2D Balls Simulation

Shashwat Gupta (14IE10028)

March 14, 2016

1 Data Structures

The following is the code for my data structures used in the program

```c
typedef struct _Ball //State
{
    double tstamp;
    int id;
    double px, py, vx, vy, radius;
    int colour;
}Ball;

typedef struct _Interaction
{
    double tstamp;
    Ball *interactee;
    Ball *interactor;
    int interactor_collision_count;
}Interaction;

typedef struct _LinkedList
{
    Interaction *event;
    struct _LinkedList *next;
    struct _LinkedList *prev;
}LinkedList;

typedef struct _Heap
{
    int *node;
    LinkedList **list;
    int *collision_count;
}Heap;
```

Listing 1: Structures

2 Initialisations of the data structures

1. Ball State The following is the codes for the initialisation of the State either randomly or parameterised. The two functions on being called create the Ball

State. There is another function for sending the random number in the given ranges and another function to send the colour to the Ball State.

The following is the Code

```
Ball *initBallRandom(int id)
{
    Ball *ball = (Ball *)malloc(sizeof(Ball));
    ball->id = id;
    ball->tstamp = sim_time;
    ball->px = randomNum(0.1, 0.9); //taking between 10 cm to 90 cm
    ball->py = randomNum(0.1, 0.9); //taking between 10 cm to 90 cm
    ball->vx = randomNum(-0.5, 0.5);
    ball->vy = randomNum(-0.5, 0.5);
    ball->radius = MIN_RADIUS;
    ball->colour = getColour();
    return ball;
}
Ball *initBall(int id, double px, double py, double vx, double vy,
    double radius, int colour)
{
    Ball *ball = (Ball *)malloc(sizeof(Ball));
    ball->id = id;
    ball->tstamp = sim_time;
    ball->px = px;
    ball->py = py;
    ball->vx = vx;
    ball->vx = vy;
    ball->radius = radius;
    ball->colour = colour;
    return ball;
}
```

Listing 2: left

2. Linked List is initialised using the Event parameter passed to it. The following is the Code

```
LinkedList *initLinkedListObject(Interaction *event)
{
    LinkedList *linkedListObject = (LinkedList *)malloc(sizeof(
    LinkedList));
    linkedListObject->event = event;
    linkedListObject->next = NULL;
    linkedListObject->prev = NULL;
    return linkedListObject;
}
```

Listing 3: Right

3. Heap The following is the Code for the Heap initialisation. It also initialises the variables of the structure. It creates ParticleCount number of variables within the structure.

```
Heap *initHeap()
{
    int i;
    Heap *heap = (Heap *)malloc(sizeof(Heap));
    heap->node = (int *)malloc(sizeof(int) * PARTICLE_COUNT);
```

```
    heap->list = (LinkedList **)malloc(sizeof(LinkedList *) * PARTICLE_COUNT);
    heap->collision_count = (int *)malloc(sizeof(int) * PARTICLE_COUNT);
    for(i = 0; i < PARTICLE_COUNT; i++)
    {
        heap->collision_count[i] = 0;
        heap->list[i] = NULL;
        heap->node[i] = -1;
    }
    return heap;
}
```

Listing 4: Right

3 Heap Functions

1. Insert

The following is the Code for inserting into the Heap. The code checks for certain cases and inserts accordingly.

```
int insertToHeap(Heap *heap, Interaction *event)
{
    if(heap == NULL || event == NULL)
        return -1;
    LinkedList *linkedListObject = initLinkedListObject(event);
    if(heap->list[event->interactee->id] == NULL)
    {
        heap->list[event->interactee->id] = linkedListObject;
        linkedListObject->next = NULL;
    }
    else
    {
        if(heap->list[event->interactee->id]->event->tstamp > event->tstamp)
        {
            linkedListObject->next = heap->list[event->interactee->id];
            heap->list[event->interactee->id] = linkedListObject;
            linkedListObject->next->prev = linkedListObject;
        }
        else
        {
            linkedListObject->next = heap->list[event->interactee->id]->next;
            linkedListObject->prev = heap->list[event->interactee->id];
            if(linkedListObject->next != NULL)
                linkedListObject->next->prev = linkedListObject;
            heap->list[event->interactee->id]->next = linkedListObject;
        }
    }
    return 0;
```

```
}
```

Listing 5: delete

2. Delete

The following is the Code for extraction and nullification of an event from the Heap

```
int removieFromHeap(Heap *heap, int interactee)
{
    if(heap == NULL)
        return -1;
    heap->list[interactee] = NULL;
    heap->collision_count[interactee] = heap->collision_count[interactee] + 1;
    return 0;
}
```

Listing 6: deleteFix

4 Interaction Functions

The following is the Code for getting the next event.

```
Interaction *getNextEvent(Heap *heap)
{
    int i, min_index;
    Interaction *nextEvent;
    while(1)
    {
        for(i = 0; i != PARTICLE_COUNT; i++)
        {
            if(heap->list[i] != NULL)
            {
                nextEvent = heap->list[i]->event;
                min_index = i;
                break;
            }
        }
        if(min_index == PARTICLE_COUNT)
            return NULL;
        for(i = min_index + 1; i != PARTICLE_COUNT; i++)
        {
            if(heap->list[i] != NULL)
            {
                if(heap->list[i]->event->tstamp < nextEvent->tstamp)
                {
                    nextEvent = heap->list[i]->event;
                    min_index = i;
                }
            }
        }
        if(nextEvent->interactor != NULL && nextEvent->interactor_collision_count != heap->collision_count[nextEvent->interactor->id])
```

```
            heap->list[min_index] = calcMinima(heap, heap->list[min_index]);
        else
            return nextEvent;
    }
}
```

Listing 7: process create

The following is the Code for calculation of the minima.

```
LinkedList *calcMinima(Heap *heap, LinkedList *list)
{
    if(list == NULL)
        return NULL;
    while(list && list->event->interactor != NULL && list->event->interactor_collision_count != heap->collision_count[list->event->interactor->id])
        list = list->next;
    LinkedList *minima = list;
    LinkedList *head = list;
    while(list != NULL)
    {
        if(list->event->interactor != NULL)
        {
            if(list->event->interactor_collision_count != heap->collision_count[list->event->interactor->id])
            {
                if(list->prev != NULL)
                    list->prev->next = list->next;
                if(list->next != NULL)
                    list->next->prev = list->prev;
                continue;
            }
        }
        if(list->event->tstamp < minima->event->tstamp)
            minima = list;
        list = list->next;
    }
    if(minima == NULL)
        return minima;
    if(minima->prev != NULL)
        minima->prev->next = minima->next;
    if(minima->next != NULL)
        minima->next->prev = minima->prev;
    while(head == minima)
        head = head->next;
    minima->next = head;
    minima->prev = NULL;
    return minima;
}
```

Listing 8: process schedule

The following is the Code for Ball Collisions

```
Interaction *eventBallCollide(Heap *heap, Ball *ball_i, Ball *ball_j)
{
```

```
    if(ball_i == ball_j)
        return NULL;
    double relPX = ball_j->px - ball_i->px;
    double relPY = ball_j->py - ball_i->py;
    double relVX = ball_j->vx - ball_i->vx;
    double relVY = ball_j->vy - ball_i->vy;
    double relVrelP = relPX * relVX + relPY * relVY;
    if(relVrelP >= 0)
        return NULL;
    double relVrelV = relVX * relVX + relVY * relVY;
    double relPrelP = relPX * relPX + relPY * relPY;
    double sigma = ball_j->radius + ball_i->radius + DELTA;
    double d = (relVrelP * relVrelP) - relVrelV * (relPrelP - sigma
    *sigma);
    if(d < 0)
        return NULL;
    double timeToCollision = -(relVrelP + sqrt(d)) / relVrelV;
    if(timeToCollision > 10)
        return NULL;
    Interaction *collisionEvent = (Interaction *)malloc(sizeof(
    Interaction));
    collisionEvent->tstamp = sim_time + timeToCollision;
    collisionEvent->interactee = ball_i;
    collisionEvent->interactor = ball_j;
    collisionEvent->interactor_collision_count = heap->
    collision_count[ball_j->id];
    return collisionEvent;
}
```

Listing 9: process schedule

The following is the Code for wall Collisions

```
Interaction *eventWallCollideY(Heap *heap, Ball *ball)
{
    if(ball == NULL)
        return NULL;
    double timeToCollision;
    if(ball->vy > 0)
        timeToCollision = (1.0 - ball->radius - ball->py)/ball->vy;
    else
        timeToCollision = fabs((ball->py - ball->radius)/ball->vy);
    if(timeToCollision > 10)
        return NULL;
    Interaction *collisionEvent = (Interaction *)malloc(sizeof(
    Interaction));
    collisionEvent->interactee = ball;
    collisionEvent->interactor = NULL;
    collisionEvent->tstamp = sim_time + timeToCollision;
    collisionEvent->interactor_collision_count = heap->
    collision_count[ball->id];
    return collisionEvent;
}
Interaction *eventWallCollideX(Heap *heap, Ball *ball)
{
    if(ball == NULL)
        return NULL;
    double timeToCollision;
```

```
    if(ball->vx> 0)
        timeToCollision = (1.0 - ball->radius - ball->px)/ball->vx;
    else
        timeToCollision = fabs((ball->px - ball->radius)/ball->vx);
    if(timeToCollision > 10)
        return NULL;
    Interaction *collisionEvent = (Interaction *)malloc(sizeof(
    Interaction));
    collisionEvent->interactee = ball;
    collisionEvent->interactor = NULL;
    collisionEvent->tstamp = sim_time + timeToCollision;
    collisionEvent->interactor_collision_count = heap->
    collision_count[ball->id];
    return collisionEvent;
}
```

Listing 10: process schedule

5 Time Complexity Calculation

1. Insert into Heap

Time complexity has the recursive relation:

$$T(n) = T(n/2) + \theta(1)$$

where n is the number of nodes. Solving this equation by recursive method yields T(n) $\in O(logn)$.

2. Extract from Heap

Time complexity has the recursive relation:

$$T(n) = T(n/2) + \theta(1)$$

where n is the number of nodes. Solving this equation by recursive method yields T(n) $\in O(logn)$.

3. Simulation Run

A ball will collide with another ball or a wall. When a collision occurs, there is extraction of event from heap and insertion of new events in the heap.

Thus, the complexity yields T(n) $\in O(logn)$.