Assignment 8

Avijit Ghosh
14CH3FP18

March 16, 2016

Abstract

Event-driven Simulation of random particle collisions using heaps.

1 Introduction

This program simulates the collision of balls on a 2D planar region bounded by straight walls using a Heap Data Structure.

2 Event Driven Simulation

We need to devise the most efficient use of the Heap data structure to simulate the collision of particles using Event driven simulation.
Event driven simulation allows us to simulate events in a chronological order.

3 Implementation

The implementation of the is described as below:

Particle The particle structure describes a sphere. It has a radius, initial x, y coordinates and initial vx, vy velocity values. Each particle also has a color. All these are randomly assigned.

List Each particle also has a linked list attached to the particle. The linked list keeps track of all the valid collisions that the particular particle has participated in. This list is later purged to obtain the data for our plot.

Collision Heap The Heap structure is what maintains the entire event data for the system. Using equations of motion we predict the next event and insert it into the heap. The heap is a min-heap, which means that the earliest event is at the top. The heap is implemented as a very large array, with data members being Particle A and B (the two particles that take part in collision). In case the collision is with a wall, one of them is null.
The main loop runs until the heap is empty and the prediction function is called until the logical time is greater than or equal to the intended time of simulation.

Validity Check The validity of a collision event is checked as follows:

```
isValid(CollisionHeap A)
    if (A.a != NULL and count(A.a) != A.countA)
        return false;
    if (A.b != NULL and count(A.b) != A.countB)
        return false;
    return true;
```

Insertion Pseudocode for insertion of events:

```
void Insert(CollisionHeap H[], CollisionHeap Element)

    i=0;
    heapsize++;
    H[heapsize]←Element;
```

```
now←heapsize;
while(H[now/2].timestamp > Element.timestamp)
    H[now]←H[now/2];
    now/=2;
H[now]←Element;
```

Minimum Element extraction and Deletion Pseudocode for extracting the next event and deleting it from the Heap:

```
CollisionHeap NextDelete(CollisionHeap H[])

CollisionHeap minElement,lastElement;
int child,now;
minElement ← H[1];
lastElement ← H[Heapsize–];

for(now = 1; now*2 <= Heapsize ;now = child)
    child ← now*2;
    if(child not equals Heapsize and H[child+1].timestamp < H[child].timestamp
)
        child++;
    if(lastElement.timestamp > H[child].timestamp)
        H[now] ← H[child];
    else
        break;
H[now] ← lastElement;
return minElement;
```

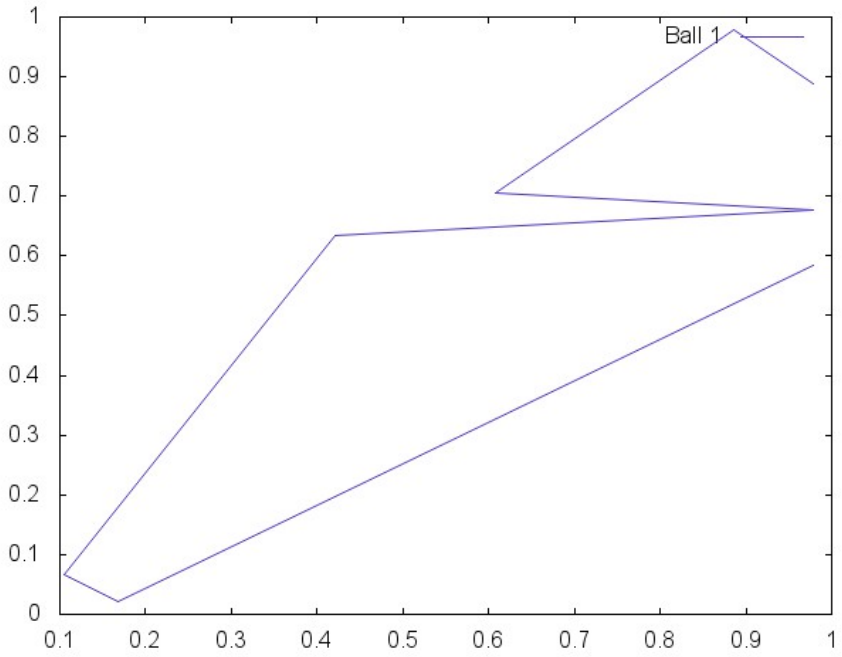
3.1 Complexity Analysis

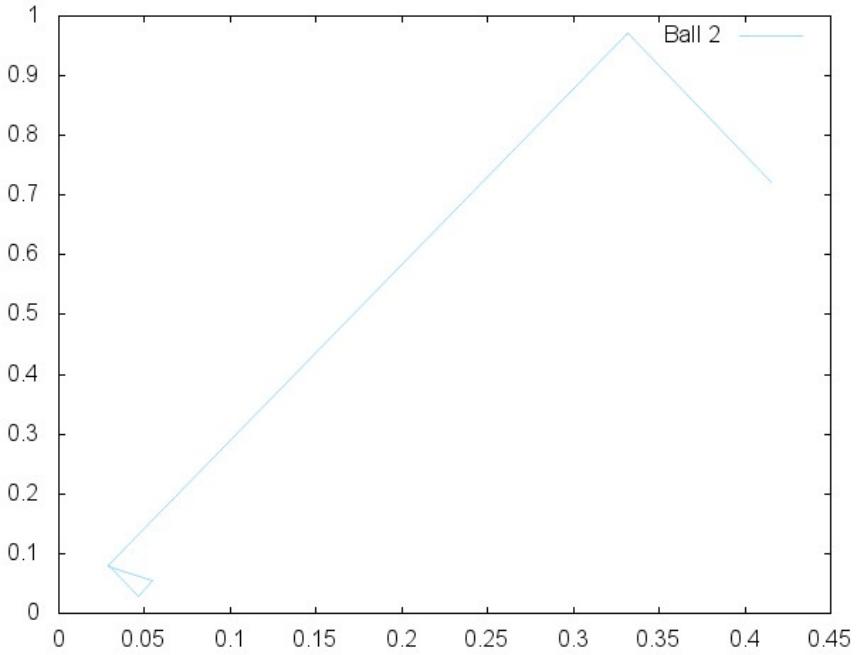
Insertion Insertion requires adding an event to its specific linked list followed by updating the heap upwards from the linked list towards the root. This operation

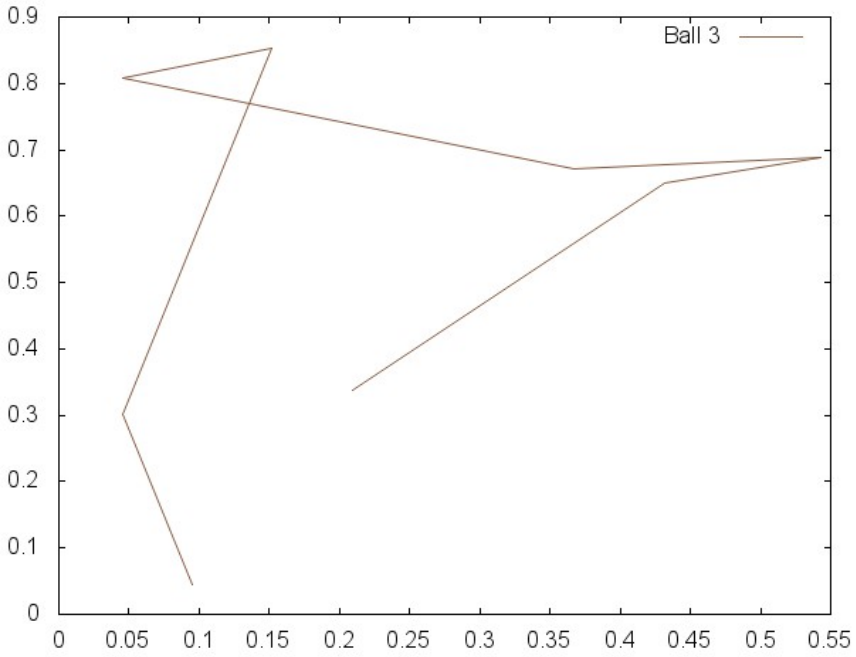
consumes $\theta(log(n))$ number of operations, where $\mathbf{n}$ is the number of particles in the system.

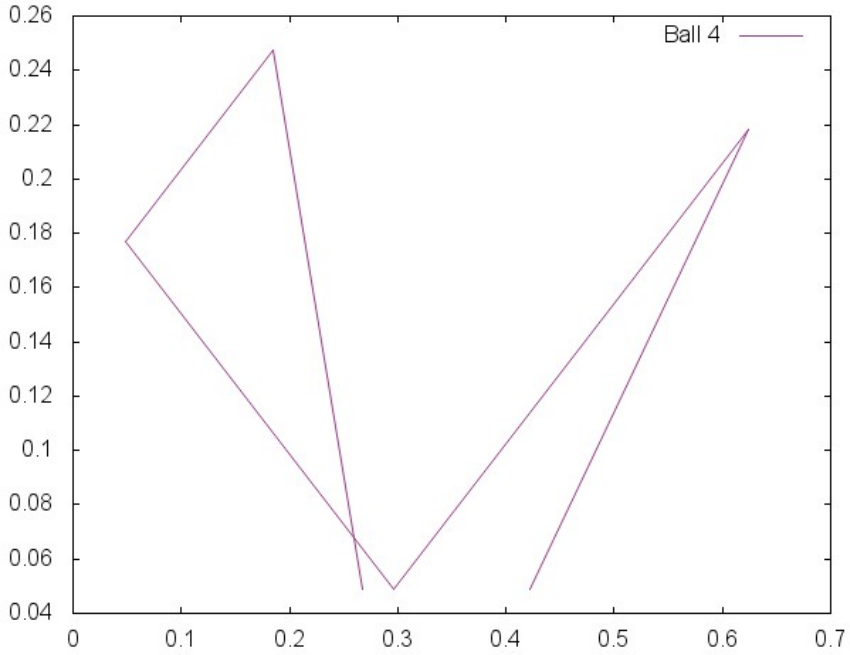
Extraction of minimum and deletion The minimum element is at the top due to the heap property, so obtaining it takes $\theta(1)$ time.
Now we take the max element and put it at the top, then recursively swap it with its children until the heap property is restored.
In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or $\theta(log(n))$

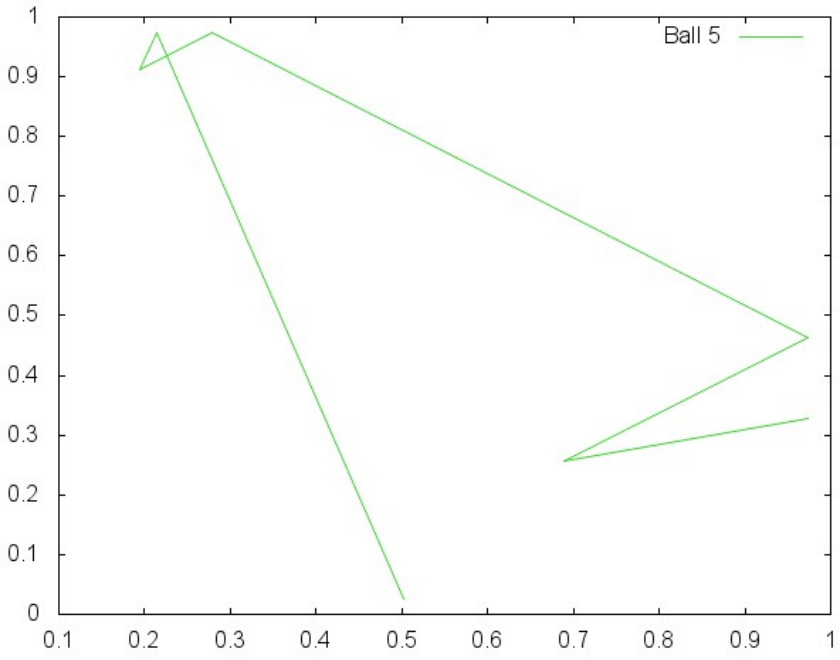
4 Plots




Ball 2
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45


Ball 3
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.55


Ball 4
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7


Ball 5
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1