1 Sweep and Prune Collision Detection Algorithm

Particle simulations with only near-field effects have two major characteristics that can be exploited when optimizing for performance:

1. Spatial Locality: A particle only interacts with its nearby neighbours.
2. Temporal Locality: If two particles are nearby in the current timestep, they are likely to be nearby in the next timestep.

Sweep and Prune is a collision detection algorithm that exploits temporal locality. The algorithm internally maintains two sorted lists, one list containing the particles sorted by their x coordinates, and another list containing the same particles but sorted by their y coordinates. Particles that are distant with regards to their list position are unlikely to be colliding. Furthermore, if particles have not moved too great a distance since the last timestep then the lists will remain *nearly* sorted. A sort routine with complexity proportional to the orderedness of the list, such as insertion sort, is then used to keep the lists sorted efficiently.

1.1 Algorithm Initialization

Here, the simple case of collision detection in 1D is used to explain the basic algorithm. Generalizing the algorithm to N dimensions is straightforward and discussed later. The algorithm initially starts with an initially empty list, L. For each object o_i, we add two tokens, MIN_i and MAX_i, to the end of L, where MIN_i is the tuple $(\text{MIN}, i, x_{\text{i,min}})$ and MAX_i is the tuple $(\text{MAX}, i, x_{\text{i,max}})$. MIN and MAX are labels that indicate the type of the token. $i \in \{1, \ldots, N\}$ is the index of the object. $x_{\text{i,min}}, x_{\text{i,max}} \in \mathbb{R}$, are the x coordinates of the left-hand and right-hand boundaries of the object respectively. The algorithm also maintains a mapping M from unordered pairs of indices to booleans, $M : (i, j) \in \{1, \ldots, N\}^2 \to \text{bool}$, that indicates whether object i and j are intersecting. M is initially false for every pair i and j.

1.2 Update Step

To detection collisions, we use an insertion sort to sort L according to the x coordinates of the tokens.

```
for $k$ in $0 \ldots 2N$ do :
  $m := k$
  while $m > 1$ :
    if $L_{m-1}.x < L_m.x$ :
      swap$(m, m-1)$
      $m = m - 1$
    else :
      break
```

where the function *swap* interchanges the positions of token m and token $m-1$, and updates the mapping M. If the MAX token of object i is swapped to the right of the MIN token of object j then we set $M(i,j)$ to true, to indicate that object i is now intersecting object j. If the MIN token of object i is swapped to the right of the MAX token of object j then we set $M(i,j)$ to false to indicate that object i is no longer intersecting object j.

```
define swap $(i, j)$ :
  if $L_i$.type = MIN and $L_j$.type = MAX :
    $M(i, j)$ = true
  else if $L_i$.type = MAX and $L_j$.type = MIN :
    $M(i, j)$ = false
  interchange $L_i$ with $L_j$
```

At the end of the sort stage, and L is properly sorted, $M(i,j)$ will indicate whether the bounding boxes of object i intersects with object j. The very first timestep will take $O(N^2)$ time because there is no guarantee on the initial ordering of L. But for all subsequent timesteps, if the timestep is small and particles do not move excessively, the sort can be completed in close to $O(N)$ computations. As a further possible optimization, a quick $O(n\log n)$ sort can be used to obtain the initial ordering for L.

1.3 Generalization to N-D

To generalize the single dimensional sweep and prune algorithm to N dimensions, we now maintain N sorted lists. We note that the bounding boxes of objects i and j are overlapping if and only if they are overlapping in all N dimensions. Thus at each timestep we perform N sorts, and check that $M(i,j) = \text{true}$ for each dimension.

2 PThreads implementation of Sweep and Prune

The Sweep and Prune algorithm maps well onto sequential hardware but parallelization across multiple processors is difficult. We opt to parallelize sweep and prune by partitioning the simulation space into separate regions and assigning each region to a separate processor. Within a single processor we use the sequential sweep and prune algorithm.

2.1 Regions

A region structure contains fields that indicate the extents of its boundaries, $\text{min}_x, \text{max}_x, \text{min}_y, \text{max}_y$, pointers to its neighbouring regions, and a local sweep and prune structure for managing the particles local to the region. Particles may only be added or removed from a region using the *add_region_particle* and *remove_region_particle* functions.

2.2 Region Bounds

Regions have several bounds for the purposes of partitioning the workload amongst processors. $\min_x, \max_x, \min_y, \max_y$ define the *responsibility bounds.* A region is responsible for correctly computing the forces and updating the positions of all particles within its *responsibility bounds.* The *send bounds* of a region starts at a distance of *cutoff* interior from the border and continues to the border. Any particles within the *send bounds* need to have its position sent to the neighbouring regions in order for the regions to accurately compute collisions along boundaries. The *ghost bounds* of a region starts at the border and continues to a distance of *cutoff* exterior to the border. The region will receive the positions of particles within the *ghost bounds* from neighbouring regions to use for handling collisions along the boundaries.

2.3 Algorithm

Every region is run in its own thread of execution, and at every time step :

1. Determines which particles are in the *send bounds* and need to be sent to their neighbours.
2. Sends the particles to its neighbours. Particles are sent to each neighbour (N, NE, E, SE, S, SW, W, NW) in turn, with a synchronizing barrier after each. This barrier ensures that a region is receiving from only one thread at a time.
3. Removes any particles that are outside its *responsibility bounds.*
4. Computes forces and updates the positions of all particles within its *responsibility bounds.*

The algorithm is summarized as :

```
for timestep in 1...NSTEPS do :
  send_particles := in_sending_bounds(all particles)
  barrier
  for neighbour in [N, NE, E, SE, S, SW, W, NW] do :
    send_to_neighbour(neighbour, send_particles)
    barrier
  remove_particles_out_of_responsible_bounds
  barrier
```