Tarea 1

Moisés Saavedra Cáceres - 16636813

1 Complejidad en construcción del árbol

El árbol solamente va a depender de la cantidad de segmentos (S) que se entrega como input, ya que las particulas y los frames no son almacenados en la estructura de nodo.

Según la función implementada, el árbol parte con la raíz que posee todos los segmentos S, pero previamente se realiza una iteración de quicksort, por lo que se tiene en el primer nodo una complejidad de O(S^2).

Ahora, como estamos ocupando un árbol binario, para cada hijo se va dividir en dos el nodo madre. En el peor caso se repartirá solamente 1 y en el otro S-1 (para la primera profundidad). Si analizamos la profundidad k óptima donde la cantidad de segmentos a repartir es 200 (cota ocupada para terminar la creación recursiva) la expresión de cantidad de nodos sería S-k+200 con k=S.

Ahora, como la profundidad k óptima es desconocida a priori y depende del tañamo del input S, tenemos que siempre existirá un quicksort O(S-k+200) en dicho óptimo. Al acotar el peor caso, notamos que para cada división existirá por un lado un O(S-k) y un O(1) en cuanto complejidad de quicksort. Luego, como cada iteración de la repartición binaria de nodos, sabemos mediante la demostración vista en clases de "dividir para conquistar", será de O(log_2(S)) para el peor caso.

Finalmente, al juntar ambas complejidades (ya que existe quicksort dentro de cada nivel de profundidad) tenemos que la creación de segmentos es del orden O($S^2 log_2(S)$).

```
void insertar_nodo(Nodo* parent) {
  /* Si tiene solamente 2 o menos segmentos se corta*/
  if (parent->largo_segmentos <= 1000) {
    return;
  }
  /* Si tiene mas de dos segmentos se crean hijos*/
  else{
    /* Profundidad par*/
    if (parent->profundidad % 2 == 0){
      quickSort(parent->segmentos, 0, parent->largo_segmentos-1, true);
    }
    /* Profundidad impar */
    else{
      quickSort(parent->segmentos, 0, parent->largo_segmentos-1, false);
    }

    /* Creacion recurisva de los hijos */
    parent->hijo_izquierdo = crear_hijo(parent, true);
    parent->hijo_derecho = crear_hijo(parent, false);

    insertar_nodo(parent->hijo_izquierdo);
    insertar_nodo(parent->hijo_derecho);

  }
}
```

Figure 1: Función insertar

2 Complejidad de búsqueda en el árbol

Para este caso, solamente va a depender de la cantidad de segmentos (S) que se entregan como input, ya que se analiza para una particula y un frame en específico.

Ahora, como estamos ocupando un árbol binario y mi implementación está basada en divisiones verticales y horizontales, donde **nunca** las cajas se solapan entre si, se escoge un nodo o su hermano, de forma excluyente para cada profundidad de búsqueda. Dicha recursión es identica a una búsqueda binaria y finaliza a una profundidad k. Si seguimos en el peor caso descrito anteriormentel, la búsqueda se detinene a la profundidad S-k = 200, ya que es la capacidad definida de creación de nuevos hijos. Luego, al llegar al nodo deseado se realiza un cómputo de cada segmento de dicho nodo para verificar si la particula entregada choca o no con algún segmento, dicha operación es de O(200).

Por lo tanto, como la búsqueda es idéntica a una búsqueda binaria (demostrada en clases) se tiene que en el peor caso, la búsqueda implementada en mi código es de O($log_2(S-k)$), lo que radica en $O(log_2(S))$.

```
void insertar_nodo(Nodo* parent) {
  /* Si tiene solamente 2 o menos segmentos se corta*/
  if (parent->largo_segmentos <= 1000) {
    return;
  }
  /* Si tiene mas de dos segmentos se crean hijos*/
  else{
    /* Profundidad par*/
    if (parent->profundidad % 2 == 0){
      quickSort(parent->segmentos, 0, parent->largo_segmentos-1, true);
    }
    /* Profundidad impar */
    else{
      quickSort(parent->segmentos, 0, parent->largo_segmentos-1, false);
    }

    /* Creacion recurisva de los hijos */
    parent->hijo_izquierdo = crear_hijo(parent, true);
    parent->hijo_derecho = crear_hijo(parent, false);

    insertar_nodo(parent->hijo_izquierdo);
    insertar_nodo(parent->hijo_derecho);

  }
}
```

Figure 2: Función búsqueda

3 Complejidad total del programa

Para este análisis, va a depender de la cantidad de sementos (S), la cantidad de partícula (P) y la cantidad de frames (F) que se entregan como input, ya que se analiza en cada frame, para cada particula, en el árbol de segmentos creados previamente.

Dentro del código, las funciones más densas en complejidad son las de creación del árbol y búsqueda, donde dichas complejidades fueron calculadas de forma análitica más arriba en el documento. Se tienen como resultado para la inserción O($S^2log2(S)$) y para la búsqueda O($log_2(S)$). Ahora bien, la búsqueda se realiza una para cada frame y dentro de cada frame para cada particula por lo que esta queda como $O(FPlog2(S))$.

Con respecto a lo anterior, tenemos tres casos:

1. S^2 = FP: Este escenario solamente se da Si la cantidad de frames y particulas es igual a las de segmentos, al ocurrir esto, el peor caso de la complejidad total del programa es O($FPlog_2(S)$) =

O($S^2 log_2(S)$).

2. $S^2 >$ FP: En este escenario se tiene que la creación de nodos es más costosa que la misma ejecución del programa, por lo que la complejidad queda O($S^2 log_2(S)$.

3. $S^2 <$ FP: Este caso se tiene que la creación de nodos es menor en complejidad que la ejecución del programa, por lo que la complejidad final del código sería: O($FP log_2(S)$).

   El resto de las operaciones se omiten, porque son "for" o "if" con cantidades O(a) con a constantes, por lo que se desprecian en el análisis por simplicidad.

4 Comparación empírica de las velocidades de ejecución

Mediante el siguiente gráfico queda detallado el tiempo de demora de la implementación inicial O(FPS) en compraración a la implementación final O($FP log_2(S)$). De las ilustraciones, se comprueba que la nueva implementación es mucho más eficiente que la original.

Test	O(FPS)	O(FP log(S))
1	32,507	0,783
2	37,802	0,860
3	40,172	0,856
4	115,394	1,087
5	123,651	1,059
6	136,453	1,345
7	121,096	1,533
8	187,473	1,953
9	187,96	2,108
10	285,576	3,010
11	298,989	2,682
12	516,042	5,482
13	610,229	4,504
14	701,633	4,098

Figure 3: Tabla de valores

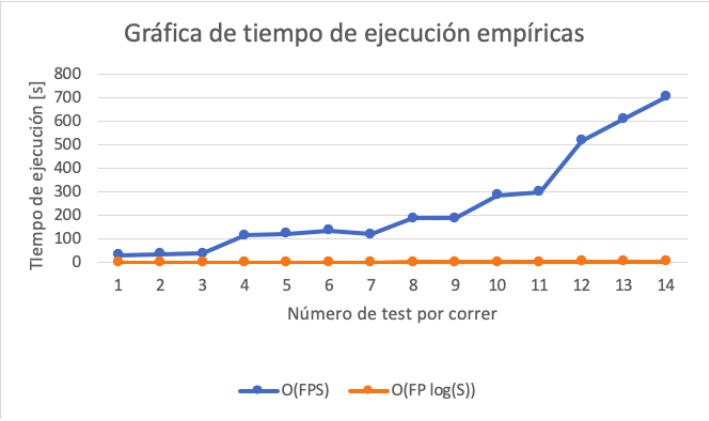

Figure 4: Gráfica comparativa