University of Cape Town - CSC2002S

PCP 2 Assignment Report

By Kelvin Wei

Synchronization Mechanisms

Data Races

Grid Block

In the program, GridBlock was the shared object that is vulnerable to data races. Therefore, following the monitor pattern, all methods that access the state of GridBlock are synchronized. This satisfies simulation rule (3). Using the monitor pattern wouldn't compromise liveliness since it's only enforced on an individual block which is not likely to be a commonly contested resource.

Additionally, an AtomicInteger is used for the synchronization of the order of swimmer team entry. This prevents another potential data race on a shared object.

Simulation Rules & Synchronization Mechanisms

CountDownLatch: predetermined number of event calls would release the threads waiting on the latch.

CyclicBarrier: threads waiting on barrier would release when the number of threads waiting is equal to a predetermined number.

Conditional Variables: wait() and notify() threads to prevent unnecessary resource usage and encourages checking only when condition to proceed is most likely met.

Note: (N) - shows how condition number N has been satisfied, of which there are 9.

MedleySimulation (Main Method)

Enforcing the start button (1) was done by adding a Latch for Start Button in the Main Method. It's appropriate since we want the simulation to start once the button has been

pressed and to continue running until the quit button is pressed or the race finished. (2) already satisfied.

Barrier added to enforce swimmers to wait at the starting block for all other swimmers that are swimming the same stroke in other teams. Swimmers are only allowed to start swimming once all swimmer's of the same stroke are ready (7). This barrier is passed to SwimTeam and to Swimmer Thread where the thread potentially waits.

Stadium Grid

In 'Stadium Grid' conditional variables were implemented on 'Grid Block objects' to wait & notify potential relevant threads. Such mechanisms were added to 'moveTo', 'enterStadium' (5.1) and 'jumpTo' methods.

Adding conditional variables for contested blocks prevents spinning with the while loop. This ensures the liveliness of other threads in the program, as eliminating the spinning opens up the resources to other threads (4)

Additionally, when a block is free due to a swimmer moving away from the block, the swimmer notifies any threads waiting on that block.

Swimmer

A barrier `teamArrivedBarrier` was added to ensure that swimmer threads only attempt to enter when all their team members have arrived - due to variable arrival times.

The mechanism that ensures that swimmers enter in order is facilitated by an Atomic Integer (`toEnterOrder`), type chosen as stated before to prevent data races. All swimmers of the same team would need to access the same `toEnterOrder` lock. The swimmers would need to wait if it's not their turn. The swimmer requested would then enter, increment the integer and notify swimmer threads waiting to request for the next swimmer to enter. (5.2)

Swimmers line up in the correct order due to not being allowed to overtake another swimmer (they have to wait if nextBlock to move to is occupied) and having the same destination (6). A barrier is added to ensure all swimmer's of the same stroke have arrived at the starting block before diving. This means that the race starts when all backstroke swimmers (first to enter) are at the starting point. (7)

Additionally each swimmer has a latch to their team member who has a swim order above them and their own latch. Swimmers that finish would release the latch for the next swimmer in their team to start swimming. The backstroke swimmer would have their latch pre-released.

SwimTeam

Initializes the barriers, latches and Atomic Integer required for each swimmer thread.

Extensions

Tracking 1st, 2nd, 3rd Team winners and Timer

The program has been modified to track and display the winning teams in a podium format. The program would then display a podium animation when all swimmers have finished. This gives a satisfactory closure to the simulation. Additionally a timer has been added to the top of the swimmer simulation. It turns green when recording the time and red when it's not recording. Implemented in a class called "Timer"

Showing Fractal animations

The program implements fractal-like animations by using recursive methods. The `branch` method draws tree-like structures by recursively reducing the length and stroke width at each step, creating a branching effect. Similarly, the `drawCircles` method recursively draws concentric circles, creating a fractal pattern of circles that changes with each frame.

The animations are dependent on the value of the `rotation` variable which stores an angle. This determines the angle at which the tree's branches diverge and the angle at which the circles are rotated by.

This animations were implemented in a class called "PodiumStand"

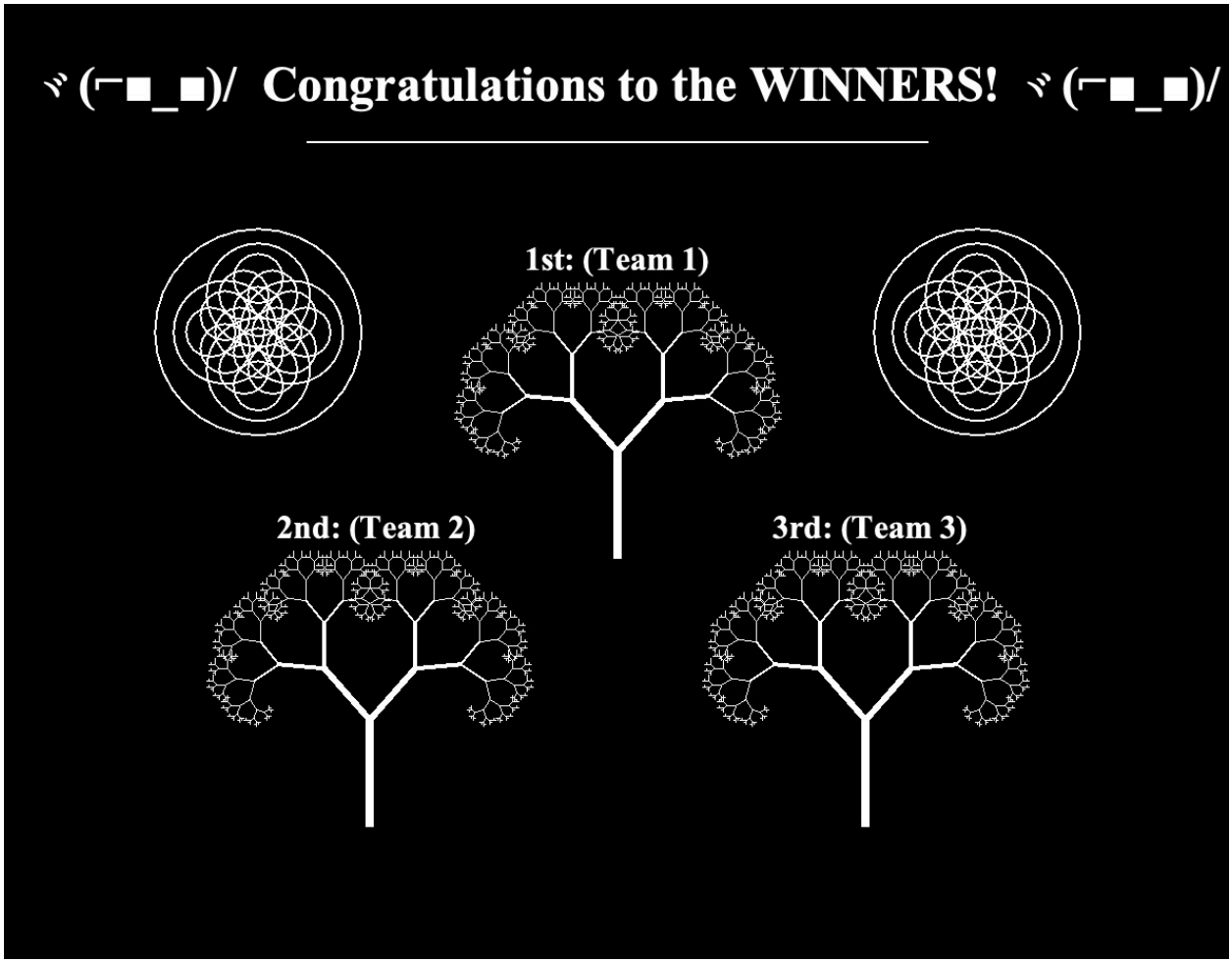
Lerp functions - assist with animations

Linear interpolation (Lerp) functions are used to create smooth motions of objects across set distances. The Lerp function has been modified to additionally use an inverse lerp

function to have objects moving according to a `rotation` value instead of a time value in between 0 and 1.

Implementing a Game Loop

Implementing a game loop helps with performance and displaying consistent and smooth animations. The game loop helps maintain a consistent frame rate of 30 fps. Without the game loop, the program will try to update the game as fast as it can. This is wasted resources and would make the animations move faster/slower depending how fast the computer running the program is.



Podium_GUI screenshot