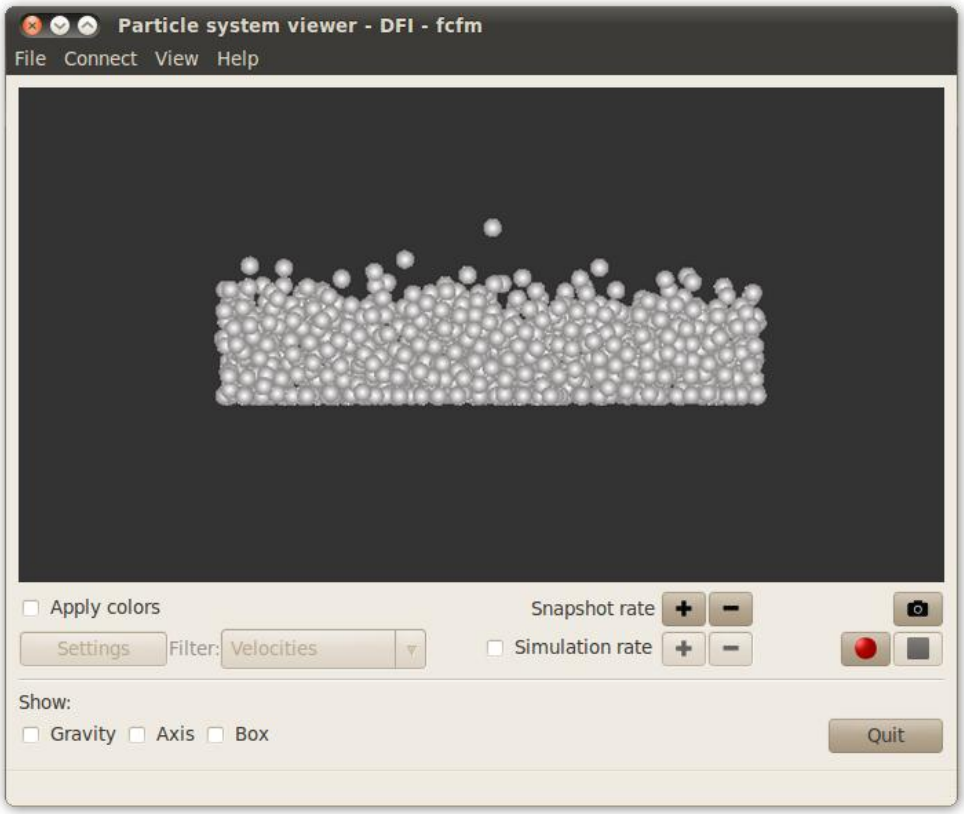


Figura 16 Visualizador desplegando una animación

4.2.4 Control de la cámara

QGLViewer posee una instancia a un objeto de tipo Camera, también de la librería externa, al cual se accede a través del método camera(). Camera, como su nombre lo indica, representa la cámara asociada a la escena, y posee métodos útiles que permiten rotar libremente la escena, hacer zoom, cambiar su orientación, etc. Como **Viewer** extiende a **QGLViewer**, hereda también todos sus elementos asociados. Por lo que usar esta librería externa facilitó notablemente la implementación de los requerimientos relacionados con el control de la cámara. **QGLViewer** implementa:

- Rotación de la vista libremente utilizando el mouse.
- Cambio a la vista ortogonal más cercana usando doble click de mouse.
- Tomar capturas de pantalla de la animación usando Ctrl + S. Al guardar se despliega una ventana para escoger el tamaño y el formato de la imagen, como lo indica la Figura 17.

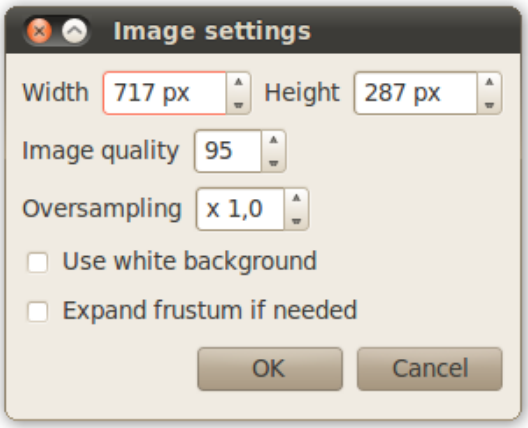


Figura 17 Ventana para configurar una captura de imagen de la animación.

4.2.5 Grabación de videos

Para grabar un video el usuario debe hacer click en el icono de grabación en la interfaz. En ese momento se emite una señal a **Viewer** para ejecutar el método rec_video(). Este método realiza las siguientes acciones:

- Envía una señal a la interfaz para indicar que la grabación comenzó y que se debe deshabilitar el botón 'Rec' y habilitar el botón 'Stop'. Además de añadir alguna señal que indique que se está llevando a cabo una grabación. En este caso se decidió añadir un elemento llamativo que consiste en un icono intermitente que indica que la grabación está activa.
- Conecta la señal drawFinished con el método saveSnapshot de **QGLViewer**. Esto hace que cada vez que se termina de dibujar un *frame*, se emite la señal drawFinished que ejecuta la acción de guardar una imagen del *frame*. Las imágenes de cada *frame* se guardan en el directorio temporal /tmp/snapshots/ creado al principio de la ejecución del programa.

Cuando el usuario detiene la grabación del video se emite una señal a **Viewer** para ejecutar el método stop_video(). Este método realiza las siguientes acciones:

- Envía una señal a la interfaz para indicar que la grabación terminó y que se debe habilitar el botón 'Rec' y deshabilitar el botón de 'Stop'. Además de quitar la señal que indica que se está llevando a cabo una grabación.
- Desconecta la señal drawFinished del el método saveSnapshot de **QGLViewer**.
- Despliega una nueva ventana para que el usuario seleccione el nombre, ubicación y formato del video generado.

- Crea una instancia de **RecVideo** y ejecuta su método principal en un nuevo *thread*. **RecVideo** extiende **QThread**, que es una clase de Qt para el manejo de *threads*. La función de **RecVideo** es generar un video en base a las imágenes guardadas de cada *frame* durante la grabación.
- Una vez generado el video, elimina las imágenes guardadas de cada *frame*.

4.2.6 Distinción de partículas

Uno de los objetivos de este trabajo es permitir la distinción de las partículas de acuerdo a propiedades que se deseen estudiar. Dentro del alcance de este trabajo se contempla la distinción de las partículas por su radio, por su velocidad y tipo. El radio, naturalmente, se distingue de acuerdo al tamaño de la partícula dibujada. Se decidió distinguir el tipo de la partícula de acuerdo al color de ésta, permitiendo diferenciar de manera rápida una partícula de un tipo de otra partícula de otro tipo, debido a que poseen distinto color. La velocidad se decidió distinguir por la intensidad de color, por ejemplo una partícula de un tipo dado puede variar su color desde azul hasta rojo, pasando gradualmente de un color a otro, donde azul indica baja velocidad y rojo indica alta velocidad.

4.2.6.1 Radios

Como se detalló en la sección 2.5.1 la comunicación entre SiMon y el servidor está implementada usando el protocolo UDP, el cual es rápido pero no otorga garantías para la entrega de sus mensajes. Sin embargo, los radios son una variable requerida para comenzar con la visualización, y debe estar garantizada la recepción íntegra de este dato al inicio del programa. La solución para recibir los radios de manera confiable es usar protocolo TCP en el envío. Por otra parte SiMon para lograr identificar el puerto al que desea conectarse debe comunicarse primero con el servicio encargado de listar los puertos donde corren las simulaciones, y esto lo hace usando protocolo UDP. Por lo que la solución debe permitir alternar entre ambos protocolos. Primero utilizar UDP para identificar el puerto donde corre la simulación, después cambiar a TCP para recibir los radios y finalmente cambiar a UDP para recibir las posiciones de las partículas. A continuación se detallan las modificaciones que fueron necesarias para lograr este objetivo.

En SiMon:

- Se agregó la función create_socket_tcp(ip,port) que crea un socket TCP, se conecta a la dirección IP y puerto dados como argumentos, y crea un thread para recibir los datos usando el socket creado.

- Se agregó el comando connect_tcp[port] para conectarse al puerto dado como argumento usando protocolo TCP.

- Se modificó el fichero de interfaces usado por SWIG para incluir las funciones y variables agregadas, así quedan disponibles para ser usadas por el visualizador.

En el servidor:

- Se creó un thread usado por el servidor para esperar por conexiones TCP.
- Cuando un cliente se conecta usando protocolo TCP, el servidor envía los radios de las partículas.

4.2.6.2 Velocidades

Uno de los requisitos es activar el coloreado de las partículas de acuerdo a su velocidad, por ejemplo, variando entre los colores azul y rojo. Siendo azules las partículas más lentas y rojas las más rápidas. Para satisfacer este requisito el primer paso fue pedir las velocidades al servidor donde se ejecuta la simulación. Para ello fue necesario modificar SiMon y el servidor ya que esta acción no estaba implementada. Se agregaron los comandos #sendvelocities y #stopvelocities al servidor, de manera que al recibir el primero de éstos el *thread* que envía los datos comienza a incluir las velocidades junto con las posiciones de las partículas. De manera análoga, al recibir el comando #stopvelocities se dejan de incluir las velocidades y sólo se envían las posiciones. En el lado del cliente se modificó el *thread* que recibe datos, para que incluya o no las velocidades dependiendo del comando enviado al servidor.

En la interfaz del visualizador se agregó un *checkbox* para activar o desactivar el envío de velocidades. Al ser seleccionado se emite una señal a **Viewer** indicando esta acción, el cual reacciona indicándole a **Visualization** que debe agregar color a las partículas y emitiendo una señal de vuelta a la interfaz para indicar que el botón "*Settings*" puede ser habilitado. Este botón despliega una ventana de configuración para los parámetros de asignación de colores al sistema. Más adelante se explicará más en detalle la funcionalidad de este elemento.

Visualization posee una instancia de la clase **System**, la cual representa al sistema y posee arreglos para almacenar las posiciones, velocidades y colores de las partículas. Para agregar color a las partículas **Visualization** utiliza un objeto de la clase **ParticlePainter**, ejecutando el método *color* con la instancia de **System** como argumento. Este método realiza los siguientes pasos:

- Envía el comando #sendvelocities al servidor usando la función de SiMon cmd_execute(comando) para que comience el envío de las velocidades.
- Los datos enviados consisten en las componentes x, y y z del vector velocidad de cada partícula. **ParticlePainter** obtiene la rapidez de la partícula calculando el módulo de este vector y la almacena en un arreglo.
- Calcula la rapidez mínima y máxima del sistema y actualiza el arreglo de colores del sistema asignando un valor RGB a cada partícula de acuerdo a la siguiente fórmula:

$$R = \frac{r_i - R_{min}}{R_{max} - R_{min}}$$

$$G = 0$$

$$B = 1 - \frac{r_i - R_{min}}{R_{max} - R_{min}}$$

Donde,

r_i : Rapidez de la partícula i

R_{max} : Rapidez máxima del sistema

R_{min} : Rapidez mínima del sistema

De esta manera cuando la rapidez de una partícula es igual a la rapidez máxima del sistema, su valor RGB es (1, 0, 0), es decir, es pintada roja. Así mismo, cuando la rapidez de una partícula es igual a la rapidez mínima del sistema, su valor RGB es (0, 0, 1), es decir, es pintada azul.

Como se indicó anteriormente al activar el coloreado de partículas, se habilita el botón "*Settings*" en la interfaz. Al hacer accionar este botón se emite una señal que es recibida por **Viewer**, el cual crea un objeto de clase **SettingsWindow**, que consiste en una ventana que contiene un histograma de frecuencia de la variable rapidez en el sistema. Además contiene dos *sliders* que permiten cambiar el valor de rapidez mínima y máxima del sistema para los cuales se calcula el color de las partículas como se muestra en la Figura 18. Cuando un usuario cambia el valor en el *sliders* se emite una señal a **Viewer**, el cual cambia el valor de rapidez mínima y máxima que usa **Visualization** para calcular el color de las partículas.

De esta manera todas las partículas con rapidez mayor a la rapidez máxima del sistema tendrán en su valor RGB la componente RED mayor que 1, por lo tanto se pintarán de rojo. De manera análoga para el color azul. Esto permite que el investigador pueda observar como varía la rapidez en grupos focalizados de partículas.

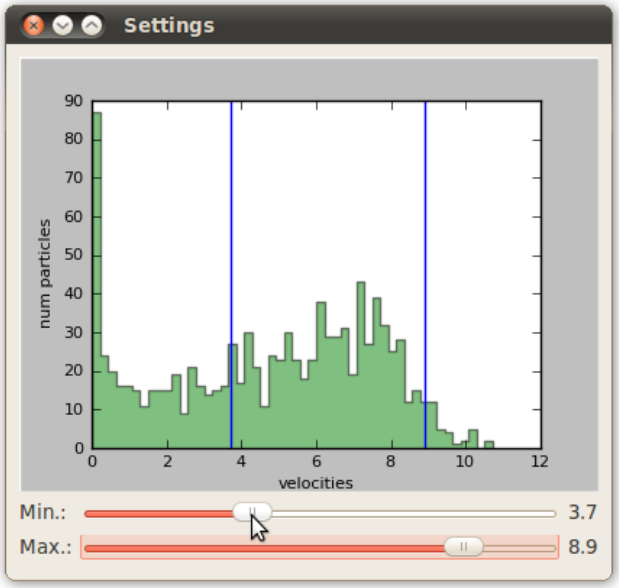


Figura 18 Ventana para configurar el valor de rapidez mínimo y máximo del sistema para el cual se calcula la rapidez de cada partícula.

4.2.6.3 Tipo

Existen sistemas que poseen partículas de distinto tipo, las cuales pueden diferenciarse en tamaño, peso, material, etc. La aplicación desarrollada permite distinguir las partículas de distinto tipo asignándole un color diferente a cada una. Como las partículas del sistema no cambian de tipo a lo largo de la simulación, basta con obtenerlo una sola vez. Para ello se aprovecha la modificación hecha en SiMon para recibir los radios de las partículas usando el protocolo TCP y se agrega además del radio la información del tipo de cada partícula al comienzo de la ejecución.

En algunos casos, además de diferenciar las partículas por su tipo, es útil distinguirlas a la vez de acuerdo a su velocidad. En este caso, se asigna un rango de colores diferentes según su velocidad para cada tipo. El alcance de este trabajo sólo contempla sistemas con a lo más dos tipos. Como se vio en la subsección anterior, en sistemas simples las partículas varían de azul a rojo de acuerdo a su velocidad. En el caso de un sistema multitipo se agrega un nuevo rango de colores que varía desde el verde al naranjo. Para ello se utiliza la siguiente función en la asignación de color de cada partícula del segundo tipo:

$$R = \frac{r_i - R_{min}}{R_{max} - R_{min}}$$

$$G = 1 - 0.3 * \left(\frac{r_i - R_{min}}{R_{max} - R_{min}}\right)$$

$$B = 0$$

Donde,

r_i : Rapidez de la partícula i

R_{max} : Rapidez máxima del sistema

R_{min} : Rapidez mínima del sistema

De esta manera cuando la rapidez de una partícula del segundo tipo es igual a la rapidez máxima del sistema, su valor RGB es (1, 0.7, 0), es decir, es pintada naranjo. Así mismo, cuando la rapidez de una partícula es igual a la rapidez mínima del sistema, su valor RGB es (0, 1, 0), es decir, es pintada verde.

Para cambiar entre los distintos criterios de coloreado se agrega a la interfaz un nuevo control que consiste en una lista desplegable que ofrece las siguientes opciones:

Distinción por velocidad: Todas las partículas son consideradas de un mismo tipo, variando sólo del azul al rojo. En la Figura 19 las partículas de distinto radio corresponden además a distinto tipo. Sin embargo, todas se encuentran en el mismo rango de colores.

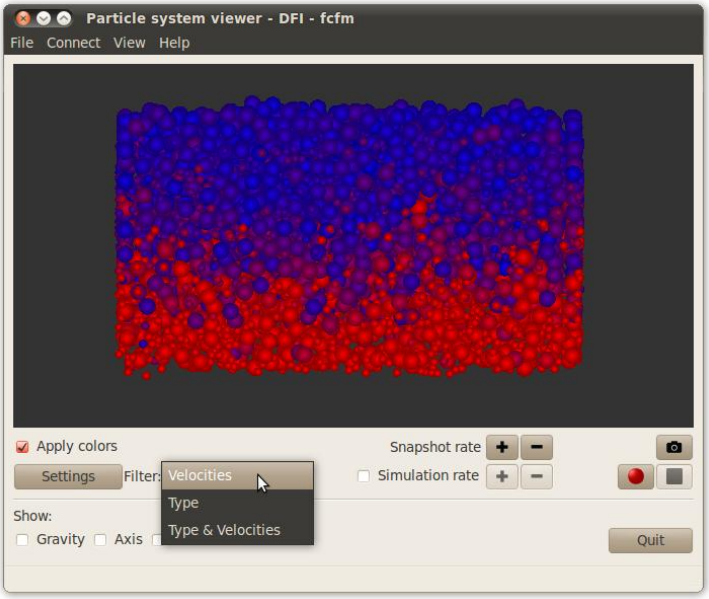


Figura 19 Distinción de partículas de acuerdo a su velocidad.

Distinción por tipo: Sólo se considera el tipo de la partícula, pintándolas con dos colores de alto contraste. En la Figura 20 las partículas de distinto radio corresponden además a distinto tipo.

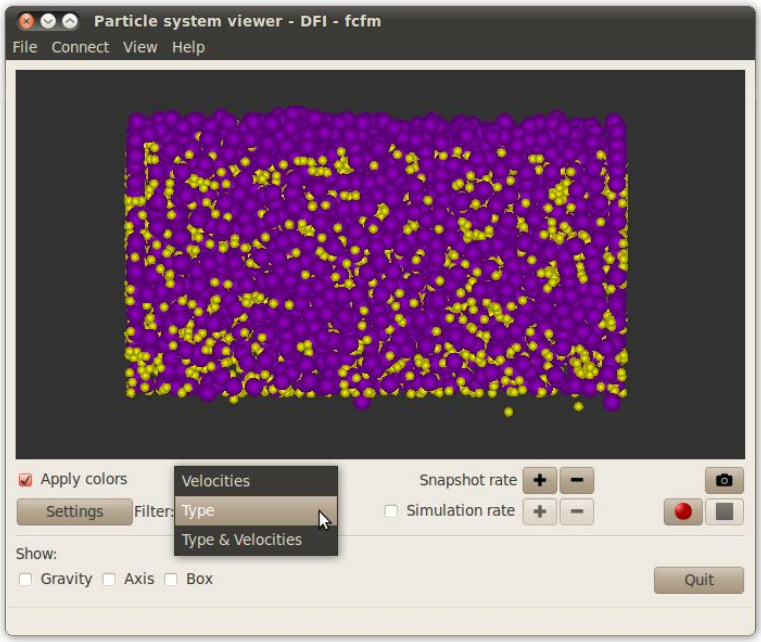


Figura 20 Distinción de partículas de acuerdo a su tipo.

Distinción por velocidad y tipo: Se considera tanto el tipo de la partícula como su velocidad. Como se mencionó previamente cada tipo varía en un rango diferente de colores. Esta variación corresponde a la variación de su velocidad. En un tipo las partículas más lentas son azules, y en el otro son verdes. Así como en un tipo las partículas más rápidas son rojas, en el otro son de color naranjo.

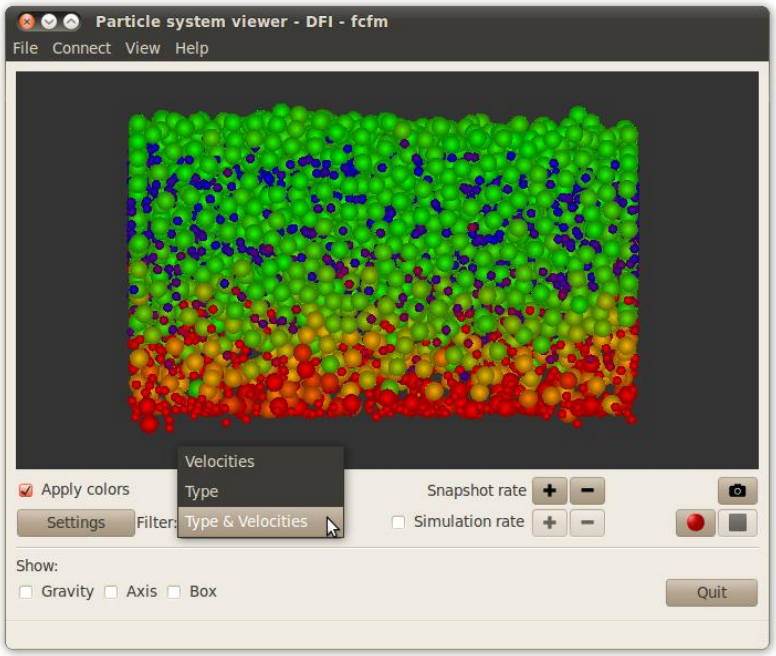


Figura 21 Distinción de partículas de acuerdo a su tipo y velocidad.

4.3 Optimizaciones

La clase **Renderer** es la encargada de hacer el *rendering* de la escena, para llevar a cabo esta tarea la primera estrategia implementada fue usar una clase **Particle** que representaba a una partícula del sistema. La cual contenía la posición, velocidad y color de una partícula. La clase System representa al sistema de partículas. En esta primera estrategia System tenía como atributos una lista de objetos de tipo **Particle** y un objeto creado con una *display list* que representaba la forma de la partícula. En este caso el objeto era una esfera creada con Glut. Una *display list* es una serie de instrucciones OpenGL almacenados para una ejecución posterior. Son útiles para casos en que una geometría se va a utilizar múltiples veces, como es el caso de un sistema de partículas, donde la geometría de una partícula se replica en todo el sistema. Para dibujar la escena, **Renderer** recorría la lista de partículas de System llamando a la *display list* que dibujaba una esfera por cada partícula en la posición y con el color correspondiente.

Esta estrategia no fue suficiente para el *rendering* de sistemas con más de 20.000 partículas, ya que si bien optimizaba recursos al usar *display list,* el hecho de usar esferas construidas a partir de polígonos seguía siendo muy costoso. Por este motivo se decidió usar dos optimizaciones para mejorar la eficiencia. La primera de ellas fue usar puntos en vez de esferas. Es decir, dibujar sólo un vértice en vez de dibujar sobre treinta vértices por cada partícula. La segunda optimización consistió en evitar el ciclo que recorría la lista de partículas y realizar el *rendering* a través de una sola instrucción de OpenGL. Ambas optimizaciones se detallan a continuación.

4.3.1 Point_sprite

Point_Sprite es una extensión de OpenGL disponible desde su versión 1.5. Consiste en puntos individuales que en algunos aspectos pueden ser tratados como cuadriláteros y cuya cara está siempre mirando al observador. Para poder representar las partículas como puntos fue necesario usar point_sprite con dos elementos adicionales:

Carga de texturas: Dado que un punto en OpenGL es dibujado con forma cuadrada, y las partículas representadas en este trabajo son esféricas, fue necesario usar una textura con forma de esfera para simular esa geometría y cargarla al *point_sprite*. El espacio que el dibujo de la esfera dejaba vacío en las esquinas de la textura debió transparentarse utilizando composición *alpha*.

***Shader* para setear el tamaño del punto:** Uno de los problemas presentados al usar esta estrategia fue que un punto en OpenGL posee un tamaño fijo, el cual puede ser escogido arbitrariamente. Por lo que al hacer zoom en la escena la partícula siempre tenía el mismo tamaño, sólo cambiaba la posición de su centro. La solución a este problema fue manejar el comportamiento de la GPU al calcular el tamaño de los vértices. Los lenguajes de shading permiten alcanzar este propósito, pues permiten escribir instrucciones que se ejecutan directamente en el pipeline gráfico de la GPU. En este caso la etapa del pipeline gráfico que se quería manipular era el procesador de vértices, el cual se encarga de las transformaciones que

afectan los vértices de las figuras en la escena. Maneja todos los cálculos sobre las características de un vértice, tales como coordenadas, tamaño, color, textura, normal, etc.

Para ello se utilizó vertex_shader, una extensión de OpenGL para manipular el procesador de vértices de la GPU usando programas escritos en algún lenguaje de *shanding*. Como se justificó en la sección 3.2.3 se decidió usar GLSL como lenguaje de *shading*. En este caso el programa de *shading* lo único que hace es asignarle un tamaño previamente calculado al vértice que está siendo manipulado. El *shader* sólo posee las siguientes instrucciones:

```
1. uniform float psize;
2. void main() {
3.     gl_Position = ftransform();
4.     gl_FrontColor = gl_Color;
5.     gl_PointSize = psize;
6. }
```

La línea 1 declara una variable uniforme de tipo *float* con nombre psize. Este tipo de variables pueden ser pasadas desde el programa principal al *shader*, el cual puede leerlas pero no escribirlas. Son llamadas variables uniformes porque no cambian de una ejecución de un *shader* a la siguiente dentro de una llamada de *rendering*. En este caso, psize almacenará el valor que se le debe asignar al tamaño del vértice.

La línea 3 lo único que hace es definir que la posición del vértice debe ser tratada de manera usual por la el procesador de vértices. La línea 4 establece que la cara frontal del vértice sea coloreada con los valores ingresados con las funciones glColor3f y glColor4f. La línea 5 asigna el valor guardado en la variable psize al tamaño del vértice, el cual se calculó utilizando la fórmula:

$$psize = \frac{(width * height)}{max(width,\ height) * distance(camera, center)} * radius$$

Para usar este *shader* en la clase Renderer se debieron seguir los siguientes pasos:

- Crear el *shader* en OpenGL (Figura 22):
    a. Crear un objeto que contenga al *shader*: En OpenGL la función que realiza esta labor es glCreateShader(GLenum shaderType), la cual recibe como parámetro el tipo de *shader*, en este caso GL_VERTEX_SHADER y retorna un controlador para el *shader*. Se pueden crear muchos *shaders* que pueden ser añadidos a un programa, pero debe existir sólo una función *main* por cada etapa del pipeline que se desee manipular.

b. Cargar el código fuente del *shader*: En OpenGL la función que realiza esta labor es glShaderSource(GLuint shader, const char **strings), la cual recibe como parámetro el controlador del *shader* recién creado, y un String con el código fuente del *shader*.

c. Compilar el *shader*: En OpenGL la función que realiza esta tarea es glCompileShader(GLuint *shader*) que recibe como parámetro el controlador del *shader* creado previamente.

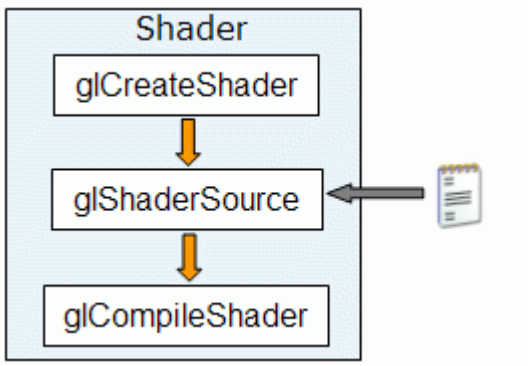


Figura 22 Pasos para crear un shader

- Crear un programa que gestione el *shader*: Para crear un programa para el *shader* se debieron seguir los siguientes pasos, resumidos en la Figura 23.

  a. Crear un objeto que contenga al programa: El primer paso fue crear el objeto que contendrá al programa. En OpenGL la función que realiza esta tarea es glCreateProgram(void). Es posible crear tantos programas como se desee. Una vez realizado el *rendering* se puede cambiar de un programa a otro.

  b. Añadir el *shader* al programa: El siguiente paso fue añadir el *shader* creado previamente al programa. La función para ellos es glAttachShader(GLuint program, GLuint shader), que recibe como parámetro el programa recién creado y el *shader* creado previamente. No es necesario que el shader esté compilado, e incluso no es necesario que tenga código fuente. Todo lo que requerido para añadir el *shader* al programa es su objeto contenedor.

  c. Enlazar el programa: El siguiente paso fue vincular el *shader* al programa. En este paso el *shader* ya debe estar compilado. En OpenGL la función que realiza este paso es glLinkProgram(GLuint program).

  d. Usar el programa: El último paso es cargar y usar el programa, para ello se usó la función de OpenGL glUseProgram(GLuint prog), que recibe como parámetro el programa creado anteriormente. Se pueden tener tantos programas enlazados y listos para usar como se desee.

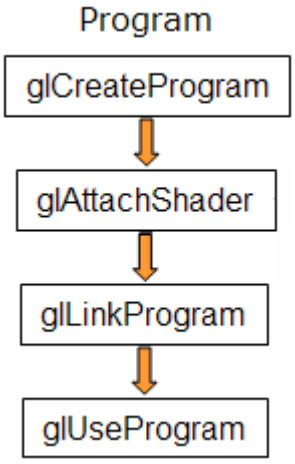


Figura 23 Pasos para crear un programa de shader

4.3.2 Vertex_array

La segunda optimización al programa fue eliminar el ciclo que recorría la lista de partículas y remplazarlo por una sola instrucción de OpenGL. Esto es posible gracias a la extensión de OpenGL vertex_array. Esta extensión añade la posibilidad de especificar múltiples primitivas geométricas con muy pocas llamadas a subrutinas. En lugar de llamar a un procedimiento de OpenGL con los vértices, normales o color de cada primitiva en la escena de manera individual, se especifican previamente arreglos que definen una secuencia de primitivas del mismo tipo y se realiza una sola llamada a *glDrawArrays* especificando la forma en que los datos están contenidos en el arreglo.

Para usar esta extensión en la aplicación fue necesario eliminar la clase Particle, ya que *glDrawArrays* sólo acepta arreglos de objetos primitivos. Se crearon arreglos usando el módulo NumPy, el cual es una extensión de Python que permite almacenar grandes volúmenes de datos y hacer cálculos sobre éstos de manera eficiente. Estos arreglos se utilizaron para almacenar las posiciones, velocidades y colores de las partículas. Las instrucciones para usar vertex_array fueron las siguientes:

```
1. glEnableClientState(GL_VERTEX_ARRAY)
2. glEnableClientState(GL_COLOR_ARRAY)
3. glColorPointer(3, GL_FLOAT, 0, colors)
4. glVertexPointer(3, GL_FLOAT, 0, positions)
5. glDrawArrays(GL_POINTS, 0, num_particles)
6. glDisableClientState(GL_VERTEX_ARRAY)
7. glDisableClientState(GL_COLOR_ARRAY)
```

Las líneas 1 y 2 establecen para qué se usará vertex_array, indicando qué representan los datos almacenados en los arreglos. Como un arreglo contiene los vértices que se dibujarán y el otro los colores, se habilitó GL_VERTEX_ARRAY y GL_COLOR_ARRAY. Las líneas 3 y 4 establecen cómo están contenidos los datos en el arreglo. El primer parámetro indica de cuantas coordenadas está definida la posición o color del vértice. En este caso las coordenadas del vértice están contenidas en grupos de a tres elementos en el arreglo. En el segundo parámetro se indica el tipo de los elementos, en este caso son de tipo *float*. El tercer parámetro indica si los elementos están consecutivos. Y finalmente, el último parámetro es el arreglo que contiene los datos. La línea 5 dibuja finalmente el sistema, utilizando puntos para representar las partículas. En la sección anterior se explicó cómo un punto logra representar una partícula por medio de point_sprite y el uso de texturas. Este paso sigue utilizando el mismo método, la diferencia radica en que en vez de recorrer los arreglos de posiciones y colores para sacar la información y luego graficar, se utiliza una sola instrucción de OpenGL entregando estos arreglos como argumentos. Las líneas 6 y 7 indican que ya se terminó de utilizar vertex_array.

4.4 Comparación de estrategias

A continuación se presenta una tabla comparativa que muestra cómo las optimizaciones detalladas en la sección anterior incrementaron la eficiencia del programa. Las pruebas fueron realizadas en un computador con sistema operativo Linux, distribución Ubuntu 10.04, memoria RAM de 3GB, procesador Pentium Dual-Core T4400 de 2.20GHz, y una conexión a Internet de 4Mbps.

Tamaño del sistema ⇨ ⇩ Técnica	100.000 partículas	200.000 partículas	300.000 partículas	400.000 partículas	500.000 partículas
Glut + display_list	0.2 fps	-	-	-	-
Point_sprite (+ texturas y shader)	10 fps	5 fps	-	-	-
Point_sprite + vertex_array	75 fps	42 fps	30 fps	23 fps	18 fps

Tabla 1. Comparación de estrategias de *rendering*

En la primera columna se listan las estrategias usadas. Glut + display_list se refiere a la primera estrategia que consistió en recorrer una lista de objetos de tipo Particle, dibujando en cada iteración una esfera en la posición y con el color de la partícula correspondiente. La esfera se dibujó llamando a una display_list previamente compilada, la cual usaba Glut para construir una esfera con la menor cantidad de vértices posibles, de radio 0.5, 6 slices (divisiones en torno al eje z, similares a las líneas de longitud) y 4 stacks (divisiones a lo largo del eje z, similares a las líneas de latitud).

La segunda estrategia consistió en representar la partícula con un sólo punto, al cual se le cambió el tamaño de acuerdo al radio de la partícula que representaba, y se le asoció una textura que consistía en una imagen de una esfera. Para ellos se usó la extensión de OpenGL llamada point_sprite, que permite dibujar puntos cuya cara siempre está dirigida al observador, así al rotar la escena el punto simula un objeto en 3D y no una imagen plana. También se utilizó un shader para cambiar el tamaño del punto.

La tercera estrategia fue agregar una optimización a la implementación con point_sprite, que consistió en disminuir la cantidad de llamadas a funciones OpenGL. Para ello, se usó la extensión de OpenGL llamada vertex_array, así en vez de recorrer los arreglos para obtener la información de cada partícula para luego graficarla, se realiza el *rendering* de todas las partículas a través de una sola instrucción de OpenGL que toma como argumento los arreglos completos con las posiciones y colores de las partículas. En esta optimización se sigue representando a las partículas como puntos con texturas.

5 Conclusiones

5.1 Resultados obtenidos

Durante el trabajo de memoria se desarrolló e implementó una aplicación que permite visualizar simulaciones físicas de sistemas de partículas y provee controles de visualización que le permiten al usuario observar distintos aspectos de las simulaciones monitoreadas.

La aplicación se diseñó utilizando el paradigma de programación orientada a objetos y patrones de diseño, a fin de otorgar una solución extensible y mantenible. De esta manera se facilita el desarrollo futuro sobre la aplicación.

En su implementación se debió adaptar el trabajo anterior de un alumno memorista de la Universidad del Bío Bío para la comunicación entre la aplicación y las simulaciones. Esta herramienta de comunicación fue implementada en C, por lo que se debió crear una capa intermedia que consta de una interfaz con funciones útiles para este contexto en Python, lenguaje de programación utilizado en este trabajo. Así, además de la aplicación, se obtiene una librería reutilizable para futuros trabajos con las simulaciones en los que se desee utilizar este lenguaje de programación.

Se implementaron funcionalidades para rotar libremente la vista, grabar videos, capturar imágenes, distinguir partículas según su tipo y velocidad, configurar de manera fácil parámetros de asignación de color, permitir conexión remota a simulaciones, contar con un historial de servidores para facilitar el acceso, etc. Estas funcionalidades son encapsuladas en distintos módulos, los cuales se comunican entre sí por una única instancia del visualizador. Esto facilita la mantención de la aplicación, pues se puede mantener cada módulo por separado sin necesidad de interferir en todos a la vez.

Se emplearon distintas estrategias para mejorar el rendimiento de la aplicación, dentro de las cuales se puede destacar el uso de *shaders* para manipular las propiedades de los vértices de OpenGL en la GPU. Esto sumado a la extensión point_sprite de OpenGL permitió utilizar puntos para la representación de las partículas, evitando el *rendering* de polígonos más complejos, como los utilizados en una primera estrategia consistente en usar esferas dibujadas con Glut. Para simular la forma esférica de las partículas se agregaron texturas a los puntos. Dentro del *shader* se modificó el tamaño del punto para que tomara el valor correspondiente de acuerdo a su posición en la escena. El uso de point_sprite aumentó la cantidad de *fps* de 1 a 10 para 100.000 particulas. Sin embargo para valores mayores ninguna de las dos estrategias fue capaz de realizar el *rendering* en un tiempo razonable. La tercera estrategia consistió en agregar una optimización a la implementación con point_sprite, para aumentar la eficiencia de la aplicación, la cual consistió en usar la extensión de OpenGL llamada vertex_array. Esta extensión añadió la posibilidad de especificar múltiples primitivas geométricas con muy pocas llamadas a subrutinas. Esto aumentó los *fps* considerablemente, pasando de 10 *fps* a 75 *fps* para 100.000 partículas, siendo la estrategia con mejores resultados.

Como resultado se obtuvo una herramienta que permite visualizar simulaciones de sistemas de partículas de manera eficiente y en tiempo real, la cual consta de múltiples funciones, es robusta, fácil de usar, fácil de mantener y extender. Gracias al uso de programación orientada a objetos el diseño permite agregar fácilmente nuevos métodos de entrada para recibir los datos de la simulación, nueva geometría de las partículas y nuevos criterios de coloreado (Ver Anexo B).

5.2 Trabajo futuro

Se puede mejorar la aplicación desarrollada en distintas maneras:

- Permitir visualización con datos de simulaciones ya finalizadas. Esto sería útil cuando las simulaciones no pueden ser monitoreadas en tiempo real, guardando sus datos, por ejemplo, en un archivo para su posterior visualización. Gracias al paradigma orientado a objetos utilizado en la implementación, agregar un nuevo método de entrada resulta más sencillo.

- Implementar la transmisión de datos mediante el pipe de procesos. Actualmente, se transmiten cerca de 4MB por *frame* para simulaciones con 100.000 partículas. Esto hace que la actualización de datos sea muy dependiente de la conexión al servidor de la cual se dispone, problema que no existiría si la transmisión fuera mediante el pipe de procesos.

- Uso de la GPU para otros cálculos. En este trabajo sólo se utiliza la GPU para calcular el tamaño del punto que representa la partícula, pero podría utilizarse para almacenar y actualizar los grandes arreglos usados para guardar los datos de la simulación. Esto aumentaría aún más la eficiencia.

- Agregar más ventanas de visualización, a fin de observar distintos modos de visualización de forma paralela. Podría ser útil en simulaciones donde ocurren fenómenos en distintos ángulos, y así poder observarlos sin tener que rotar la vista y al mismo tiempo.

6 Referencias

[1] X. Cai, J. Li, J. Yang y Z. Su, «Advanced GPU-Based State-Preserving Particle System» *Proceedings of the 7th World Congress on Intelligent Control and Automation*, Chongqing, China, 25 - 27 June 2008.

[2] T. Akenine-Möller, E. Haines y N. Hoffman, Real-Time Rendering, Wellesley, Massachusetts: A K Peters, Ltd., 2008.

[3] J. B. Barra, «Desarrollo de una aplicación gráfica en modalidad Cliente-Servidor para el monitoreo de simulaciones de Dinámica Molecular conducida por eventos». Tesis para optar al título de Ingeniero civil informático. Universidad del Bío Bío. Profesores guía: Dino Risso Rocco y Sergio Bravo Silva. Concepción. 2010.

[4] D. Luebke y G. Humphreys, «How GPUs Work». *IEEE Computer Society Press*, vol. Vol. 40, nº N° 2, pp. 96-100, February 2007.

[5] L. Latta, «Building a Million Particle System» Proceedings of Game Development Conference, 77-82, San Francisco, CA. March 2004.

[6] S. Drone, «Real-Time Particle Systems on the GPU in Dynamic Environments». Proceedings of the conference on SIGGRAPH 2007 course notes, 80-86. 2007.

[7] D. Risso, «Convección en sistemas finitos». Tesis para optar al título de Doctor en ciencias mención física. Universidad de Chile. Profesor guía: Patricio Cordero. Santiago. 1994.

[8] M. Marín, «Diseño de un ambiente de simulación especializado en dinámica molecular conducida por eventos». Tesis para optar al título de Magister en ciencias de la ingeniería mención computación. Universidad de Chile. Profesores guía: Patricio Cordero y Ricardo Baeza. Santiago. 1992.

[9] http://www.opengl.org/, *OpenGL documentation,* 20 Abril 2011.

[10] http://msdn.microsoft.com/en-us/directx/bb896684, *DirectX documentation,* 20 Abril 2011.

[11] http://docs.python.org/extending/extending.html, *Extending Python with C or C++*, Python documentation, 09 Agosto 2011.

[12] http://www.opengl.org/documentation/glsl/, *GLSL,* OpenGL documentation, 29 Junio 2011.

[13] http://developer.nvidia.com/cg-toolkit, *CG,* Nvidia documentation, 29 Junio 2011.

[14] R. Pang, *GLSL vs Cg,* http://www.eng.utah.edu/~cs5610/lectures/GLSLvsCG.pdf, 29 Junio 2011.

[15] http://qt.nokia.com/qt-in-use, *Usos de Qt,* 05 Febrero 2012.

[16] http://qt.nokia.com/about/licensing, *Licencia de Qt.*

[17] http://www.swig.org/, *Sitio oficial de SWIG,* 19 Septiembre 2011.

[18] D. M. Beazley, «Swig Master Class,» Chicago, Illinois., 2008.

[19] D. Norman, The design of everyday things, 2002.

[20] http://www.libqglviewer.com/index.html, *QGLViewer,* 05 Febrero 2012.

[21] http://pyqglviewer.gforge.inria.fr/, *PyQGLViewer- Bindings de QGLViewer para Python,* 05 Febrero 2012.

[22] http://www.riverbankcomputing.co.uk/software/pyqt/intro, *PyQt - Bindings de Qt para Python,* 05 Febrero 2012.

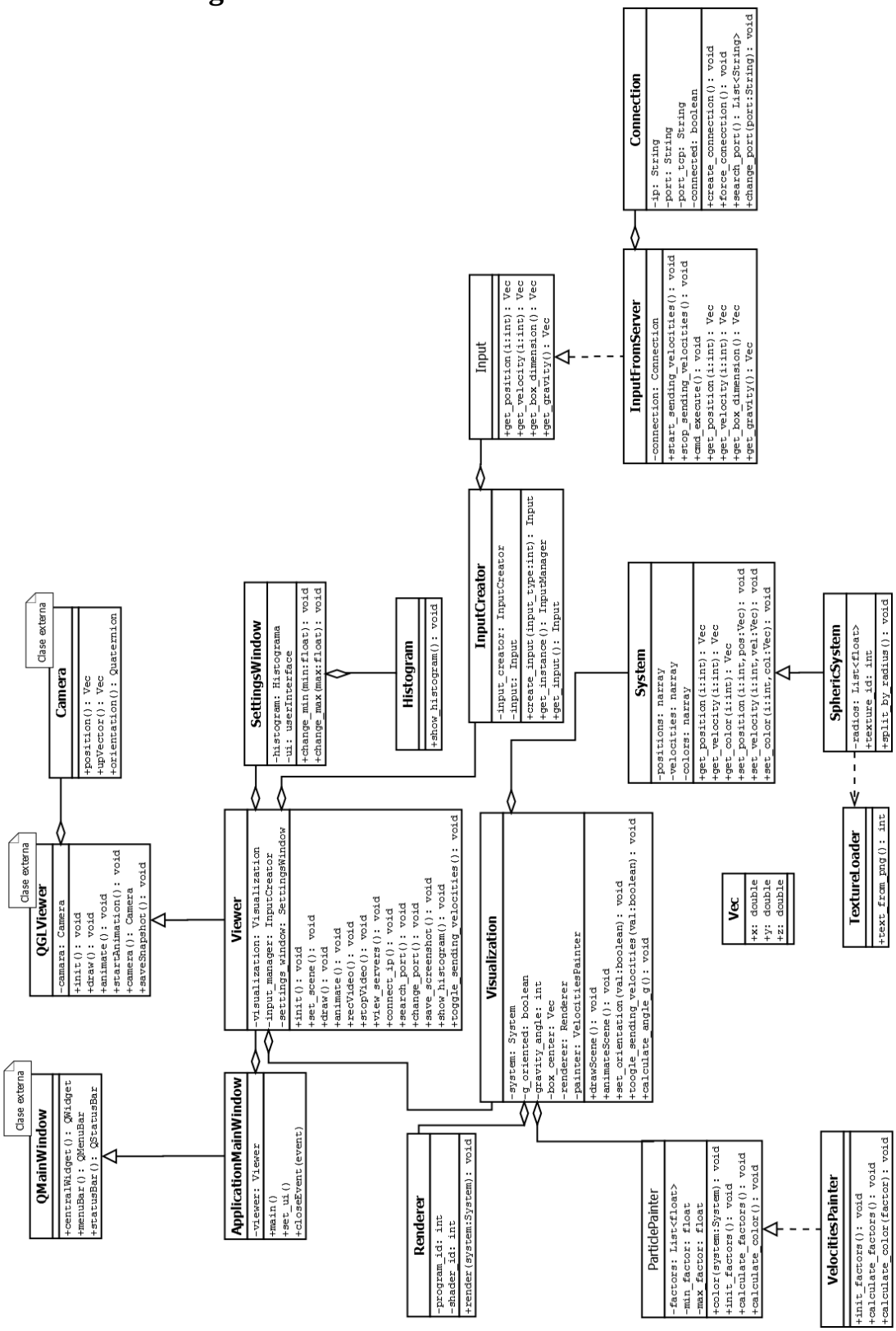
[23] http://matplotlib.sourceforge.net/, *Matplotlib,* 05 Febrero 2012.

[24] http://numpy.scipy.org/, *NumPy,* 05 Febrero 2012.

[25] R. R. a. J. ErichGamma, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Ed., USA, 1994.

7 Anexos

7.1 Anexo A: Diagrama de clases



7.2 Anexo B: Configuración básica de la aplicación

7.2.1 Personalizar colores

El archivo config.py contiene la definición de colores para distintos elementos del visualizador. Cada variable se define como una tupla de tres elementos que representan los valores RGB, en un rango de 0 a 1. Se encuentran disponibles las siguientes variables:

- initial_color: Define el color inicial de las partículas. El valor predeterminado es (1.0, 1.0, 1.0).
- min_color1: Define el color para el valor mínimo de la característica por la cual pinta las partículas del tipo 1. Esta característica se define en la clase ParticlePainter. El valor predeterminado es (0.0, 0.0, 1.0).
- max_color1: Define el color para el valor máximo de la característica por la cual pinta las partículas del tipo 1. El valor predeterminado es (1.0, 0.0, 0.0).
- min_color2: Define el color para el valor mínimo de la característica por la cual pinta las partículas del tipo 2. El valor predeterminado es (0.0,1.0,0.0).
- max_color2: Define el color para el valor máximo de la característica por la cual pinta las partículas del tipo 2. El valor predeterminado es (1.0,0.7,0.0).
- type_1_color: Define el color de las partículas del tipo 1. El valor predeterminado es (1.0,1.0,0.0).
- type_2_color: Define el color de las partículas del tipo 1. El valor predeterminado es (0.6,0.0,0.8).
- ice_color: Define el color de las partículas inmóviles, usadas principalmente para definir el borde en algunos sistemas. El valor predeterminado es (0.0 ,0.0, 0.0).
- box_color: Define el color de la caja que contiene al sistema de partículas. El valor predeterminado es (1.0, 1.0, 1.0).
- background_color: Define el color de fondo de la ventana de visualización. El valor predeterminado es (51.0, 51.0, 51.0).

Además, en este mismo archivo de configuración es posible definir la figura utilizada para representar la geometría de las partículas. Basta cambiar la variable path_image especificando la ruta de la textura que definirá la nueva geometría.

7.2.2 Extender ParticlePainter

Para colorear el sistema por una nueva característica, es necesario extender la clase ParticlePainter y sobrescribir los métodos function_color e init_factors. Por ejemplo, si se desea calcular el color de las partículas considerando sólo la coordenada *x* de la velocidad, se debe agregar la siguiente clase al archivo Painter.py:

```
1.  class SamplePainter(ParticlePainter):
2.
3.      def function_color(self, *args):
4.          velocity = args[0]
5.          y = velocity[1]
6.          return y
7.
8.      def init_factors(self, system):
9.          total = system.num_particles[0]+system.num_particles[1]
10.         for i in range(0, total):
11.             velocity = system.get_velocity(i)
12.             system.factor_color[i] = self.function_color(velocity)
```

El método function_color debe recibir como parámetro los datos necesarios para calcular el color de la partícula de acuerdo a la nueva característica. En este caso necesita las coordenadas de la velocidad de la partícula. Si se desea calcular de acuerdo a cualquier otro dato, éste debe ser pasado como argumento en la llamada al método paint de ParticlePainter, la cual se hace en System.py, en la rutina de actualización de datos del sistema.

El método init_factors inicializa el arreglo del sistema factor_color, que almacena los valores de la característica para cada partícula. Init_factors usa factor_color para calcular estos factores.

```
1.  class ParticlePainterFactory(object):
2.
3.      def createParticlePainter(self, name):
4.          if name == "VelocitiesPainter":
5.              return VelocitiesPainter()
6.          elif name == "SamplePainter":
7.              return SamplePainter()
8.          else:
9.              raise NotImplementedError
```

Por otro lado se debe agregar a ParticlePainterFactory en el mismo archivo, el caso en el que se desea crear un objeto del tipo SamplePainter, asignándole un String para representar su nombre. Para definir el ParticlePainter que utilizará el visualizador se debe indicar en el archivo config.py su nombre en la variable particle_painter, en este caso:

```
1. particle_painter = "SamplePainter"
```

Anexo C: Ejemplos de visualización

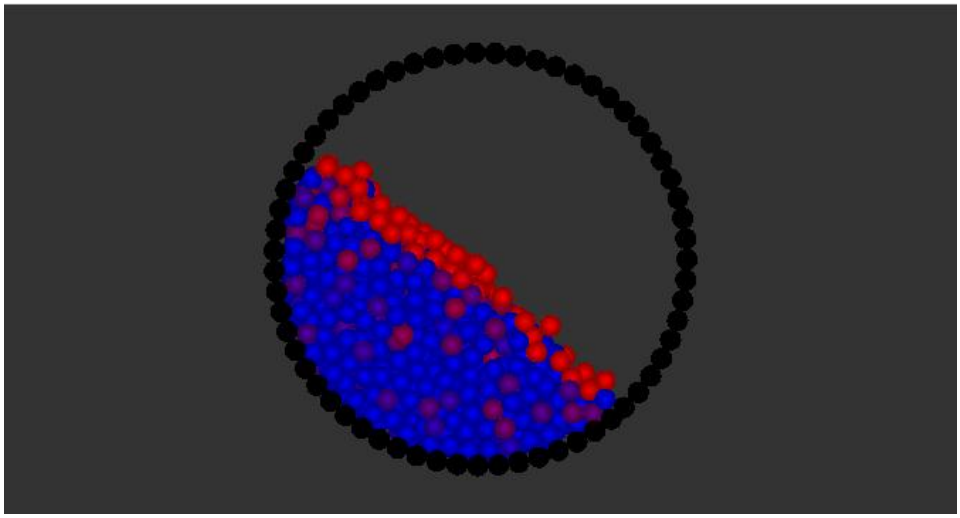


Figura 24 Distinción por tipo de partículas. Partículas negras son inmóviles y representan el borde del sistema.

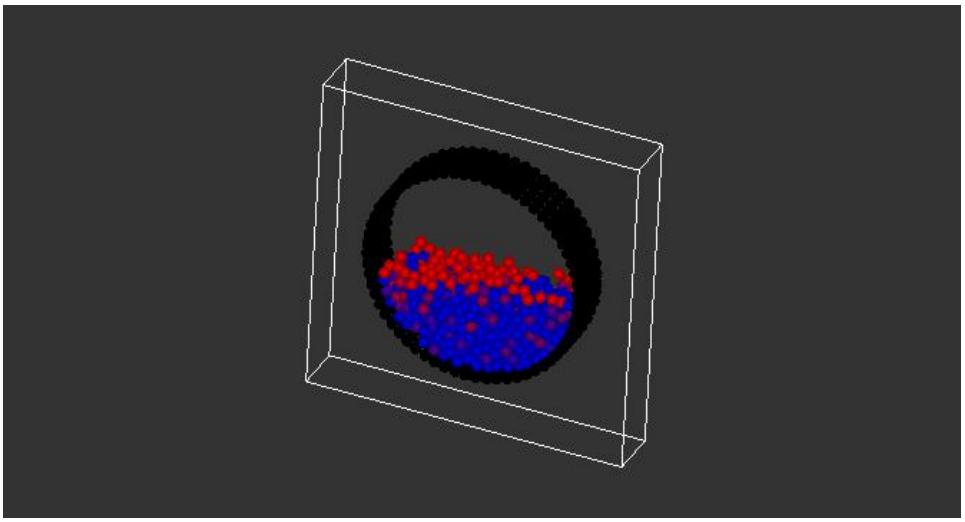


Figura 25 Visualización que incluye caja que contiene al sistema

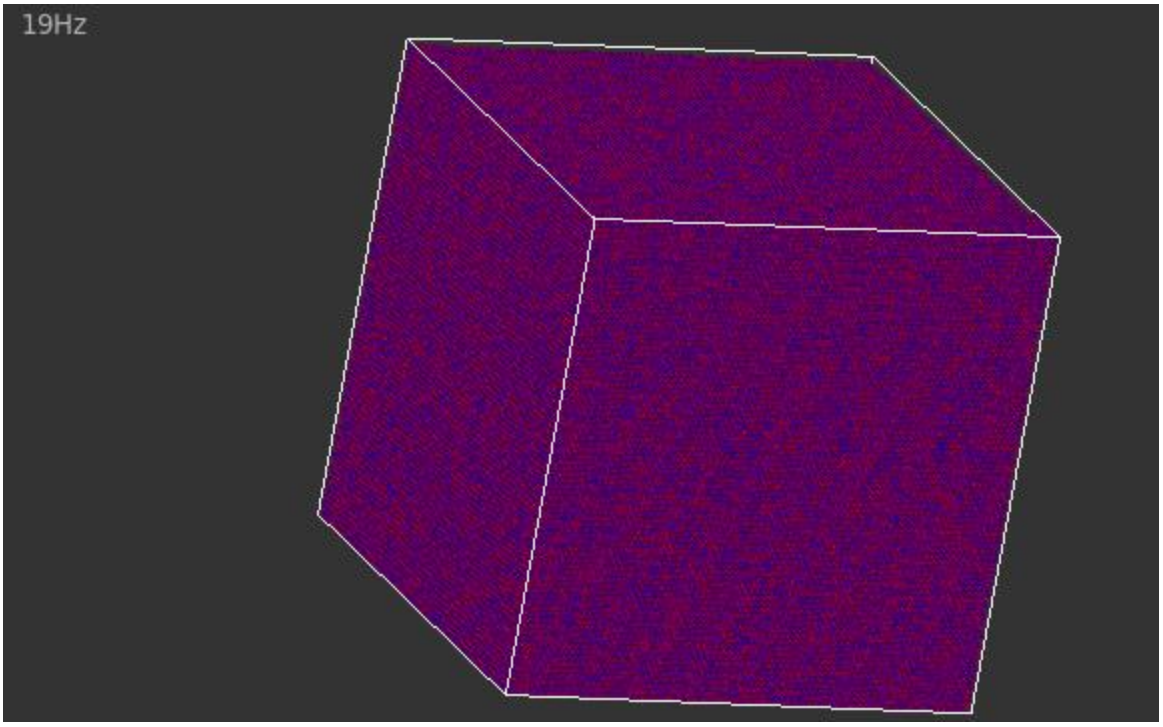


Figura 26 Visualización de sistema de 500.000 partículas

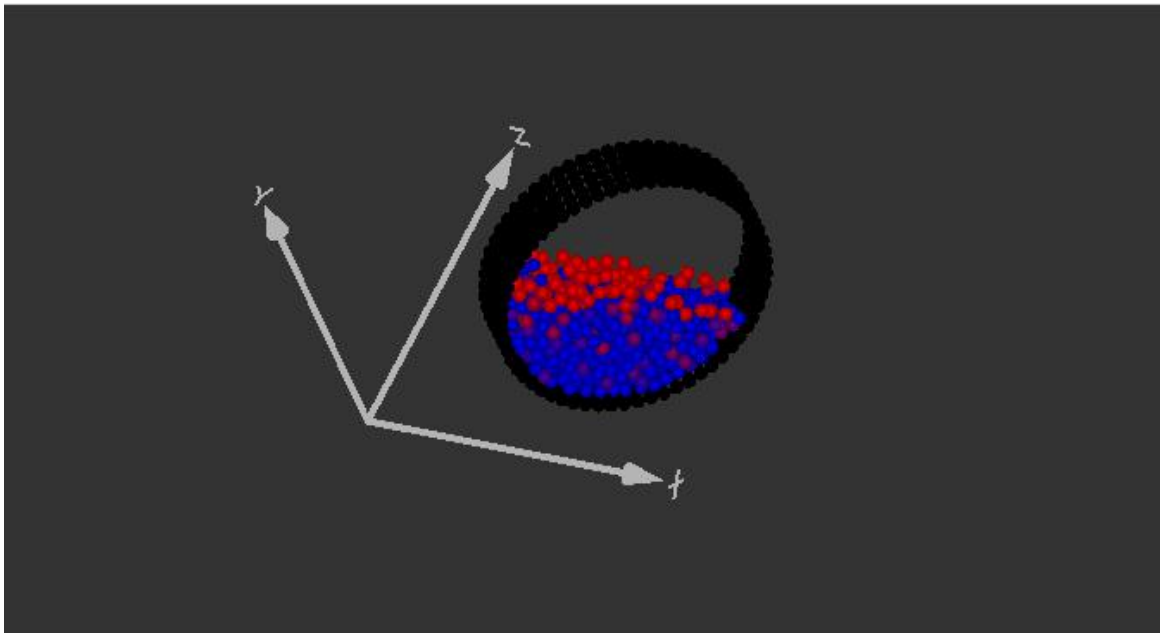


Figura 27 Visualización que incluye eje de coordenadas