

Designing and prototyping a visual,
incremental composing tool for change
ringing

Trinity Term 2021

Candidate Number: 1034710

Computer Science

Abstract

This project concerns the designing and prototyping of Jigsaw, an application to aid church bell-ringing composers. Jigsaw aims to be for bell-ringing composers what Microsoft Word is for writers.

Change ringing is where a team of people ring a set of 6–12 church bells in continually evolving and pre-determined sequences. The art of composing concerns designing these sequences for ringers to perform and dates from the 1700s. Composing was traditionally done by hand, and there are some computer tools to automate parts of the process. Jigsaw breaks new ground and, as far as the author is aware, is the first graphical, user-friendly and intuitive tool to aid the composer.

Jigsaw has an innovative graphical interface to make composing easier, more intuitive and more accessible for beginners. The tool was designed to a general goal and specific requirements, and was beta-tested by experienced composers to gain feedback and improve the design. Jigsaw is deployed as a static web app using GitHub pages.

The project also created BellFrame, a fast and idiomatic Rust library which provides easy-to-use data types for ubiquitous concepts in change ringing. BellFrame has good documentation with code examples. In the next stage of development, once its API is more stable, it will be published to Rust's package repository for general use.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 5 |
| 1.1 | Project Goals | 5 |
| 1.2 | Specific Requirements | 5 |
| 1.3 | Jigsaw | 7 |
| 1.3.1 | Overview | 7 |
| 1.3.2 | Static Web Application | 7 |
| 1.3.3 | JavaScript and Rust/WebAssembly | 8 |
| 1.4 | BellFrame | 10 |
| 2 | Overview of Change Ringing | 11 |
| 2.1 | Background | 11 |
| 2.2 | Methods | 12 |
| 2.3 | Compositions | 14 |
| 2.4 | Features of Compositions | 14 |
| 2.4.1 | Stage | 14 |
| 2.4.2 | Truth/Falseness | 15 |
| 2.4.3 | Length | 15 |
| 2.4.4 | Music | 17 |
| 2.4.5 | Complexity | 17 |
| 2.4.6 | Multi-Part Compositions | 18 |
| 2.4.7 | Summary | 18 |

| | | |
|----------|--|-----------|
| 3 | Design | 20 |
| 3.1 | Process | 20 |
| 3.2 | User Interface | 21 |
| 3.3 | Music Highlighting | 23 |
| 3.4 | Falseness Display | 24 |
| 3.5 | Lead Folding | 25 |
| 3.6 | Linking Fragments | 27 |
| 4 | Implementation | 28 |
| 4.1 | Overview | 28 |
| 4.2 | BellFrame | 28 |
| 4.2.1 | Rows vs Permutations | 29 |
| 4.3 | Model-View-Presenter | 30 |
| 4.4 | Specification vs Derived State | 30 |
| 4.5 | Undo History | 32 |
| 4.5.1 | Passing Data Between Rust and JavaScript | 33 |
| 5 | Assessment and Analysis | 36 |
| 5.1 | Jigsaw | 36 |
| 5.1.1 | Overview | 36 |
| 5.1.2 | Ease of Use & Visual | 36 |
| 5.1.3 | Completeness | 37 |
| 5.1.4 | Incremental | 37 |
| 5.1.5 | Instant | 38 |

| | | |
|----------|---|-----------|
| 5.2 | Final Statistics | 40 |
| 6 | Conclusion | 41 |
| 7 | Appendix | 43 |
| A | Glossary of Change Ringing Terms | 43 |

1 Introduction

1.1 Project Goals

This project concerns prototyping an application to *aid* composers whilst requiring the smallest possible change to their existing workflow. This is analogous to what Microsoft Word provides for writers — a kind of ‘automated paper’ which automatically annotates your work with data that is tedious to compute manually whilst still providing it in a format that you are familiar with (e.g. words on a page). Additionally, it must provide correct feedback regardless of the completeness of the composition.

1.2 Specific Requirements

Whilst the goal above gives a general design direction, more specific requirements are needed to build a cohesive application. Therefore, I split this general goal into five specific goals which were used to design and then assess the prototype application. These goals were derived from observing the limitations of existing tools, conversations with well-known composers, and experience gained from experimentation early in the project.

Ease of Use The application should be graphical and present its data in a way that is intuitive and accessible to the user. It should also be made so that anyone can install and use it easily, without any technological knowledge or a lengthy installation process.

Completeness There is a concrete definition of what is considered ‘Change Ringing’, and the states representable in the application must be a superset of those allowed by this definition. This definition is provided by the Framework for Method Ringing¹.

Incremental The application should allow the user to incrementally build up their composition, in whatever order they want, starting wherever they want to. Specifically, it must be able to represent ‘partial’ compositions — i.e. a set of composition ‘fragments’ which will eventually be combined into a full composition.

Instant Feedback on important measures like truth, music content and length should update instantly whenever the composer changes their composition.

Visual Everything that can be understood visually should be displayed visually. Where specific numerical data is required (e.g. length), these should also be provided in a suitable format.

¹https://cccbr.github.io/method_ringing_framework/classification.html

1.3 Jigsaw

1.3.1 Overview

The application I designed and prototyped for this project is called Jigsaw. The GUI has two main parts: an infinitely scrollable canvas containing the composition fragments, and a folding sidebar which overlays the canvas and shows information that cannot be represented in the canvas. Jigsaw is intended to be used on a desktop with a keyboard and mouse.

Jigsaw supports a large number of operations to modify partial compositions. All of these are bound to keys on the left side of the keyboard (because the user's right hand is expected to be on the mouse), and for consistency they always apply their effects to the location of the cursor.

1.3.2 Static Web Application

Jigsaw is built to be run as a single static web page (i.e. it is a fixed set of files that can be served by a simple HTTP server). Releasing applications as static websites has many advantages:

1. There is no start-up time for a new user — they open the page in any browser and the application starts immediately.
2. HTML canvas rendering (while extremely inefficient compared to OpenGL or Vulkan) is easy, fast enough to feel instant and looks consistently good across all browsers and operating systems.

Any performance concerns turned out to be a non-issue; after implementing simple culling of off-screen rows, there have been no issues with rendering performance when running on modern hardware. Additionally, there are many opportunities to optimise the rendering if required, meaning that this will still be an easily fixable issue in the future.

3. Having no dependence on a server both obviates the need for the user to make an account (which worsens the user experience) and also means that it can be deployed using GitHub pages or a similar service.

Persistent storage is the main challenge of static web applications, since websites cannot directly access the file system for security reasons. Cookies are one possible option but they are sent over the network with every request, making them unsuitable for storing large amounts of data. Browsers do provide a persistent cache for network requests which can be arbitrarily read from and written to, thus providing persistent storage.

1.3.3 JavaScript and Rust/WebAssembly

For Jigsaw’s user interface, the native web languages (JavaScript, HTML and CSS) are the obvious choice. The ‘back-end’ of Jigsaw, however, has to do a lot of complex operations on data every time the user changes the composition being displayed. This must be done quickly enough that the user cannot perceive the delay, and must be reliable.

JavaScript is very unsuitable for this kind of workload. Firstly, it has no static analysis, making it unnecessarily difficult to write complex code correctly. It also gives no control over memory layout and is absolutely reliant on a JIT (Just In Time) compiler for performance. Jigsaw's compute pipeline consists of several thousand lines of code, which are unlikely to all be compiled by the JIT compiler.

Therefore, another language is required. Jigsaw's back-end is written in Rust, a systems programming language which emphasises performance and safety (the two things Jigsaw needs). The Rust code is compiled to WebAssembly so it can run natively in the browser. Other languages could work for this purpose (for example TypeScript, C/C++ or Go) but I chose Rust for the following reasons:

1. Rust provides direct control over memory layout and has no garbage collector, providing consistently excellent performance when manipulating complex data structures. The performance of TypeScript, for instance, is by definition no better than that of JavaScript.
2. The Rust compiler performs a large amount of strict compile-time correctness checks on your code (including guaranteeing memory safety), making it easy to implement complex algorithms in ways that you know is correct and fast. C/C++ and Go cannot compete in this regard.
3. Rust has first class support for WebAssembly. Jigsaw is greatly aided by `wasm_bindgen`, which automatically generates binding code between

native Rust and native JavaScript to let the two languages interface seamlessly.

4. I am already proficient in Rust, so no time will be spent learning new languages for the project.

1.4 BellFrame

When implementing Jigsaw, it became clear that a large amount of the Rust code was providing data structures that were specific to change ringing but not just to Jigsaw. Therefore, it made sense to keep this code separate as a stand-alone library (named BellFrame) that could be re-used in other ringing-related projects. I made sure that BellFrame has good documentation and test coverage, and I will publish it to Rust's package repository once its API becomes more stable. Despite not being an explicit goal of this project, BellFrame is as much a product as Jigsaw itself.

2 Overview of Change Ringing

2.1 Background

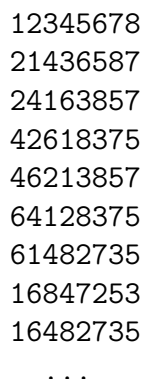
Change ringing is a niche and not well-understood activity so this section will serve as a short introduction to the topic and overview of the aspects of change ringing which are relevant to this project. A more complete overview can be found on Wikipedia², and I have also provided a glossary of commonly used terms in Section A.

Change ringing (sometimes known as full-circle church bell ringing) originated in English churches in around 1600, where churches began hanging sets of tuned bells on wheels in towers. The installations range from 3 to 16 bells, with even numbers from 6 to 12 being almost ubiquitous. These bells are typically quite heavy (bells over a ton are fairly common) and are spun back and forth through 360 degrees like a pendulum.

All the bells are rung once in a sequence which changes each time. A piece of change ringing is a sequence of these sequences, known as ‘changes’ (hence ‘change ringing’). Because ringers have little control over the frequency of their bell, bells only move by at most one place between two adjacent rows. On paper, we write these permutations in order as rows of numbers and by convention, the bells are numbered sequentially with ‘1’ referring to the highest-pitched bell in the tower (see Figure 1).

In order to break up the walls of numbers, we will often draw coloured

²https://en.wikipedia.org/wiki/Change_ringing



```

12345678
21436587
24163857
42618375
46213857
64128375
61482735
16847253
16482735
...

```

Figure 1: Writing change ringing compositions as lists of rows

lines through the path of some bells. By far the most common colouring is to give the highest pitched bell (the ‘1’) a thin red line, and the lowest pitched bell a thick blue line (this is used extensively in the diagrams throughout this report — the leftmost column of Figure 2 is showing the same rows as Figure 1).

2.2 Methods

A performance of ringing will often contain thousands of individual rows. Physical aids to memory are not permitted during performances so the ringers must ring thousands of unique rows completely from memory. Memorising them all directly is clearly not feasible, so compositions are built up from repeating patterns known as ‘methods’.

Every method has a name (often with historical or geographical connections). In the method each bell moves through a pre-determined path, moving back and forth within the sequence. Ringers learn these paths to

help them remember where to ring in the sequence of rows.

A single instance of a method is called a ‘lead’ and can be thought of as a super-permutation; it takes a start row and produces a sequence of rows, ending with the row which the next lead should be started. This final row is part of the next (non-existent lead) is considered ‘leftover’. Leftover rows are greyed out to emphasise this. When multiple leads are chained together, lines are drawn to differentiate individual leads and increase readability.

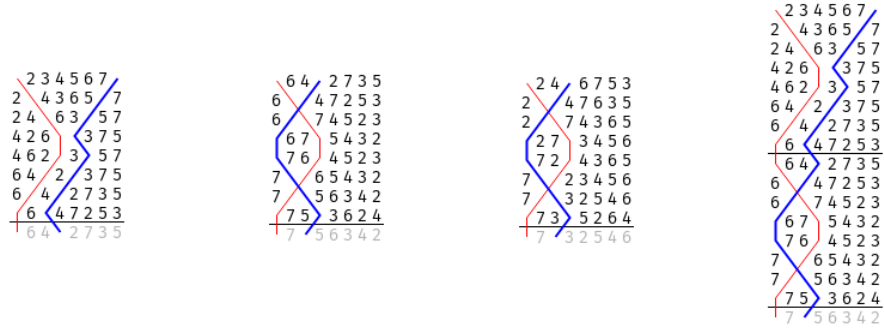


Figure 2: A collection of leads of the same method. Note the use of lines to represent the paths of bells 1 and 8.

For example, the left three columns shown in Figure 2 are all leads of the same method (this particular method is known as ‘Little Bob Major’). The second lead starts with the ‘leftover’ row of the first, and therefore they can be concatenated to form the fourth column.

Note that a single lead of this method preserves the location of the bell at the start of the row. This is an extremely common feature of methods because it allows ringers to orient themselves around this ‘fixed’ bell and therefore helps prevent mistakes.

2.3 Compositions

A ‘composition’ is a sequence of rows of some length which satisfies the adjacency property and starts and finishes with the row containing bells in descending order (123456 . . . , known as ‘rounds’).

The task of a composer is to create compositions which have some set of desirable features, which depends heavily on the circumstances of each performance. In order to understand the rationale behind this project and gauge its success, we must first understand some of the most important features (and thus constraints) of compositions.

2.4 Features of Compositions

2.4.1 Stage

The “stage” of a composition refers to the number of bells which it uses. These are given specific names (for example “Triples” refers to 7 bells and “Major” refers to 8 bells), but these are often confusing and are outside the scope of this report.

Since towers almost always contain an even number of bells, the stage of a composition is not necessarily the number of bells that the composition is rung on. For example, compositions of “Triples” (i.e. 7 bells) are almost ubiquitously rung on 8 bells with the lowest bell always remaining at the end of each row (this is known as a “cover bell”).

2.4.2 Truth/Falseness

For a given stage n , there are $n!$ possible rows that a composition of that stage can use. If we were to count the number of occurrences of each row in a composition, then the composition is “true” if these counts are as balanced as possible (and “false” otherwise). Specifically, if a true composition on stage s contains n rows, every possible row must be repeated either $\lfloor \frac{s}{n!} \rfloor$ or $\lceil \frac{s}{n!} \rceil$ times.

For example, imagine a composition of 1260 rows on 5 bells. There are $5! = 120$ possible rows so for this composition to be true, every row must be repeated at least $\lfloor \frac{1260}{5!} \rfloor = 10$ times and 60 of these must be repeated an 11th time.

Because $n!$ quickly gets huge as n grows, a very common case is where $s < n!$. In this case $\lfloor \frac{s}{n!} \rfloor = 0$ and $\lceil \frac{s}{n!} \rceil = 1$ so for these compositions, “truth” is equivalent to the rows being unique.

It is worth noting here that compositions are written with rounds at both the start and end. However, when discerning truth these are only counted once — otherwise, nearly every composition would be false.

2.4.3 Length

The ‘length’ of a composition is the number of rows which it contains. Like falseness, external rounds is only considered once.

96.5% of performances and 90.5% of compositions³ have one of two length

³There have been 136,206 performances of peals and 312,004 of quarter peals (out of

ranges:

1. A **Peal** consists of at least 5000 rows (about 3 hours' ringing). The number 5000 is chosen because historically the pinnacle of ringing challenge was to ring every permutation of 7 bells, which results in $7! = 5040$ rows. This was recently rounded down to 5000 for all numbers of bells, but the majority of people prefer the completeness of 5040 rows for peals of 5, 6 or 7 bells (where every possible row is rung the same number of times).
2. A **Quarter Peal** consists of at least 1250 rows (a quarter of a peal's length, and about 45 minutes' ringing). Due to the reduced completion time, these are much more common than peals. Half peals (≥ 2500 rows) and double-length peals ($\geq 10,000$ rows) are both recognised lengths but performances of them are very rare.

Half peals (≥ 2500 rows) and double-length peals ($\geq 10,000$ rows) are also recognised lengths but performances of them are very rare.

Also, every row above the lower bound is extra ringing time for little gain in performance merit, and therefore composers try to make compositions which are as close as possible in length to these lower bounds. Indeed, lengths within the ranges $1250 \leq l < 1300$ or $5000 \leq l < 5100$ are almost ubiquitous.

464,564 total). There are 31,827 published peal compositions and 8,597 quarter peal compositions (out of 44,676 total). Statistics were taken from BellBoard (bb.ringingworld.co.uk) and Composition Library (complib.org) on 23rd of May 2021.

2.4.4 Music

If we are composing for ≥ 8 bells, then our composition will not contain every possible row and we therefore have the opportunity to decide which rows get included and which do not. Of these, some may be more favourable or “musical” than others.

Ringers like rows which start or finish with at least 4 consecutive bells, which give a pleasing musical effect. The exact bells are usually unimportant and the longer the run the better. For example 34567812 starts with a descending run of 6 bells, and is therefore preferable to 57864321 which ends with an ascending run of 4 bells. Both of these are preferable to 18346527 which has no runs.

There are (many) other forms of music, but they are outside the scope of this report.

2.4.5 Complexity

Complexity does not have a clear mathematical definition and therefore this project will make no attempt to automate it. However, it is still an important concept which roughly corresponds to how repetitive and predictable the composition is.

In general, we want to make these things as simple as possible because that will open our composition to be performed by more people. However, sometimes *the point* of a composition is to be a challenge for the ringers, in which case complexity is a target not a limitation.

```

12345678
<part>
13425678
<part>
14235678
<part>
12345678

```

Figure 3: An example of a 3-part composition.

2.4.6 Multi-Part Compositions

Because repetitive compositions are easier to learn, it is very common to create compositions which have multiple identical repeated ‘parts’. A ‘part’ is a fragment of a composition which, when repeated end-to-end, produces the full composition. This is clearly an advantage since, if a composition has n parts, a potential conductor only needs to learn $\frac{1}{n}$ th of the composition and repeat it multiple times.

Each ‘part’ will not end in rounds and therefore is not a composition itself. A common way to create multi-parts is to rotate bells 2, 3 and 4 to give 3 parts. In writing this would look like Figure 3.

2.4.7 Summary

In general, stage, length and falseness are hard constraints which composers have little choice over and must work around. Most of the time, the composer fixes some target level of complexity and tries to get as much music or interest as possible within that constraint. Also, many composers make compositions that they intend to conduct themselves, allowing for a higher

level of complexity because they already have an intimate knowledge of the composition before learning it.

3 Design

3.1 Process

When starting to design Jigsaw I had a rough sketch of what the application would look like (for example, I knew I wanted to embed composition fragments into an infinite canvas). I then refined my rough ideas into the specific requirements enumerated in Section 1.2.

At each point in the development process, I would pick the goal that has the least dependence on goals or code which I had not completed. Care was taken to move on to new goals instead of fixating on one part of the design, since often improving other parts of the design provides additional context that affects already existing changes. This creates a more cohesive overall design, as well as preventing the project from stalling over small design problems.

For example, the first goal I worked on after writing some necessary utility code was the visual aspect of rendering fragments — because regardless of future decisions, Jigsaw has to have be able to render fragments nicely in order to satisfy the goals.

On the most part, this ordering was quite obvious, but there were a few cases where the I approached problems in what turned out to be the wrong order. This did not detriment the resulting design — it just meant that some features took longer than necessary to implement. In particular, lead folding was implemented late in development and required radical changes to the

Jigsaw’s internal pipeline. A large amount of time would have been saved by implementing it earlier in the design process.

From a design perspective, this approach worked well — Jigsaw’s design is quite consistent and intuitive and I am pleased with the result.

3.2 User Interface

The user interface of Jigsaw consists of an infinitely scrollable canvas which can contain any number of composition fragments in any locations. Overlaid to the right of this canvas is a sidebar containing mostly numerical information which is ill suited to the canvas-style interface. The sidebar contains a lot of dense information, so all sections can be folded to only show what’s relevant. See Figures 5 and 4 for a screenshots.

Jigsaw is also designed to be used on a desktop with a 3-button mouse and keyboard, and optimises for this use case. All operations are bound to single keys with helpful mnemonics (e.g. `a` and `A` to *add* to the composition), and these quickly become muscle memory and allow an experienced user to quickly manipulate compositions. For consistency, all operations are applied at the location of the mouse cursor. Editing speed is very important for software designed for creative users, since work throughput is basically inversely proportional to iteration time — the more iterations are possible, the quicker and better the result will be.

Since most people use a mouse with their right hand, all operations are bound to keys on the left side of the keyboard even if doing so requires worse

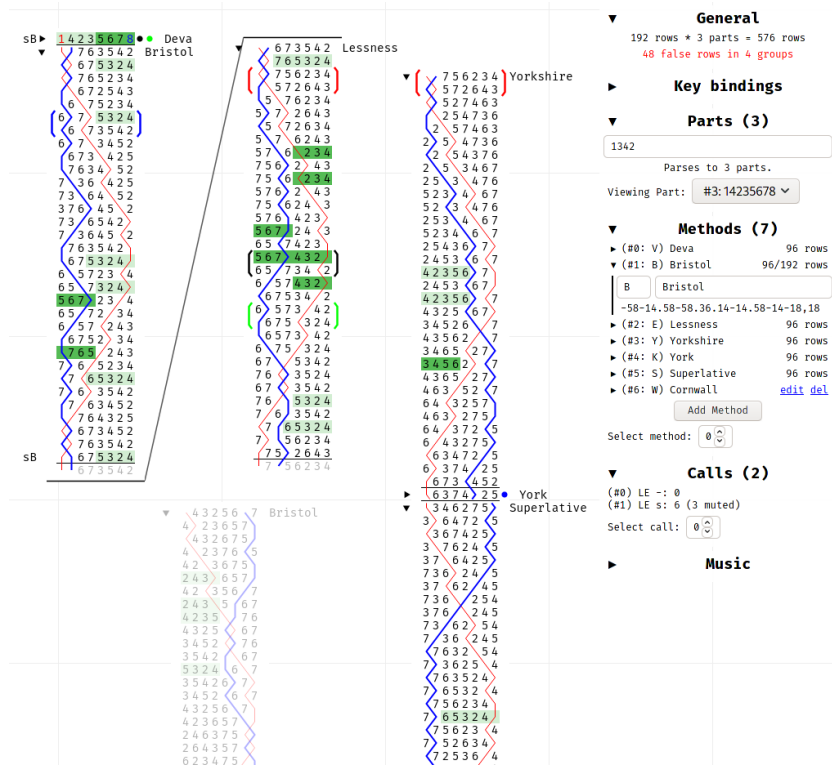


Figure 4: Screenshot of Jigsaw, showing the folding UI and canvas display

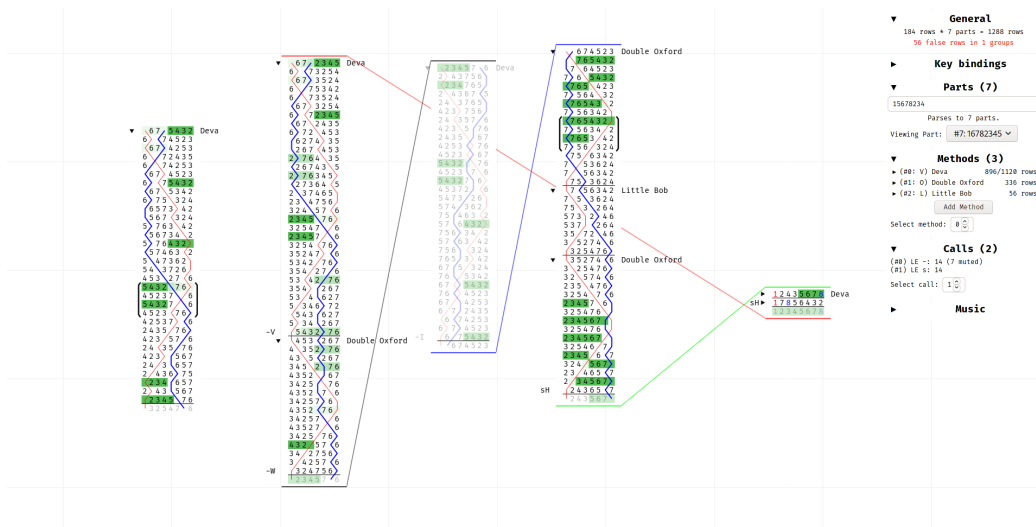


Figure 5: Editing a Quarter Peal composition

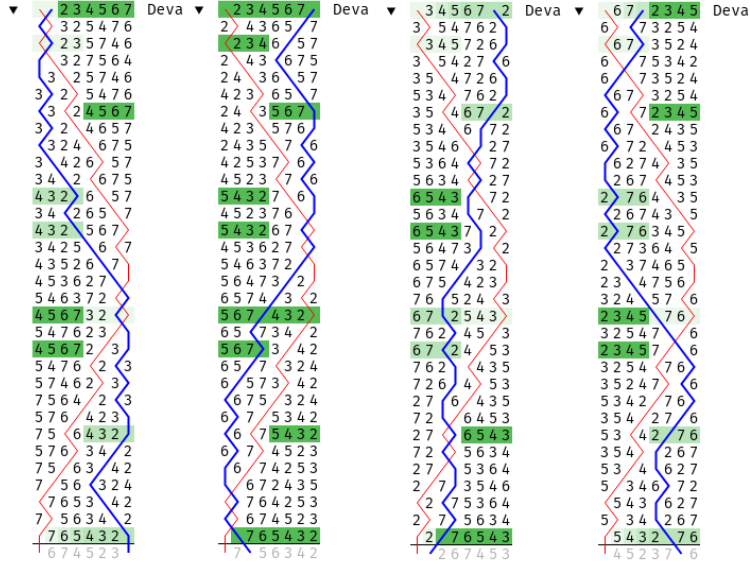


Figure 6: The same lead in four different parts (rotating 2–8)

mnemonics. This means that every operation is always instantly accessible to the user, maximising efficiency of editing and therefore improving the user experience.

3.3 Music Highlighting

When music appears in a composition, the bells causing the music are highlighted green, giving an obvious visual indication of where the music is.

Additionally, non-musical rows can generate music in other parts of the composition, and in this case the music is ‘onion-skinned’ — the more parts contain music, the darker the colour. In Figure 6, for example, 4328 in the 1st lead is highlighted quite densely because it generates 5432, 6543, etc. in

than just relying on colour.

3.5 Lead Folding

To prevent the user from having to look at thousands of rows at the same time, Jigsaw allows leads to be folded into a single on-screen row. Jigsaw then converts falseness groups into dots next to the row (see Figure 8), and switches between lines and numbers for displaying bells when necessary.

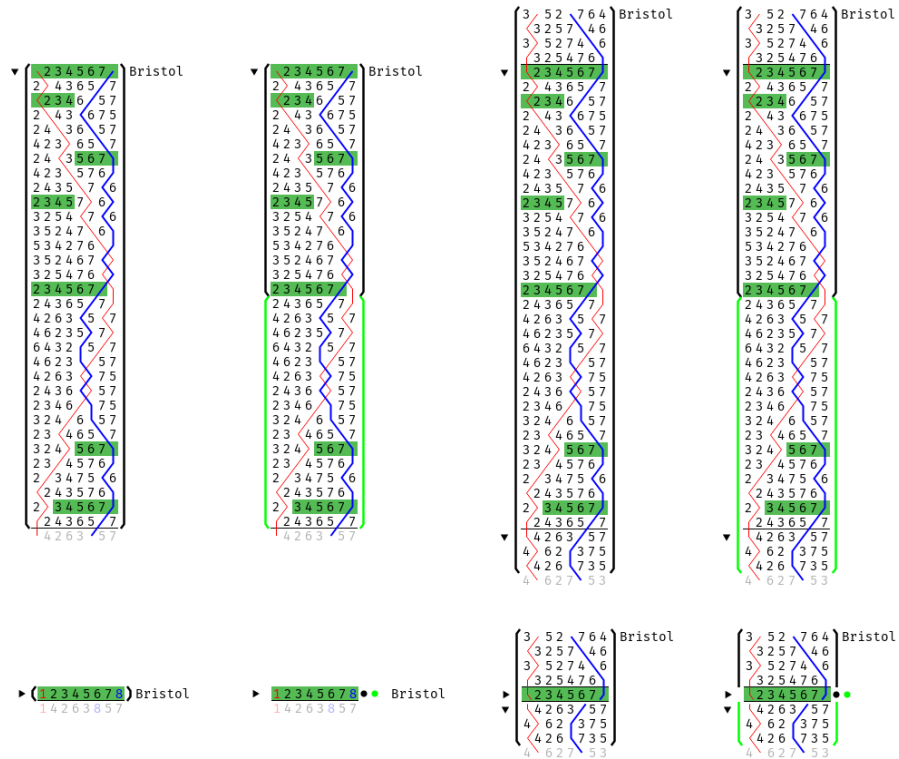


Figure 8: Unfolded (top) and folded (bottom) versions of the same lead. Note the handling of falseness ranges and line segments.

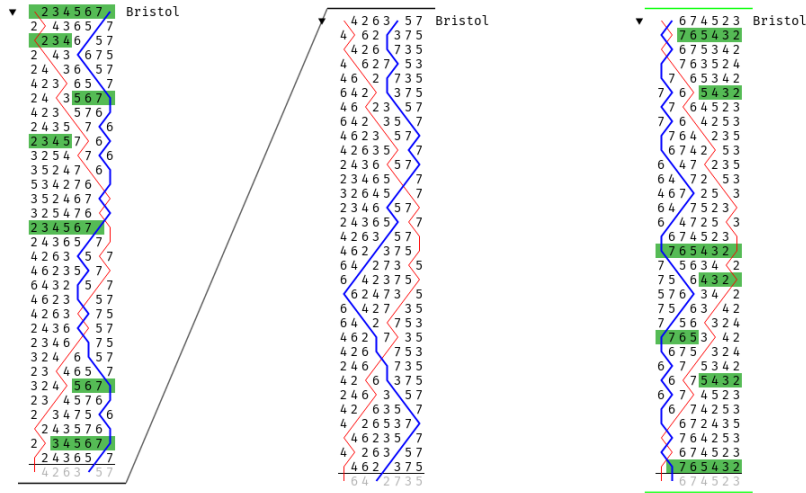


Figure 9: Two fragments which link together, and one that links to itself.

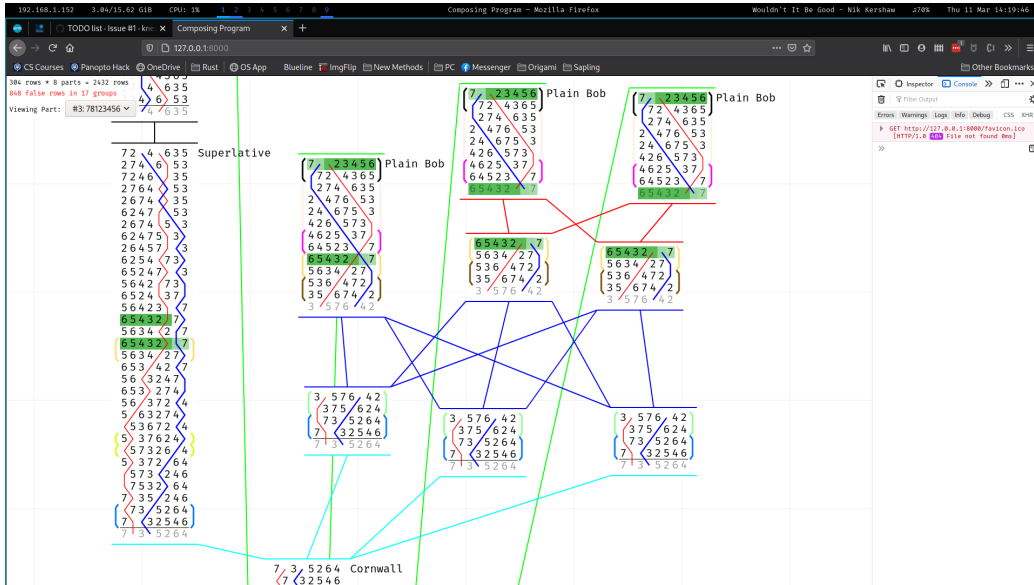


Figure 10: A lot of fragment links on one screen. The GUI has changed since this screenshot was taken but the linking display has not.

3.6 Linking Fragments

Two fragments can be linked together if the leftover row of one is equal to the first row of the other. To represent this visually, Jigsaw draws a coloured line between the two (see Figure 9). If multiple different sets of fragments link, then each group is coloured separately (see Figure 10).

Hovering a line and pressing `c` (for *connect*) will join those fragments into one. Note how this operation does not change the rows contained in the composition.

4 Implementation

4.1 Overview

The code for Jigsaw is split into two main parts: a pair of Rust libraries which together form a ‘server’ and a web GUI which forms a ‘client’. The Rust code is compiled to WebAssembly, and exports an API which allows the JavaScript client to request changes to the internal ‘model’ and after each update return a JSON serialisation of the information to display to the user.

This architecture plays to the strengths of both languages — Rust provides fast and reliable data manipulation whilst JavaScript, HTML and CSS provide a user-friendly cross-platform GUI. The binding and serialisation code is automatically generated at compile time using *wasm-bindgen*⁴ and *serde*⁵ to prevent human error.

4.2 BellFrame

Roughly half of the Rust code in the project is a cross-platform utility library for core data types which are not specific to Jigsaw. This provides safe and performant Rust data types for common constructs related to change ringing (`Row`, `Bell`, `Method`, `Call`, etc.). Its development and API design was motivated by Jigsaw but once its API is stable it will be published to Rust’s central package repository for general use.

⁴<https://github.com/rustwasm/wasm-bindgen>

⁵<https://github.com/serde-rs/serde>

The stable part of the API has good documentation and is well tested, and all data types only provide public functions which preserve internal invariants, obviating the need for unnecessary run-time checks. It also provides ‘unsafe’ versions of functions which remove runtime checks but rely on their caller to uphold the invariants. Therefore, no undefined behaviour is possible if the user only writes ‘safe’ code.

Despite having no stable API, I have already built other projects on it. In its own right, BellFrame is a valuable product of this project.

4.2.1 Rows vs Permutations

When writing BellFrame, I initially thought that the concept of rows and permutations were different enough to warrant separate data types. My thinking was that a **Row** is a sequence of **Bells** (plus some additional invariants), and a permutation can be thought of as a function of type `[a] -> [a]` which can permute sequences of any type. Mathematically this feels like an elegant separation, but using the library in earnest made me realise that the **Perm** type does not pull its weight given its large overlap with **Row**. I would almost always use **Perm** only as an intermediate step to use one **Row** to permute another. Therefore, a few months into the project the operations provided by **Perm** were merged into **Row**.

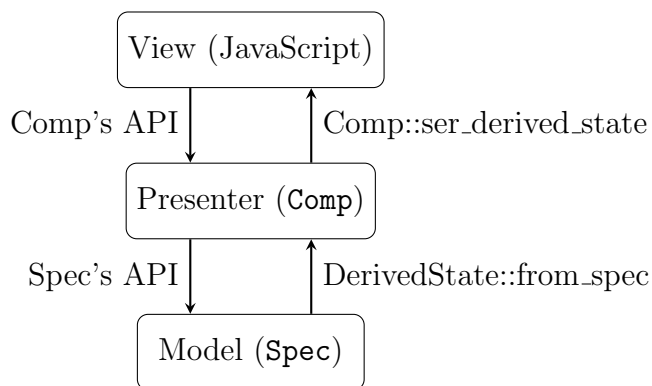


Figure 11: Jigsaw’s implementation of model-view-presenter

4.3 Model-View-Presenter

Jigsaw’s architecture follows the Model-View-Presenter architecture, a relatively common variant of Model-View-Controller. This was not a conscious design decision; it felt like the most natural way to structure the code.

Figure 11 shows how Jigsaw’s code maps to Model-View-Presenter. Additionally, the Model-View-Presenter means that every edit passes completely through the model, resulting in a data flow shown in Figure 12.

4.4 Specification vs Derived State

The code in the ‘server’ library differentiates between the unique specification of a composition (the **Spec** type) and the information required to display it to the user (the **DerivedState** type). The **DerivedState** contains additional data like the music locations, falseness and other annotations. **DerivedState** is completely specified by the contents of the current **Spec**, so every change

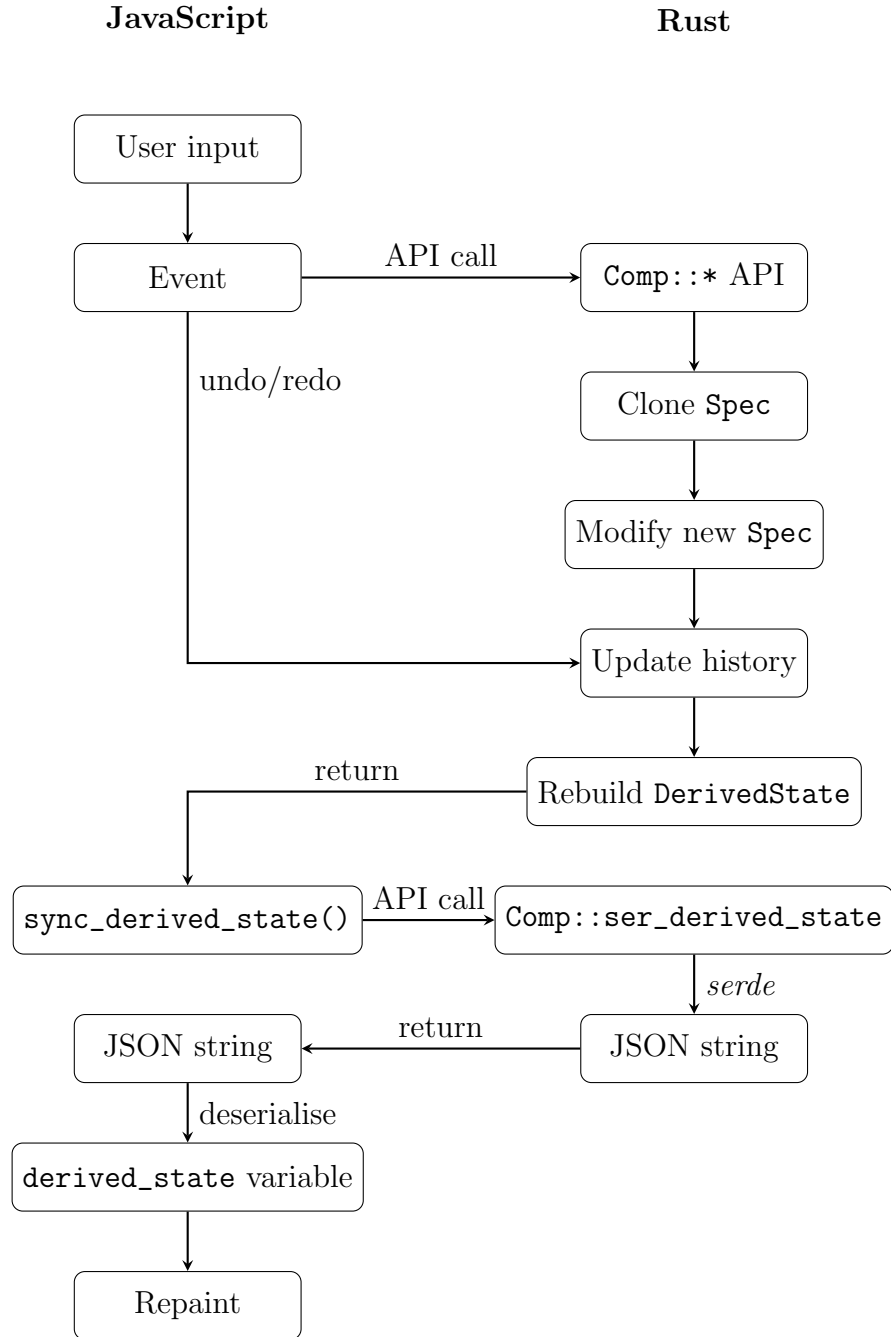


Figure 12: The data flow of user's changes

to the `Spec` must also cause the `DerivedState` to be updated before the changes can be displayed to the user.

In terms of Model-View-Presenter, the `Spec` type represents the Model and `DerivedState` represents the data sent from the Presenter to the View.

4.5 Undo History

Jigsaw uses the ‘immutable data structure’ method of storing undo history, where the history is a sequence of fully defined states (as opposed to an event system where the history is a sequence of modifications). Therefore, the undo history is simply a list of `Spec` types along with an index referring to the ‘current’ `Spec` being displayed on the screen.

In order to prevent unnecessary duplication of data, the internal data of the `Spec` type is always stored behind reference-counted pointers. Therefore, cloning a `Spec` adds one to the reference count rather than cloning the backing data. In Rust, reference-counted pointers are always immutable so in order to modify some part of a `Spec`, we must first make the backing data unique using `Rc::make_mut` provided by the standard library. Rust checks this at compile-time and will reject code which tries to modify shared data without performing this step.

This approach works excellently, and furthermore allows Jigsaw to fully update the display *without* adding steps to the undo history. This allows Jigsaw gives the user instant and complete feedback when performing longer operations like changing the start row of a fragment.

4.5.1 Passing Data Between Rust and JavaScript

Even with *wasm-bindgen* providing binding code, all data types that are sent over the language boundary must be flat structures with no references. This is unfortunate, since `DerivedState` has no fewer than four layers of references (`DerivedState` \rightarrow `DerivedFrag` \rightarrow `DisplayRow` \rightarrow `Row` \rightarrow `Bell`), all of which is required to render each frame. Therefore, it is not possible to simply return a `DerivedState` from Rust into JavaScript, and another strategy is required.

Initially, I did not want JavaScript and Rust to hold their own copies of `DerivedState`, because then there would be two different sources of the same truth (potentially diverging and causing bugs which would be extremely hard to fix). Additionally, I wanted to limit the complexity handled by the JavaScript code (due to its inferior performance and maintainability) so it felt natural for Rust to handle all the state and then expose a simple API to JavaScript to read the data.

The original API was a flat set of getter functions, looking like Figure 13. This worked fine for a while but as the complexity of `DerivedState` grew, the number of exported functions exploded and refactoring the API became unnecessarily challenging. Also, every WebAssembly function call takes a non-trivial amount of time and this flat API encourages huge numbers of calls in each frame which started causing performance issues. It is also worth noting that the `DerivedState` only updates when the user changes the composition, but is needed every time the UI is painted.

```

#[wasm_bindgen]
impl Comp {
    ...
    /// Get the number of fragments
    fn get_num_fragments() -> usize { ... }

    /// Get the number of rows in a given fragment
    fn get_frag_length(frag_index: usize) -> usize { ... }

    /// Get the bell at a given location
    fn get_row(
        frag_index: usize,
        row_index: usize,
        bell_index: usize
    ) -> Bell { ... }
    ...
}

```

Figure 13: Initial getter API design

Eventually, I settled on the strategy that Jigsaw currently uses: to pass the entire `DerivedState` structure as a JSON string after each edit. JavaScript then parses and caches the resulting object which it reads from when drawing each frame. This turned out to work really well — almost no WebAssembly calls are made during each frame, the ‘getter’ API is reduced to a single function and long-term divergence is still impossible because JavaScript completely overwrites its copy of the `DerivedState` each time the composition updates.

The serialisation code is generated at compile-time using *serde*⁶, removing the possibility for human error and guaranteeing that the data in JavaScript and Rust will always have the same structure, making the whole code base much easier to read and refactor.

⁶<https://github.com/serde-rs/serde>

5 Assessment and Analysis

5.1 Jigsaw

5.1.1 Overview

I think that Jigsaw is a successful prototype and has potential to be built into a feature-complete application, which I now plan to do. I demonstrated it to several other composers during development for feedback on the design. This was overall very positive; they all thought it looked very promising, but identified a number of minor issues that would allow it to be more useful in finished form.

In order to decide on the success of the project, I will rank it against the design goals set out in Section 1.1.

5.1.2 Ease of Use & Visual

Firstly, building Jigsaw as a single static web page means it requires no installation. The design of an infinite canvas plus a folding sidebar feels very natural and intuitive to use.

A large amount of effort went into making sure that canvas GUI is visually crisp and responsive. This goes as far as rounding all coordinates to the nearest pixel to prevent fuzzy boundaries and erroneous gaps, as well as a more involved rounding system for line rendering. Attention to detail like this makes the experience of using Jigsaw feel very smooth and polished.

All actions are currently bound to easily accessible keys which means that editing compositions quickly becomes muscle memory for experienced users, but feels unintuitive to beginners because there is no indicator of what actions are available at a given point. In a future development of Jigsaw I will consider adding a contextual right-click menu for this — such a menu can even show the key bindings to help beginners learn them.

5.1.3 Completeness

Jigsaw’s internal representation is complete, but the current user interface is not. In other words, Jigsaw is internally able to represent a strict superset of compositions which are considered change ringing, but the prototype GUI means that the user is unable to express some very obscure states which are technically considered change ringing. For perhaps 99% of use cases, Jigsaw’s user interface is complete enough that most users would not notice the difference.

5.1.4 Incremental

Jigsaw is completely built around providing incremental editing. It also does its best to preserve ‘continuity’ of operations — that is, a small operation should cause a small change to the composition’s rows. For example, some operations (like splitting and joining fragments) change only the structure of the composition, not the output rows.

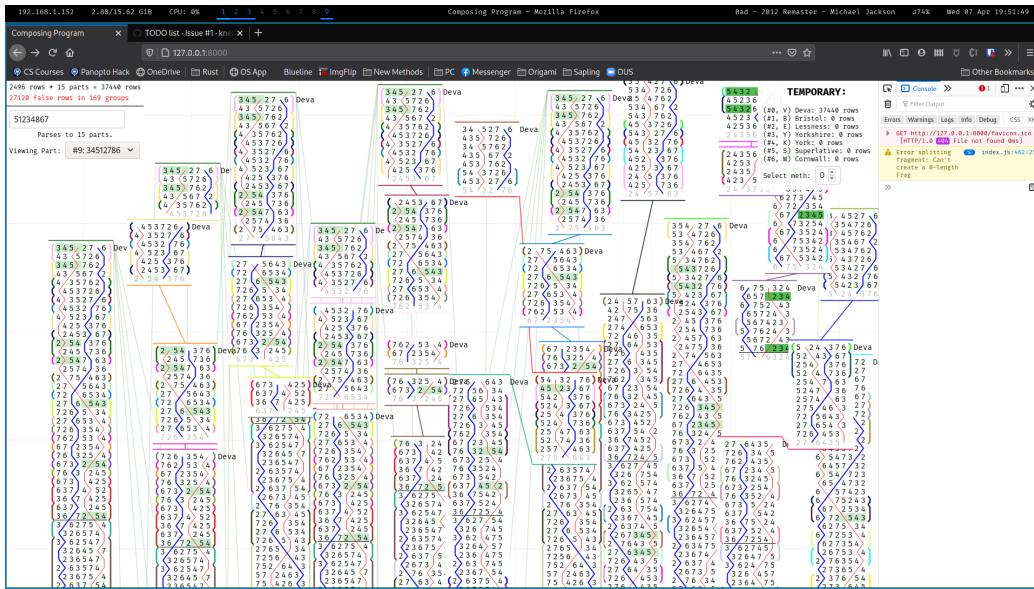


Figure 14: Stress testing Jigsaw (the GUI has since changed, but the pipeline has not). The fragments continue off the bottom of the screen, so roughly a third of the rows are on screen.

5.1.5 Instant

Due to the internal architecture, the composition is always fully annotated. Whenever the user causes an update, the entire state of the display is recalculated then cached for repainting. Therefore, it is simply not possible for any annotations to lag behind because the Jigsaw’s full state is simultaneously recalculated after every change.

This does, however, mean that Jigsaw is extremely dependent on the efficiency of its update pipeline. Luckily, Rust and WebAssembly are working extremely well in this regard, since the pipeline is fast enough to reliably feel instant even for large compositions. I have not done any specific optimisa-

tions, but I have been mindful of high-level performance when writing the code — the algorithms are all $O(n \log n)$ in the number of bells in the composition, although the link detection is $O(n^2)$ in the number of fragments. This means that Jigsaw is far from its potential performance ceiling, allowing for straightforward optimisations in case the UI stops feeling instant in the future.

Whilst stress testing Jigsaw, I discovered that the pipeline’s latency becomes noticeable at around $35,000 \times 8 = 280,000$ bells (see Figure 14). Since 99.7%⁷ of compositions have at most 10,000 rows each of 16 bells (making 160,000 bells total) I think that the performance is adequate.

The longest piece of ringing ever rung contains $72,000 \times 6 = 432,000$ bells, which would challenge the performance of the current implementation. In reality, the composition for this was split into many separate pieces to ease memorability so I doubt anyone would want to work on a composition of this size in full. However, I would like Jigsaw to eventually support huge compositions and there is plenty of opportunity to optimise — for example, the pipeline does no caching or re-use of heap allocations so the code generates huge numbers of needless allocations during each pipeline run.

Additionally, `BellFrame` contains the `SimdRow` type which uses SIMD instructions (specifically `pshufb`) to perform row permutations in a single clock cycle, which usually increases performance by at least an order of magni-

⁷44,551 out of 44,683 compositions on Composition Library have this property, as of 23rd of May 2021.

tude over the list-based `Row`. Furthermore, the size of `SimdRow` is known at compile-time meaning that their creation does not require a heap allocation. In return for this huge performance improvement, `SimdRow` has a fixed limit of 16 bells which requires any code to dynamically fall back on the heap-allocated `Row` when required, significantly increasing the complexity and potentially creating obscure bugs. Finally, SIMD is not supported in mainstream WebAssembly yet so Jigsaw will have to wait before using `SimdRow` is a viable option.

5.2 Final Statistics

As of writing this report, Jigsaw's GitHub repository has 275 commits and contains 5,736 lines of source code (4,233 of Rust, 1,033 of JavaScript, 240 of HTML, 114 of CSS and 85 of Python for the build script). Including comments and blanks increases this total to 7,556 lines. For completeness, the LaTeX file for this report is 968 lines long.

6 Conclusion

This project set out to create a prototype for an application to aid composers of church bell ringing. A general design goal was set, and this was broken down into five specific requirements. The resulting tool, named Jigsaw, has met all of the five requirements and its usefulness has been validated by beta-testing and feedback from experienced composers. Jigsaw’s combination of intuitive graphical interface and efficient implementation has proven highly effective.

The reusable utility library on which Jigsaw is built (named BellFrame) is also a significant contribution to composition which others will be able to benefit from.

In the next stage of development I would propose to take the prototype and address the minor issues identified in beta-testing and further improve usability, for example with a right-click menu. I would use a more complete testing process — automated testing of the code to improve reliability and user testing to refine the user experience. This will ensure that Jigsaw is consistently robust in a wide range of use cases.

Creation of this prototype has been a valuable learning experience. If I were to do a similar project again I would be more careful about which order features should be designed and implemented. For example, retrofitting lead folding was unnecessarily difficult and this extra complexity would be avoided had I implemented it earlier. When building a production application, I

would place a much heavier emphasis on stability and automated testing than I did for Jigsaw. However, for prototypes like Jigsaw, the rapid iteration means that tests are often invalidated shortly after their creation.

Finally, I would like to acknowledge the guidance of my project supervisor, Dr Geraint Jones and the composers — past and present — who inspired me to create this tool.

7 Appendix

A Glossary of Change Ringing Terms

Stage: The number of bells which the composition uses. This is not necessarily the number of bells used to ring a given composition, but for the purposes of composing that distinction is mostly irrelevant.

Row: A sequence of bells which forms a permutation. Rows are the fundamental building blocks of compositions. The words ‘change’ and ‘row’ are often used interchangeably (hence the name ‘Change Ringing’), but to avoid confusion I will always use ‘row’ in this report.

Composition: An ordered sequence of rows which tell ringers in which order the bells should be rung.

Truth: A composition is ‘*true*’ if the rows included are as balanced as possible, otherwise it is ‘*false*’. This is the single most important property of a composition, since any performance of a ‘false’ composition is considered invalid. See Section 2.4.2 for a longer explanation.

Method: A short, usually symmetrical, pattern that is repeated throughout a composition with small and well-defined modifications (known as ‘calls’). This is the key to how human ringers can ring over 5000 rows worth of composition without any memory aid — in reality, all ringers in the band mem-

orise the method(s) and one ‘conductor’ memorises the sequence of calls and shouts them in the right places for the other ringers to obey. It is usually the job of the composer to make sure that the call sequence is predictable and easy to memorise whilst preserving other valued properties of compositions (see Section 2.4).

Lead: A single instance of the repeating pattern of a method.

Rounds: A special row with all the bells in ascending order of number (written 123456...). All compositions must start and finish in rounds.