Efficient Implementation of Spreadsheet User Application

Tjaša Repič
tjasa.repic@student.um.si
Faculty of Electrical
Engineering and Computer Science,
University of Maribor
Koroška cesta 46
SI-2000 Maribor, Slovenia

Aljaž Jeromel
aljaz.jeromel@um.si
Faculty of Electrical
Engineering and Computer Science,
University of Maribor
Koroška cesta 46
SI-2000 Maribor, Slovenia

Sašo Piskar
saso.piskar@dewesoft.com
DEWESoft d.o.o.,
Gabrsko 11a
SI-1420 Trbovlje, Slovenia

Domen Dolanc
domen.dolanc@dewesoft.com
DEWESoft d.o.o.,
Gabrsko 11a
SI-1420 Trbovlje, Slovenia

Niko Lukač
niko.lukac@um.si
Faculty of Electrical
Engineering and Computer Science,
University of Maribor
Koroška cesta 46
SI-2000 Maribor, Slovenia

ABSTRACT

Processing measurement data is fundamental in the field of high-tech instrumentation, where precise measurement, collection, analysis, and visualization of data are of importance. When dealing with extensive amounts of data, it is necessary to display and process it in a way that ensures the cleanest user experience possible. We therefore often resort to tabular displays of data, since they are more comprehensible for the average user. In this paper we propose a solution, envisioned by the company Dewesoft - a computationally efficient spreadsheet editor widget tailored for their data acquisition software DewesoftX, additionally compatible with separate plugins within the software. Given that using commercially widespread tools to do so often results in setbacks when seeking to integrate those within existing software, our focus was on developing a user application, which will be functionally comparable to similar commercial solutions, while also complying with the existing software standards of the company.

KEYWORDS

spreadsheet, tabular data, optimisation, user experience, data visualisation.

1 INTRODUCTION

We widely adopt spreadsheets for their familiarity and versatility, offering extensive features for data manipulation, statistical calculations, data collection, and visualization. Their accessibility makes them a preferred choice for both individuals and businesses. However, spreadsheets also have notable drawbacks. They are susceptible to various input errors, including clerical mistakes, rule violations, data-entry errors, and formula errors, which can significantly distort the data. Additionally, spreadsheets are not inherently designed for efficient data storage or seamless connectivity to relational databases, posing challenges in effective data management and retention [1, 2].

In modern spreadsheet tools, providing a clear and efficient user experience involves several essential elements. These include an intuitive interface with a clean layout, consistent design, tool tips, and help guides. User-friendly navigation is achieved through a well-organized toolbar, robust search functionality, and keyboard shortcuts. Data visualization and formatting are enhanced by features like conditional formatting, and predefined styles. Comprehensive formula support includes auto-complete, error-checking tools, and a rich library of functions. Robust data management capabilities are also crucial, including import/export options, data validation, and integration with other tools [3, 4].

Within the initial design phase of the proposed Spreadsheet plugin solution, a crucial aspect of planning involved acquiring a deeper understanding of the DewesoftX software for which the plugin development was intended [5]. Processing measurement data is a crucial aspect of the advanced test and measurement industry, where the company Dewesoft operates [6]. A key component of Dewesoft's offering is the DewesoftX software, designed for use across various fields where precise measurement, data collection, analysis, and visualization are crucial. Given the large volume of data, there is a need to present and process it in a way that ensures a clear and straightforward user experience. DewesoftX is regularly used for data acquisition, signal processing, and data visualization across multiple industrial and commercial sectors. The software supports a wide range of interfaces for data visualization, allowing for the synchronized acquisition of data from nearly any analog sensor, storage, and visualization within the same file.

When discussing tools designed to display tabular data, it is essential to consider mathability. Mathability in spreadsheet tools refers to their capacity to perform complex mathematical operations with efficiency and accuracy. This capability is crucial as it ensures precise calculations, boosts productivity, and accommodates various applications across fields such as finance, engineering, and data science [7].

The paper presents our proposed solution's basic functions and the thought process behind their implementation, emphasizing the visualization aspect. It provides a detailed explanation of the implementation that ensures proper functionality. Additionally, we have included the results of duration and memory usage of various functionalities supported by our proposed solution.

2 METHODOLOGY

The purpose of the following subsections is to provide a breakdown, as well as a thorough explanation of the implementation process of the spreadsheet user application.

2.1 Fundamental Features

2.1.1 Spreadsheet Widget Layout. To enhance data accessibility during development and make the layout overview clearer, the widget workspace was divided into three parts. The visual widget is segmented into two primary regions. The first is the context menu, containing various button shortcuts for features that will be discussed further in this article. The software's user interface was originally developed in Delphi, using its own VCL (Visual Component Library) [8]. VCL is built on the Win32 architecture, sharing a similar structure but offering much simpler usage [9].

The second region, referred to as the spreadsheet rectangle or "TableRect", encompasses all data and its layout within the widget, including information about cells, columns, rows and the spreadsheet title. A smaller section called the data rectangle or "DataRect" additionally handles cell information. The context menu and the spreadsheet workspace can be seen on Figure 1.

We also provide users with the option to save data for future use. This is done by writing cell values, styles, merged cell information, resized columns and manually adjusted titles to a custom DewesoftX file format for saving the workspace which will further be referred to as .DXS or setup file. The data can then be read from the setup file after loading the workspace at a later time.

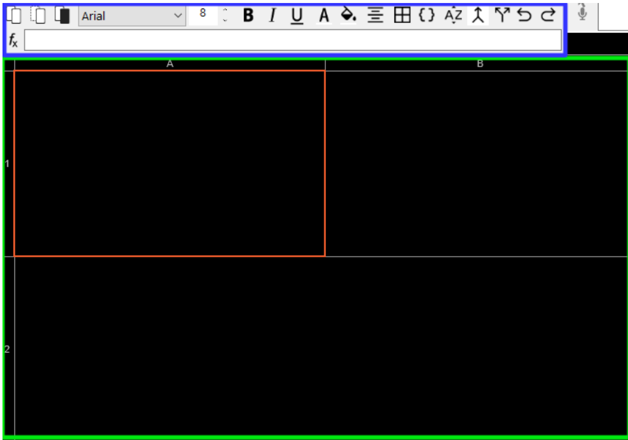

Figure 1: Separation between the context menu (blue) and area within which the spreadsheets's data is displayed (green).

2.1.2 Cell Manipulation. Initially we have made an effort of defining essential functionalities that define the main purpose and value of the application by dismantling related spreadsheet editors [10–12]. The most fundamental feature of the spreadsheet grid is the ability to insert and update data within the cells.

In order to grant each cell its own unique value we implemented a hash function, which generates a hash value from two integer inputs. These are determined by the cell's position within the grid, with the x value corresponding to the column and the y value corresponding to the row. The row index masks the lower 16 bits of the integer and shifts them 16 bits to the left. The column index is masked in the same way and remains unshifted. Using the bitwise OR operation, the two 16-bit values are combined into a single 32-bit integer. Implementation of a hash map designed to store the 32-bit value grants us a key for each cell, allowing us to insert and update data within the cell's index by combining characters received through user input into coherent values.

Where y is the row index, x is the column index, & represents the bitwise AND operation, << represents the bitwise left shift operation, | represents the bitwise OR operation, and $0xFFFF$ is a hexadecimal constant representing the lower bits, the hash key formula can be expressed as:

$$hash_key = ((y\&0xFFFF) \ll 16) \mid (x\&0xFFFF) \tag{1}$$

The hashmap provides $O(1)$ time complexity for retrieving data from the selected cell, which is highly desirable, because it means that the time required to perform an operation is constant and does not depend on the size of the data set.

The Spreadsheet widget also supports cell splitting and merging. Selected cells can be merged into a larger cell, which then behaves like a standard cell. To facilitate this functionality, cells include an additional parameter, "merged to", which records the cell to which they are merged. By default, for cells that are not merged, this parameter is set to -1.

A dedicated class manages cell selection within the spreadsheet, defining it by specifying the starting and ending column and row indexes. This enables users to efficiently apply operations to a range of cells, rather than being limited to individual cells.

2.2 Spreadsheet Formatting

Allowing users to stylize components in a user application is important for several reasons, including enabling customization to tailor the application's appearance to the user's individual preferences and allowing them to highlight important information and organize content according to their needs. For this purpose, we have incorporated various styling features into the spreadsheet user application.

2.2.1 Spreadsheet Stylizing. To increase customization possibilities, we added a structure within the Style class, containing eighteen properties. Thanks to this approach, the structure can be applied to all spreadsheet components, including individual cells, cell selections, rows, and columns.

Sixteen of these properties are dedicated to styling either the selected component or its contents. They include font family, font size, font colour, cell background colour, bold, italic, underlined text, six different cell border properties, and horizontal and vertical text alignment. It is also worth mentioning that only the values of properties set by the user are saved to the file, which speeds up reading/writing processes and reduces the size of the .DXS file.

The remaining two properties are the component's property mask, which assigns an integer value of either zero or one to each style feature in order to indicate whether that feature is enabled on a component, as well as a vector of integer values, indicating the style priorities for individual cells, rows and columns. This ensures

style properties are applied in the correct order were the cell's style to intersect with the style of its row or column.

2.2.2 Resizing Columns and Rows. We further enhanced spreadsheet customization by implementing the functionality for users to resize columns according to their specific needs. To resize a column, the user must trigger a mouse down event on the right edge of any column. This interaction detects the user's intention to change the column width and memorizes the index it has been detected on than determines the mouse movement according to the following equation:

$$moveX = x - O[r-1] - D_x, \tag{2}$$

where $moveX$ represents the horizontal distance the mouse has moved during the column resizing operation, x is the current horizontal position of the mouse cursor. The offsets array O holds the horizontal positions of the left edges of each column that is currently displayed in the spreadsheet, while r represents the selected index intended to be resized. Lastly, D_x represents the x-coordinate of the data rectangle's origin. This value is subtracted to ensure the calculation is relative to the data area.

We proceed with the operation by applying the following equation:

$$resizeRatio = \max\left(\frac{moveX}{cellWidth}, 0.1\right) \tag{3}$$

The equation calculates the ratio of the mouse movement to the cell width. It then ensures that this ratio is not less than 0.1 by using the max function. The calculated ratio is then set as the new column width on the resize index. Additionally, note that the cell width is the default width of the cells, which is calculated based on the font size settings.

2.3 Undoing and Redoing Spreadsheet Actions

Within the context of the spreadsheet application, we defined the undo and redo functionality as a state machine capable of switching between the current and previous states after applying a change to the spreadsheet and calling one of said operations.

To track state changes, we designed and implemented the "TableAction" structure. Different state changes require modifications to various types of data. The TableAction structure simplifies the process by encapsulating the type of action triggered along with parameters necessary for adjustment. We have provided detailed definitions of various Spreadsheet actions as shown in 1.

The spreadsheet actions are managed within the respective undo and redo vectors whose maximum size is set to twenty actions. Upon triggering an add action event, the initial state prior to the change is recorded in the undo vector, while the post-change state, along with its corresponding action type, is recorded in the redo vector. When the user initiates an undo or redo event, either via the context menu or keyboard shortcuts, the Algorithm 1 is executed.

To summarize, the algorithm begins by verifying the feasibility of the state change, ensuring that there is at least one recorded action in the action counter. If this condition is met, the action counter is adjusted appropriately. The algorithm then executes the necessary statements based on the type of action, ensuring that the corresponding data is modified accordingly. Finally, the visible state of the spreadsheet is updated to reflect these changes.

Data: Action History
Result: Undo or Redo Action

```
1  UndoOrRedo(isRedo) if no actions available then
2  |  return;
3  end
4  get action, adjust counter;
5  if action is Insert/Update then
6  |  set cell, edit value;
7  else if action is SetStyle then
8  |  apply styles;
9  else if action is Resize then
10 |  apply size changes;
11 else if action is Merge/Split then
12 |  update merge states;
13 else if action is TextPaste then
14 |  apply text;
15 else if action is Sort then
16 |  apply sorting;
17 else if action is Paste then
18 |  apply data and styles;
19 end
20 update visible cells;
```

Algorithm 1: Undo/Redo algorithm.

3 RESULTS

The measurements leading to the results presented in this paper were conducted on a system equipped with an AMD Ryzen 9 3900x 12-Core Processor, an NVIDIA GeForce RTX 2070 GPU, and 64GB of RAM. It is also important to note that DewesoftX version 2024.2 was used when conducting these measurements.

For the testing, we evaluated three spreadsheet functionalities across three progressively larger cell selections. The functionalities tested included pasting values into cells (see Table 1), undoing and redoing font colour changes (see Table 2), and loading from and saving to the setup file with the font colour state being set (see Table 3 and Table 4). The cell selection ranges used for these tests were 5x5, 25x25, and 50x50.

The data used in these experiments comprised text, numeric values, and dates, with font colour applied as specified. Each evaluation was conducted ten times under identical conditions, and the results were averaged to ensure accuracy.

Table 1: Evaluation of Cell Value Pasting.

Cell Selection Range:	Memory Usage [MB]:	Duration [ms]:
5x5	4.4	0.812
25x25	16.1	13.238
50x50	44.1	50.179

Within Table 1 the data shows that memory usage increases significantly with the size of the cell selection, from 4.4 MB for 5x5 to 44.1 MB for 50x50. The duration also increases with the size of the cell selection, from 0.812 ms for 5x5 to 50.179 ms for 50x50. A larger selection is expected to be more memory-intensive than its smaller counterpart. Interestingly, a selection 100 times larger is only 10

times more memory-intensive. This can be explained by the fact that the memory required for the basic widget to display correctly is also involved within the measurement and is independent of the amount of data stored in the spreadsheet. It's also worth noting that the duration does not increase linearly.

Table 2: Evaluation of Undoing/Redoing the Font Colour Property State.

Cell Selection Range:	Memory Usage [MB]:	Duration [ms]:
5x5	0.134	0.292
25x25	0.722	1.157
50x50	1.5	5.766

Within Table 2, memory usage increases modestly with larger cell selections, from 0.134 MB for 5x5 to 1.5 MB for 50x50. The duration also increases, from 0.292 ms for 5x5 to 5.766 ms for 50x50. The memory and time required to undo/redo font colour changes grow as the cell selection range expands. Comparing the measurements of the undo/redo function with the paste-into-cells function, we can conclude that the memory usage for the former is greater than that for the latter.

Table 3: Evaluation of Loading from Setup where Font Colour Property is set.

Cell Selection Range:	Memory Usage [MB]:	Duration [ms]:
5x5	97.8	1.769
25x25	117.6	33.173
50x50	123.6	132.758

For Table 3 memory usage increases with larger cell selections, from 97.8 MB for 5x5 to 123.6 MB for 50x50. The duration, however, shows an increase from 1.769 ms for 5x5 to 132.758 ms for 50x50, indicating that loading from setup becomes more time-consuming with larger selections.

Table 4: Evaluation of Saving to Setup where Font Colour Property is set.

Cell Selection Range:	Memory Usage [MB]:	Duration [ms]:
5x5	0.234	3.752
25x25	0.293	66.618
50x50	0.356	260.720

Lastly for Table 4, memory usage shows a slight increase with larger cell selections, from 0.234 MB for 5x5 to 0.356 MB for 50x50. The duration increases significantly with the size of the cell selection, from 3.752 ms for 5x5 to 260.720 ms for 50x50. This indicates that saving to setup is considerably more time-consuming as the cell selection size grows.

As expected, the results indicate that both memory usage and duration generally increase with larger cell selection ranges across all functionalities. Pasting values and saving to setup are particularly resource-intensive, whereas undoing and redoing font colour changes show a moderate increase in resource requirements. Loading from setup shows a notable increase in duration with larger selections, highlighting the complexity of handling larger datasets.

4 CONCLUSION

In this paper, we proposed a solution for handling measurement data in high-tech instrumentation through a computationally efficient spreadsheet editor widget. This widget is tailored for integration with Dewesoft's data acquisition software, DewesoftX, aiming to offer functionality comparable to commercial spreadsheet tools while ensuring compatibility with existing software standards. The solution focuses on enhancing data accessibility and user experience by providing an intuitive interface, robust navigation, and comprehensive formatting and state manipulation features.

In future development, we aim to enhance advanced mathability features essential for an efficient spreadsheet application. Specifically, we plan to implement a formula system within the widget, enabling users to input formulas and perform calculations directly within the spreadsheet. Additionally, we intend to incorporate conditional cell formatting, which automatically changes the appearance of cells based on their content to improve data visualization and analysis. We will also continue refining the existing features, taking user feedback into consideration to ensure the highest quality user experience possible.

ACKNOWLEDGMENTS

We are deeply thankful to Dewesoft and its representatives for their collaboration on this project, without which this paper would not have been possible.

REFERENCES

[1] F. Nurdiantoro, Y. Asnar, and T. E. Widagdo. The development of data collection tool on spreadsheet format. *Proceedings of 2017 International Conference on Data and Software Engineering, ICoDSE 2017*, 2018-January:1–6, Jul. 2017.
[2] Srideep Chatterjee, Nithin Reddy Gopidi, Ravi Chandra Kyasa, and Prakash Prashanth Ravi. Evaluation of open source tools and development platforms for data analysis in engine development. *SAE Technical Papers*, pages 1–11, Jan. 2015.
[3] Sabine Hipfl. Using layout information for spreadsheet visualization. *Proceedings of EuSpRIG 2004 Conference Risk Reduction in End User Computing: Best Practice for Spreadsheet Users in the New Europe*, pages 1–13, 2004.
[4] Bernard Liengme. *A Guide to Microsoft Excel 2013 for Scientists and Engineers*. Academic Press, London, United Kingdom, 2013.
[5] Dewesoft. Introduction | Dewesoft X Manual EN. https://manual.dewesoft.com/x/introduction, 2024. Accessed: 18-08-2024.
[6] Dewesoft. DewesoftX Award-Winning Data Acquisition and Digital Signal Processing Software. https://dewesoft.com/, 2024. Accessed: 21-08-2024.
[7] P. Biro and M. Csernoch. The mathability of spreadsheet tools. *6th IEEE Conference on Cognitive Infocommunications, CogInfoCom 2015 - Proceedings*, pages 105–110, Jan. 2016.
[8] Embarcadero Technologies. VCL Overview - RAD Studio. https://docwiki.embarcadero.com/RADStudio/Sydney/en/VCL_Overview, 2024. Accessed: 18-08-2024.
[9] Thomas Lauer. *Porting to Win32™: A Guide to Making Your Applications Ready for the 32-Bit Future of Windows™*. Springer-Verlag New York Inc., New York, NY, USA, 1996.
[10] Isaac Alejo. *Google Sheets Tutorial Guide*. Google Books, Online, 2024. Accessed: 18-08-2024.
[11] LibreOffice Documentation Team. *LibreOffice 4.1 Calc Guide*. Google Books, Online, 2024. Accessed: 08-08-2024.
[12] Karl Mernagh and Kevin Mc Daid. Google sheets vs microsoft excel: A comparison of the behaviour and performance of spreadsheet users. *Proceedings of the Psychology of Programming Interest Group (PPIG) 2014 Conference*, 2014.