Ramez Nahas
I.D#: 26718108
Assignment 1 Report October 7, 2019

Overview:

The design of my control and worker threads was done by splitting two task, computation and update, between them.

Computation involves detecting ball-wall and ball-ball collisions, fixing the overlap (by moving the ball center), and computing the new velocities. It is the more time-consuming task and must be done for each ball, so for this reason my worker threads handled this task. I created one worker thread per ball.

The task of updating (i.e: drawing) the screen with the new frame, and limiting that update to 30 fps, was done by the main thread (the control thread). This thread is responsible for spawning the worker thread and notifying them when to compute the new frame. It then draws the new frame.

Thread synchronization:

For thread synchronization, I created a “barrier” class. This class is responsible for making all threads wait until every other thread is at the same stage. I used 3 different barriers, wall_computation, do_frame and frame_done.

The wall_computation is a worker-threads-only barrier that synchronizes computation between workers. Basically, all workers must be done doing the ball-wall collisions computation before starting the ball-ball collisions, as ball-wall involves center and velocity changes. The wall_computation barrier enforces this.

The do_frame and frame_done barriers are all-threads barriers (control and workers). The idea behind them is the following: the workers cannot start the computation of the new frame before the control tells them to, and this is done through the do_frame barrier. Then, the control cannot draw the new frame until the new frame is done computing. This is done though the frame_done barrier.

All this leads to the following pipeline: workers wait for control to give them the order to compute the new frame → order is given → workers starting computing the new frame and control waits for the new frame → workers synchronize between them for the ball-wall and ball-ball computation → workers signal new frame is done → control draws new frame → loop.

Thread optimization:

For optimization, I did two things that both involve ball-ball collision computation.

First, instead of looping *every frame* through every ball and checking for every other ball if there is a collision (an $O(n^3)$ operation every single frame), I do this operation once in the control thread, during initialization of the program. What I do is create a “std::vector<std::pair<ball*, ball*>> pairs”, which represents every unique pairs possible with the given balls. Now that my pairs are cached, it is simply a matter of splitting the vector between the different worker threads, so that each worker thread only works on a portion of the vector. This greatly optimizes the ball-ball computation, since the operation is now essentially O(1) per frame.

Second, I use the axis-aligned bounding-box (aabb) optimization. The idea is to check if two balls are close enough to go ahead with the actual ball-ball collision computation. If two balls are far away, aabb will see that, and then we don’t need to do ball-ball collision computation. If two balls are close, aabb will signal that, and then we will check if an actual collision happened, and if so, do the rest of the computation.