Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
2020 - 2

Tarea 1

Fecha de entrega código e informe: Martes 15 de Septiembre

Objetivos

- Modelar un problema y sus propiedades de manera recursiva, usando una estructura de datos adecuada para resolverlo eficientemente.
- Utilizar técnicas de dividir para conquistar para encontrar una solución eficiente a un problema.
- Investigar de manera personal estructuras de datos para resolver un problema específico.

Introducción

Habiendo trascendido sus limitantes biológicas con la ayuda de tu programa en C, e inspirado por cierta policía japonesa, Guido Van Slimossum ha decido hacer un *upgrade* a su nuevo cerebro digital.

Estando sólo restringido por sus provisiones de silicio, Guido ha diseñado un complejo sistema de espejos y partículas de luz con el cual busca remplazar la anticuada sinapsis. En este sistema cada fotón reemplaza a un estímulo nervioso que debe ser transmitido desde un punto a otro del cerebro digital, rebotando en los espejos que encuentre en su trayectoria[1].

Debido a sus reducidas capacidades en C tras una fallido intento de auto-neuro-cirugía[2], Guido se ha visto en la necesidad de recurrir a ti, su experto favorito, para modelar el comportamiento de los componentes de su nuevo fantasma en la máquina. Afortunadamente para ti, te ha facilitado una primera versión totalmente funcional[3] de su nueva conciencia, por lo que solo debes ocuparte de hacerla eficiente usando técnicas de Dividir para Conquistar.

Problema: Divide and conquer goes brrr

Dado que el código base ya permite identificar correctamente las colisión entre un fotón y un espejo, tu labor es optimizar la búsqueda de segmentos con los que cada partícula puede chocar. De esta manera, debes implementar una estructura de datos apropiada para resolver este problema de búsqueda, **no solucionar el problema geométrico.**

[1] Fenómeno que los tecno-psicólogos llaman auto-reflexión
[2] En donde también perdió sus habilidades para diseñar un lenguaje OOP de calidad
[3] Tampoco es de calidad

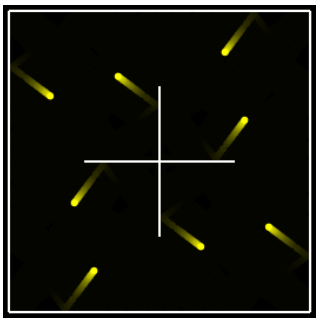

Figura 1: Fotones y espejos.

Modelación del Problema

Para modelar el problema, se usarán las estructuras de *particle* que simulan los fotones y *segment* que representan los espejos. Cada partícula tiene un *id*, un atributo de posición, y otro de velocidad. Los segmentos, por otra parte, tendrán un *id* y una ubicación fija dentro del mapa dada por un punto inicial y final en que se traza una recta.

Tras tener el estado inicial de fotones y espejos, comienza la simulación, que será modelada con *frames*. Al comienzo, se revisan todas las colisiones entre fotones y espejos, y se actualizan la dirección del movimiento de los fotones implicados en dichas colisiones. Luego, se actualiza la posición de cada fotón según su velocidad, lo cuál afectará las colisiones en el próximo *frame*.

Solución del Problema

En la versión entregada en el código base, se encuentra implementada una solución *naïve*. Esta consiste en, para cada fotón, revisar cada espejo para ver si hay colisión. En el caso en que una partícula tenga más de una colisión por *frame*, se desempatará por *id* de segmento, es decir, **sólo se considerará aquella colisión con el segmento con menor id**. El pseudocódigo de este algoritmo se ve así:

```
1:  function FIND_COLLISIONS(particles, segments, frames):
2:      for all frame F in frames do:
3:          for all particle P in particles do:
4:              set colliding segment X to none
5:              for all segment S in segments do:
6:                  if particle P collides with S then
7:                      update colliding segment X with S
8:                  end if
9:              end for
10:             update direction of particle P according to X
11:             update position of particle P
12:         end for
13:     end for
14: end function
```

Como se puede observar, este algoritmo tiene una complejidad $\mathcal{O}(FPS)$, donde F es la cantidad de frames, P es la cantidad de partículas y S la cantidad de segmentos.

El código base implementa este algoritmo directamente en `C`. Para esta tarea deberás organizar los obstáculos en una estructura de datos de manera de poder ejecutar este algoritmo en tiempo $\mathcal{O}(FP \cdot log(S))$. De esta forma, el algoritmo será eficiente en escenarios complejos con una alta cantidad de segmentos.

Para realizar esto debes ocupar un **B**ounding **V**olume **Hi**erarchy, el cual es un tipo de árbol para objetos geométricos, en la que cada uno de estos objetos se encuentra totalmente contenido en una caja. Esta estructura tiene la característica de que la caja que envuelve un nodo x siempre contiene a las cajas de los hijos de x. Por ejemplo, en la Figura 2, la caja del nodo 1 envuelve completamente las cajas de los elementos **A** y **B**. Asimismo, la totalidad del árbol esta contenido en una sola caja que corresponde al **nodo raíz** 0.

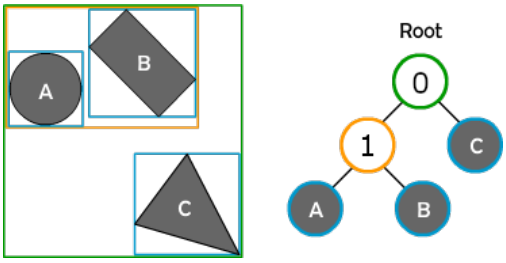

Figura 2: Ejemplo gráfico de BVH.

Para la solución de este problema se recomienda implementar un **KD-Tree** de segmentos en dos dimensiones, sin embargo, eres libres de investigar la estructura de datos que consideres más apropiada para resolver el problema, mientras no superes la complejidad esperada.

Ejecución

Tu programa se debe compilar con el comando `make` y debe generar un binario de nombre `simulate` que se ejecuta con el siguiente comando:

```
./simulate <input.txt> <output.txt>
```

Donde `input` será un archivo con los espejos y el estado inicial de los fotones. Por otro lado, `output` será el archivo en el cual se irán reportando las colisiones de cada frame.

Tu tarea será ejecutada con *tests* de dificultad creciente, asignando puntaje a cada una de las ejecuciones que tenga un `output` igual al esperado.

Se les facilitará archivos de *debugging* para que puedan visualizar y entender el algoritmo. Sin embargo, estos archivos son solo para este fin y **no serán evaluados.**

Input

El input es procesado por la función `simulation_init_from_file(char* input_file, bool visualize)` la cual retorna un objeto `Simulation*` que contiene la cantidad de frames de la simulación, y los arreglos de partículas y segmentos. El flag `visualize` indica si se debe abrir la ventana para visualizar el problema.

Output

El output de tu programa corresponde al mismo output que genera el código base, por lo que tu programa debiese replicarlos. Recuerda que tu implementación tiene que ver con mejorar la complejidad del algoritmo, que de por sí, es correcto.

Código Base

Este problema ya viene completamente resuelto en el código que se te ha dado. En particular, podrás encontrar 4 módulos:

- **src/simulate/**: solución del problema. Tus cambios deberán ir en esta carpeta.
- **src/visualizer_core/**: módulo a cargo de la visualización del problema. Utiliza GTK+3, asegúrate de leer esta guía de instalación
- **src/visualizer/**: API de **visualizer_core** para usarlo desde tu programa. Las funciones para mostrar la simulación ya se llaman en el código base. Para debugear, puedes utilizar además las siguientes funciones:
  - `void visualizer_set_color(double R, double G, double B);` indica el color que se deberá usar para las futuras operaciones de dibujo de segmentos. Los valores de R, G y B van entre 0 y 1.
  - `bool visualizer_draw_box(BoundingBox boundingbox);` dibuja la caja en la interfaz.
- **src/engine/**: contiene la modelación del problema y sus operaciones. Dentro de este módulo se encuentra particle.h con las siguientes funciones:
  - `bool particle_boundingbox_collision(Particle particle, BoundingBox boundingbox);` retorna `true` si la partícula está chocando con la caja, `false` si no.
  - `bool particle_segment_collision(Particle particle, Segment segment);` retorna `true` si la partícula está chocando con el segmento, `false` si no.
  - `void particle_bounce_against_segment(Particle* particle, Segment segment);` cambia la dirección de la partícula luego de chocar con un segmento.
  - `void particle_move(Particle* particle);` actualiza la posición de la partícula según su velocidad.

Análisis

Deberás escribir un informe de análisis donde menciones los siguientes puntos:

- Calcula y justifica la complejidad en notación $\mathcal{O}$ para la construcción del árbol en función de los parámetros **S** (*segments*), **P** (*photons*) y **F** (*frames*), según corresponda.
- Calcula y justifica la complejidad en notación $\mathcal{O}$ para la búsqueda en árbol en función de los parámetros **S** (*segments*), **P** (*photons*) y **F** (*frames*), según corresponda.
- Justifica la complejidad en notación $\mathcal{O}$ para la ejecución del programa total en función de los parámetros **S** (*segments*), **P** (*photons*) y **F** (*frames*), según corresponda.
- Compara empíricamente las velocidades de ejecución de su programa respecto al programa inicial con el algoritmo *naïve* entregado, considerando el efecto de los parámetros **S** (*segments*), **P** (*photons*) y **F** (*frames*), en la velocidad de cada uno.

Evaluación

La nota de tu tarea se descompone como se detalla a continuación:

- 70% a la nota de tu código: que retorne el `output` correcto. Para esto, tu programa será ejecutado con archivos de input de creciente dificultad, los cuales tendrán que ser ejecutados en menos de 10 segundos cada uno.
- 30% a la nota del informe, que debe contener tu análisis.

Entrega

Código: GIT - Repositorio asignado. Se entrega a más tardar el día de entrega a las 23:59 hora de Chile continental.

Informe: SIDING - En el cuestionario correspondiente, en formato PDF. Sigue las instrucciones del cuestionario. Se entrega a más tardar el día de entrega a las 23:59 hora de Chile continental.

Bonus

Los bonus sólo aplican si la nota respectiva es mayor o igual a **5.0**.

- **Análisis de árbol a medida:**
  - **(5 décimas a la nota de informe)** Analiza teórica y empíricamente el efecto que se obtiene en la complejidad de construcción y búsqueda en el árbol utilizado para los espejos, al variar el tamaño de sus hojas (cuántos espejos hay en cada hoja del árbol). Elige un tamaño óptimo.
  - **(5 décimas a la nota de informe)** Propón y analiza una forma de mejorar el criterio de división de cada nivel del **K-D Tree** (o equivalente), considerando como criterio por defecto el elegir la mediana. Compara las complejidades teórica y empíricamente al usar la mediana y el(los) criterio(s) mejorado(s).
- **Manejo de memoria perfecto:** (5 décimas a la nota de código) obtendrás este bonus si solicitan y liberan memoria de manera perfecta, es decir, **valgrind** reporta en tu código 0 leaks y 0 errores de memoria en todos los archivos de input. Se aplicará este bonus solo si el output de tu programa es correcto.