Implementation of Dynamic Plotting for a Ventilation Simulator on Android Mobile Devices

T. Lehmann¹, T. Nguyen², K. Moeller³
¹, ², ³Furtwangen University, Department of Mechanical and Process Engineering, Schwenningen, Germany,
t.lehmann@hs-furtwangen.de
²Thu.Nguyen@hs-furtwangen.de
³moe@hs-furtwangen.de

Abstract
A ventilation simulator is implemented in android mobile devices to simulate the mechanics of the human lung. The simulation results shall be displayed by a dynamic plot to illustrate the trend of the ventilation parameters in real-time. The chosen software library to perform the plots is AndroidPlot [1]. The library provides basic features and tutorials for dynamic plotting. Tests were performed to find out whether dynamic plotting with dynamic speed and in real-time is possible and feasible with AndroidPlot. The tests consist of temporal analyses of redraw durations for the two used redrawing methods. Both methods use a separate thread to redraw the plot in fixed time intervals or to pass a signal which indicates a necessity for redrawing the plot.

The tests showed that dynamic plotting with static speed and close to real-time is possible. Occurring limitations are the sampling frequency and domain width. These limitations should be dealt with to make the dynamic plotting more widely useable.

1 Introduction
Despite ventilation being the standard procedure in surgery with general anesthesia, it is a vital tool for intensive care to keep a patient alive [2]. Today, there is a vast variety of ventilation modes and settings which enable health professionals to make individual adjustments to fit the ventilation strategy to the situation of the patient. To support the decision making while choosing a ventilation strategy, mathematical models are used to simulate the mechanical behavior of the human lung in various detailing [2]. Hence, the simulation application which is being developed for tablet PCs aims to support decision making for health professionals.

The high mobility of tablet PCs is a major advantage for this goal, since decisions have to be made quickly and on the spot in many situations.

To visualize the trend of ventilation parameters like pressure, tidal volume and flow, a dynamic plotting tool has to be implemented. The speed of the data flow should be static in order to be able to display the data in real-time or as close as possible to real-time.

A promising software library for plotting is AndroidPlot [1]. Tests with the library shall show to which extend dynamic plotting is possible and feasible.

2 Methods
The used device for the tests was a Motorola XOOM (MZ601) tablet which runs the Android 3.2 operating system. The used plotting tool is AndroidPlot 0.5.0 because it is an actively developed and maintained open source project. The developers also provide tutorials for common plotting applications. Moreover, AndroidPlot is the only found library which includes the basic features for dynamic plotting. The provided tutorials for dynamic plotting are used as templates.
The data, which was used for the tests, was pre-recorded patient data. It was read from a file prior to plotting.

To make a plot dynamic, the data has to be re-rendered and redrawn in fixed time intervals. The data to draw has to be updated before each redraw.

The dynamic plotting procedure keeps all samples in place and appends new samples to the before drawn (see figure 1). Once the end of the graph is reached, it is cleared and plotting is restarted from the left side. The label of the abscissa is also updated when plotting restarts at the left side.

In figure 1, the duration on the abscissa was set to five seconds, which is approximately the ventilation cycle duration.

When a redraw needs to be performed, the method `redraw()` respectively `postRedraw()` of an instance of the class `Plot`, which extends `View`, is called. This causes the current plot to be invalidated. Invalidating a view always means that the view’s callback method `onDraw()` is called at some time in the future. In the class `Plot`, `onDraw()` contains the method calls for a complete redraw of the plot. This includes the before mentioned re-rendering, finding the minimum and maximum value in the data to plot, redrawing the grid and finally drawing the bitmap onto the screen.

The method `postRedraw()` must be called when the plot needs to be redrawn in a different thread (worker thread) than the User-Interface (UI) thread. It is not allowed to directly manipulate views that were created in the UI thread within a worker thread since this can result in unexpected events [3].

2.1 Redrawing within a worker thread

The redrawing iteration loop is run in a separate thread since iterating including redrawing does not work in the start button callback method of the User-Interface (UI) thread. As long as the program remains in the `onClick()` callback method, updating the UI thread is blocked [3]. By using `postRedraw()`, the redrawing is in a sense “posted” to the UI thread and marked for being executed at some time in the future.

To force the worker thread to wait until the redrawing is done, the Boolean parameter `isBlocking` can be assigned to `postRedraw()`. If `isBlocking` is set to “true”, the worker thread stays in `postRedraw()` until it is notified by `onDraw()` that redrawing has been completed. The effects regarding the plotting speed when setting `isBlocking"true"` or “false” will be explained in section 3.1.

2.2 Redrawing within the UI thread

The second approach uses the worker thread to signal the UI thread that the plot needs to be redrawn in fixed time intervals. To achieve the desired sampling frequency of the data, the worker thread is put to sleep for the appropriate time span. At the end of a sleep phase, the worker thread passes a message to the so called `Handler`. This procedure is repeated in a loop for the size of the data. Figure 6 illustrates the implementation of the procedure.

An object of the `Handler` class “receives” messages which were sent to it somewhere else in the program. Here, a `Handler` object is instantiated in the UI thread. It overwrites the method `handleMessage()`. Whenever a message is passed to the `Handler` object, `handleMessage()` is invoked and its contents are executed. The contents are updating the data to plot and are calling the method `redraw()` for each plot. Now the plotting view can be manipulated directly because it is initiated in the UI thread.
The advantages of this approach are demonstrated in section 3.

2.3 Data storage and updating
The data to plot is stored in a linked list, because this list type provides a fast way to shift or append data. The method `addLast()` is used to append data and `clear()` to clear the plot. As mentioned, in each iteration one sample is appended to the linked list (see figure 6, “Update data_to_plot”).

2.4 Time Measurement
To measure the execution duration of program segments, a class was implemented which records the current system time at the start and end of the desired segment. The start and stop points can be placed at any position in the code. The resulting duration can then be stored in a list to calculate the average and standard deviation of multiple durations. Since the system time can be measured in nanoseconds, a very high resolution is possible.

3 Results
Generally, there is for both approaches a certain duration for redrawing. The difference between the two applied approaches is perceivable as the plot is drawn.

3.1 Redrawing within a worker thread
In every iteration, the complete plot with all the containing data has to be redrawn. This means that the more samples are appended, the more data has to be rendered and consequently the redrawing duration increases.

Setting the parameter `isBlocking` to “true” means, as mentioned, that `postRedraw()` waits until all actions in `onDraw()` are performed. As illustrated in figure 5, this duration is not static in relation to the amount of drawn samples. Figure 2 shows the averages of intervals of 20 redraws for a domain size of 1600 samples. Here the time that is spent in `postRedraw()` while redrawing one plot was measured. While the program is waiting in `postRedraw()`, it cannot proceed and the data can not be updated.

The results when setting `isBlocking()` to “false” reduces the duration drastically (cf. figure 3). Figure 3 suggests that with a domain size of up to 400 samples, the added delay with every redraw stays relatively low. If 400 samples domain size are exceeded, the redraw duration rises faster.

3.2 Redrawing within the UI thread
The major difference to invoking a redraw in a worker thread is that there is vanishingly low time spent in `redraw()`. Therefore the program is not blocked by waiting until redrawing is done.

The duration for the actions performed in `onDraw()` is still the same and increases with growing samples to display (cf. figure 5). As long as the plot is being redrawn in `onDraw()`, a call of `redraw()` has no impact. This leads to the effect that the linked list, which stores the samples to be plotted (see figure 6, variable `data_to_plot`), is updated multiple times while the program is still redrawing the plot. This problem is illustrated in figure 4. The figure shows how many updates are made to the data list within one redraw for all performed redraws. It becomes obvious that the more samples have to be drawn, the more samples are added to the list within one redraw.

This issue manifests itself in an increasingly interrupted flow of the dynamic plot as the end of the plot is reached. With domain sizes of up to approximately 800 samples, the effect is hardly noticeable. Beyond this size, the dynamic plot gets increasingly jerky when up to 40 samples are appended to the plot at once (cf. figure 4).

Nevertheless, the end of the plot is reached accurately in the desired time because updating is never delayed in contrast to approach one where `postRedraw()` causes delay.
One aspect which can speed up redrawing is omitting the calculation of maximum and minimum values in each redraw. Figure 5 shows how much additional time is consumed by renormalization. When using method one, especially when isBlocking is set to “true”, a constant plotting speed cannot be guaranteed. Furthermore, the update rate of the plot is limited to roughly 50 samples per second (1000ms/20ms, cf. figure 2), which is insufficient for our purposes. Setting isBlocking to “false” still does not eliminate the growing redraw duration relative to the amount of samples. The duration is reduced drastically but a growing delay is still occurring (cf. figure 3).

Since the current version AndroidPlot 0.5.0 does not provide alternatives to the applied methods, the library would have to be manipulated or enhanced by adding classes specifically designed for dynamic plotting. These improvements for dynamic plotting should include to only redrawing absolutely necessary parts of the plot. In the current version, the maximum and minimum values are calculated and the grid is redrawn with every redraw. In our application, the maximum and minimum values of the ordinate are limited to a physiologically and anatomically reasonable range. Therefore they can be fixed anyway.

Fig. 5: A flowchart of the implementation, which redraws the plot within the UI thread via the Handler class.

Also, the re-rendering of all samples is not needed with every redraw. It would be sufficient to re-render only the most recently appended sample and to store the once rendered path. Since the AndroidPlot library is rather advanced, the effort for manipulating it to include the mentioned points could so far not be estimated.

5 Conclusion

Despite the limitations present with AndroidPlot, the described methods of plotting can be used as a basis to implement further improvements.

6 References