Introduction

The aim of this assignment was to create a multithreaded typing game, making use of the various methods learnt in lectures in order to create a thread-safe, concurrent game. The game described below makes use of appropriate synchronization and threading achieve this.

Classes

The following classes were added to the game:

Controller

This class is a controller class that deals with the scoring in the game. It has two methods, one that checks the words to see if any of them have been missed (missWord()) and another that compares a text parameter against the words, intended to compare text entered in the JTextField (matchWord(String text)). It implements Runnable and in its run() method, it loops, checking for any missed words. There is a boolean flag to request the loop to stop and the Thread to return.

WordUpdater

This class is a controller class that updates the WordRecords in the game. It animates the words by updating their positions. It doesn't draw them, the drawing still happens in the WordPanel. It implements Runnable and in its run() method, it loops, updating the word positions. Each word, when updated, is given a time when it was last updated so that they can be independently and more accurately updated (over simply putting the Thread to sleep). The method that drops the words (dropByLastTime(long lastLoop)) is explained further on. There is a boolean flag to request the loop to stop and the Thread to return.

ParticleUpdater

This class is a controller class that updates the Particles in the game. It animates the particles by updating their positions. It doesn't draw them, the drawing still happens in the WordPanel. It implements Runnable and in its run() method, it loops, updating the particle positions and removing old particles. As with the word updates, the particles are updated based on how much time has passed since the last update. There are also two methods (explode() and implode()) that create bunches of particles at a specified location. There is a boolean flag to request the loop to stop and the Thread to return.

Particle

A repurposed class that models particles for additional feature. When words are caught, or missed, an explosion of particles is made. The code that was used had to be adjusted to be used in this concurrent game, mainly by synchronizing many of the methods. The particles were originally kept in an ArrayList, but because there were concurrent accesses to it, a CopyOnWriteArrayList was used instead which worked with the concurrent accesses.

Vector

A helper class for the Particle class used for vector calculations on the particles. This was also reused code from another program.

Significant Modifications

WordRecord

In the WordRecord class, dropByLastTime(long lastLoop) is used over the default drop(int increment) method. This was done so that the words could be updated based on how much time had based, incase each call to update happened at different intervals. While probably not essential, it did make the game look much smoother over using Thread.sleep(), as well as keeping the words falling at the same speed. The method calculates how many pixels to move based on the time passed and the speed of the word. The y-coordinate is the updated and the remaining time not accounted for in the discrete pixel movement is accumulated for future updates. This ensures that is updates are too frequent, their time is still used in a future update so that, for instance, half a pixel isn't lost every few updates.

Also added was a retireWord() method which is similar to the resetWord() method although it sets the word speed to 0, sets the text to an empty string and sets it to not falling. This method is used when the game is in its end stages and not all the words can fall.

WordPanel

The WordPanel class now renders the words as well as the particles in its paintComponent(Graphics g) method. It also starts the WordController and ParticleController Threads. It has a boolean flag to request the loop to stop and the Thread to return.

Score

The getters and setters in Score needed to be synchronized, however AtomicIntegers were used instead as these are faster. Some methods had two or more atomic accesses and so were synchronized.

WordApp

The WordApp class creates the Controller Thread for tracking the score and the WordPanel Thread for painting the game. It then loops on updateScore() which updates the score JLabels and checks if all of the words have been caught or missed, ending the game. The methods restartGame() and stopGame() were also added.

Some Validation

Getters and setters in all classes were either synchronized or their variables were made atomic. No two methods require each others lock simultaneously.

The boolean flags across the controllers are made volatile so that a fresh value is always available when checking in the loop. This ensures that the thread ends soon after it is asked.

By updating each word with its own time and accumulating the “unused” time, they each update by the amount that they need. An earlier implementation had a single time difference applied to all of the words which resulted in the fastest word moving by a decent amount (distance = speed * time) with the others moving extremely slowly, if at all.

Updating by time over simply putting threads to sleep also has the benefit of reducing the effects of starvation by updating words and particles to where they need to be at a time rather than just by some amount. Even if an update takes relatively long, the animation will appear consistent and smooth. Unfortunately however, if it takes exceptionally long, the user might experience a lag and then a jump rather than simply slow performance which could cost them the game. Events like this were not observed in testing, as this game is relatively light and they are unlikely to happen.

Model-View Controller Pattern

Model

WordDictionary, Score, WordRecord, Particle and Vector are model classes. These classes contain and model the data used in the game such as the score, the words and the particles.

View

WordPanel is the view. It draws all of the words and particles to the screen without updating them.

Controller

Controller, ParticleUpdater, WordUpdater are the controller classes. These are the classes manipulating the data, updating the positions of the words and particles, catching and missing words.

WordApp doesn't really fit into one of these categories. It sets up the game and then monitors for button presses and text entries. So it could be thought of as a controller but also as part of the model.

Additional Feature

An additional feature present is the particles that are created when a word is created or missed. The code for this was repurposed from another program and changed to be used in a concurrent context, using the same features of synchronizing and locking to ensure mutual exclusion.

Conclusion

The typing game runs smoothly, without locking and by delegating tasks to different threads, can take advantage of multicore systems.