Visualising Information Change Over Time

Samuel Grant Dawson Williams

October 8, 2010

Abstract

Entropy dictates that over time, energy will transform from a higher state to a lower state. However, Nature also creates complexity by building and constructing new and wonderful things. In many ways, humans are the same, we create and adapt information for our own purposes; from moment to moment, things are changing. This research explores the various ways the change in information can be visualised over time, with a specific focus on programming related projects.

Contents

1 Introduction

Information often changes over time, and these changes can be recorded in many different ways. Visualising the change of information over time is ultimately about looking at trends in data, and trying to present them in a meaningful way. These trends – if presented effectively – can provide insight into the size, scale, structure and history of the information.

1.1 Information Repositories

This visualisation research focuses on general sets of information. It builds on top of source code management tools in order to extract the way information has changed over time.

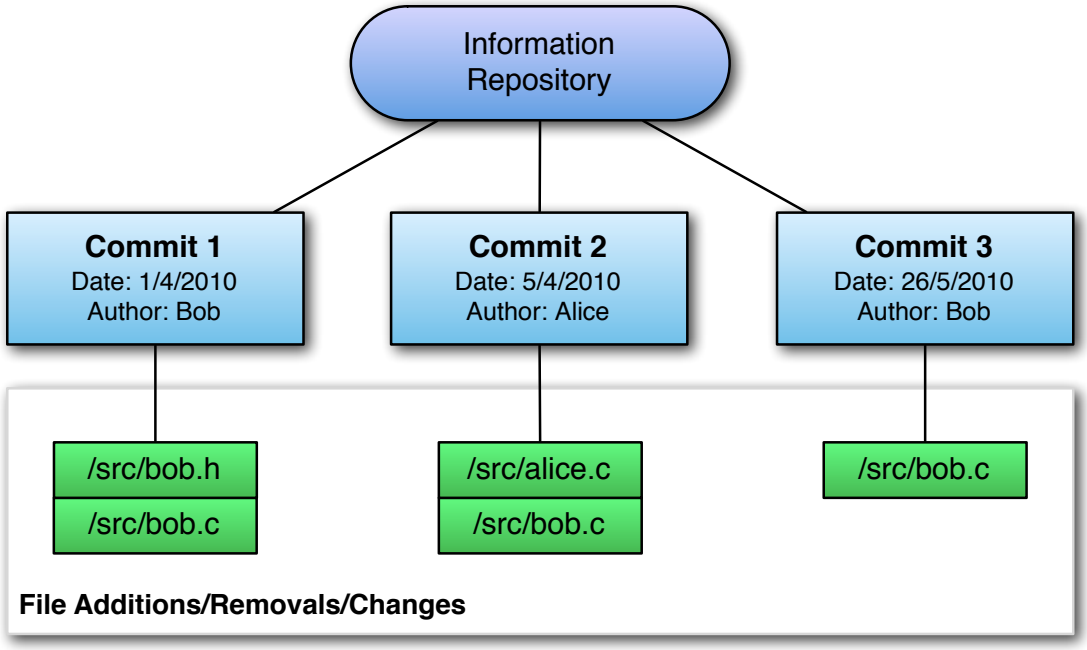

Figure 1: Information changes in discrete commits.

Information in a set of files can be considered a repository (see figure 1). A related set of changes to one or more files can be referred to as a commit, and this represents a single atomic change in the repository. A commit may also include relevant meta-data such as author, the date of the commit, and other important details.

The primary unit of change is lines added or removed. In addition, at each step throughout history the entire information set can be reconstructed, and then more complex semantic models can be constructed and compared through time.

1.2 Goals

The primary goal of this project is a general information visualisation tool that conveys to the viewer the size and scope of a project, including where[1] information has changed and the scale of the change. The main use for this information is for both historical analysis[1] and for high level project management tasks; to bring visibility to the otherwise typically invisible.

More specifically, many of the case studies consist of large bodies of programming language code and related data. It will be important for any visualisation tool to provide insight into program structure and help programmers understand a program more quickly, especially if they are not familiar with it. Source code visualisations are useful for several reasons:

- Comprehending the size of a project and the various components of a project. This might be useful for new programmers, or people who are interested in the size and scope of a project.
- Identify patterns correlating release cycles to work done. These might be useful for finding out if people are working effectively or if there is a huge rush before the next release.
- Keep track of when and where work is being done in a project. This could be useful for a manager to direct resources towards specific parts of a project.
- Historical purposes such as looking at parts of the project that have merged or diverged over time.

[1] Such as placement in time, structural position, etc.

2 Background

Existing software visualisation and analysis tools were reviewed to provide a wide background on which to build my own ideas.

Many popular visualisation tools appear to have two important traits: they are visually engaging, and they present interesting information. Even if users can't understand the information, if the visualisation is exciting they will spend time exploring it – they may even learn something about the data after being engaged with the visualisation for a period of time.

2.1 Existing Visualisation Tools

Gource[2] is a software history visualisation tool. It allows the end user to review the changes to source code over time at a very high level, including the person who makes the change. While the idea is fascinating and the implementation visually exciting, the actual technical information that is conveyed is minimal. In many ways, it might be more useful when used in real time; for example, it would allow the user to find out who is working on a particular part of a large project. It could also be used as a navigational tool for understanding the structure of a project.

Extravis[3] (Execution Trace Visualizer) is a visual tool for inspecting the execution of a program. In this sense, it allows the end user to accurately visualise the connections between components which would otherwise be separated by layers of indirection.

Doxygen[4] is a fantastic tool for generating documentation from source code. It can create high level diagrams and outputs documentation in multiple formats. It is a fantastic tool for understanding source code and quickly getting up to speed with a large project.

SLOCcount[5] (Source Lines Of Code Count) is an excellent tool for understanding the high level composition of a program in terms of what languages are used. It could easily be extended to show this information over time by integrating it with a version control system.

3 Project Work

Most of my own projects are stored in git[6] repositories. I also use a wide variety of programming languages. For a visualisation to be useful to me, it must be sufficiently general – this may be true for many users who want to produce their own visualisations. The focus of my project work was thus on the visualisation of information change over time, where change in this case is represented as the addition or removal of lines in text files.

3.1 Visualisation #1: Files and Commits

My first visualisation tool (see figure 2) processes a repository of information and produces VRML. It has the following primary features:

- A list of all files in a repository are placed along the x axis.
- Commits are placed along the y axis. For each file that changed in a particular commit, the total lines added and removed are used to place a red and green box. These are stacked up vertically.
- For each file, a blue accumulator bar is defined throughout time. When a file is changed, the change is multiplied by the current accumulator value. If there is no change, the accumulator is divided in half. The goal is to improve visual clustering of volatile areas.
- Two viewports are defined, from the front, and from the back, looking down over the data.
- The user can interact with the graph using the basic navigation features of their VRML viewer.

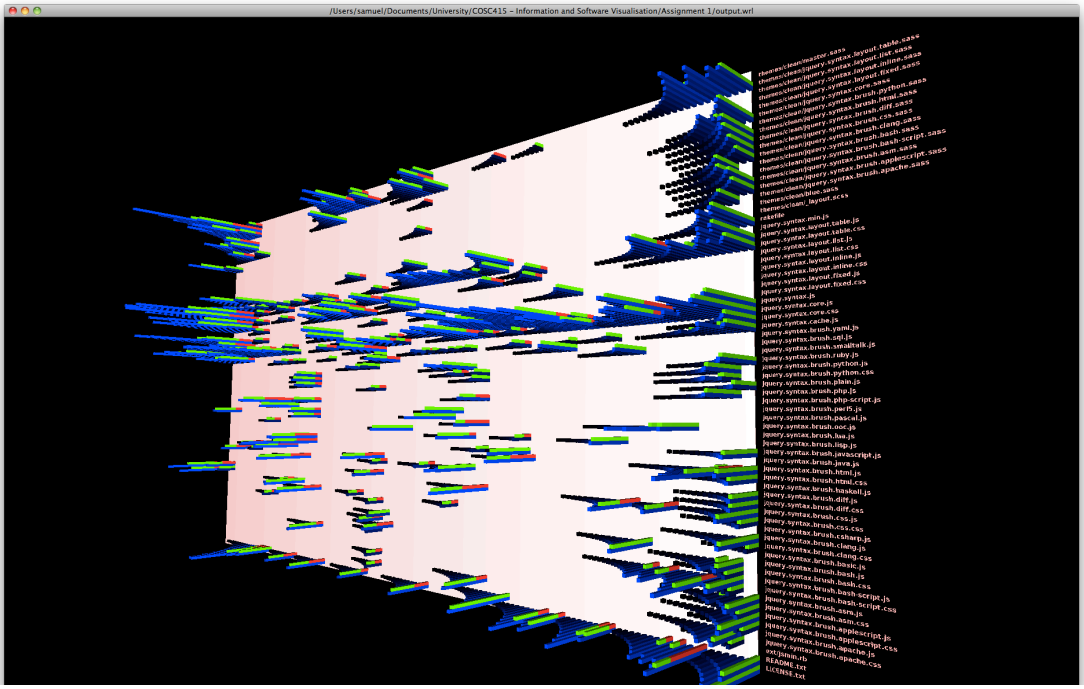

Figure 2: My first attempt at source code visualisation over time.

3.1.1 Good Features

This visualisation represented a test case for loading data from a version control system. It demonstrated that it was possible and that there were many possibilities for analysis and visualisation.

The blue bar used a non-linear calculation for its colour $1 - \frac{1}{0.1x^2+1}$. This function allows data to be mapped between 0.0 and 1.0 regardless of the input value. The result quickly expands and then tends towards 1.0. This allows us to avoid preprocessing the data in order to recognise the maximum value.

3.1.2 Usability Issues

The size of the visualisation was linearly related to the size of the data being visualised. If there are many files, or many commits, the size of the visualisation may become large, and the usability will suffer (see figure 3). This exposes itself in several ways: the performance of the client application decreases, the ability of the user to correlate data in a meaningful way is decreased and the visual quality decreases as the number of elements on the screen exceeds the resolution of the display.

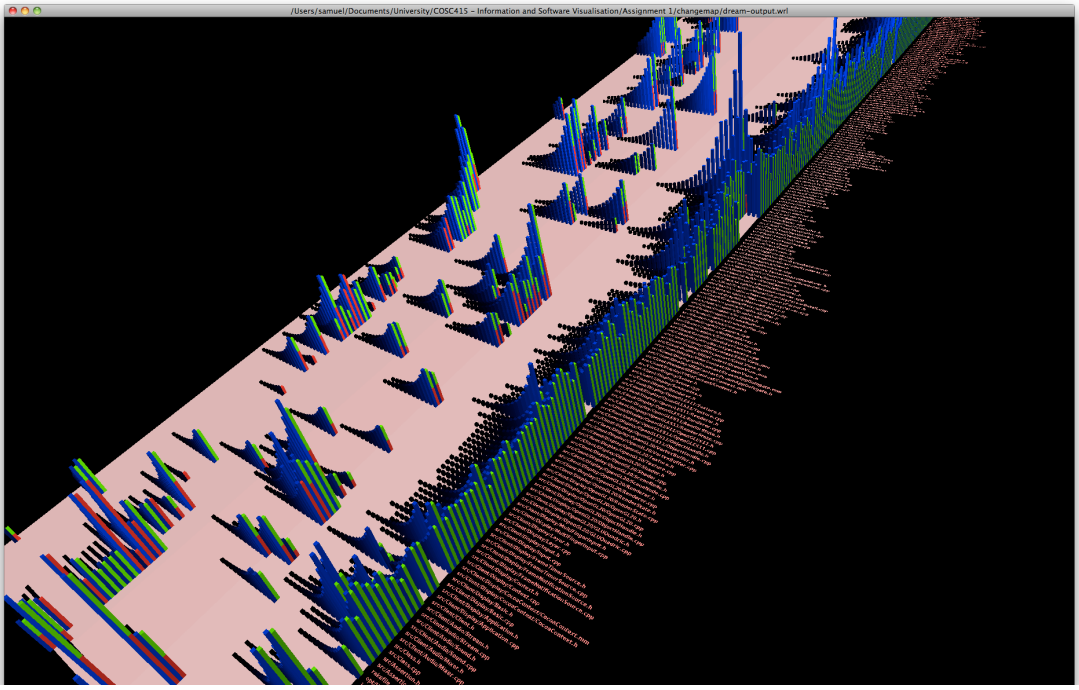

Figure 3: The visualisation is much wider than it is deep, and it is difficult to get a broad overview without data being obscured.

Navigation and interpretation of 3D data is difficult because the changes in one file may obscure changes in another file, depending on perspective. It is also difficult for the user to correlate the column with the associated file, especially as the graph includes more and more data. From a top down perspective, we can see the changes over time, but not the size of the change. From a side/isometric perspective, we can see some changes, their size, but some information is then obscured.

Commits are not linearly related to time in this graph; they are simply placed one after the other. This means that it is impossible to correlate the relationship between time and change in the graph in a consistent way (i.e. relative distances along the time axis have no meaningful interpretation). This behaviour could be improved by changing the visualisation implementation.

3.1.3 Summary

Many ideas were explored and those that were successful provided a sound basis for further work. This particular implementation encountered both technical limitations of the platform and intrinsic limitations of the 3D format. These included accessibility, interactivity, text rendering and layout issues. The visualisation was abandoned at the point where further work did not yield the desired improvements.

3.2 Information Extraction

When looking at information over time, one interesting question is 'Does this change fit with my previous expectations?'. Because we have access to all changes throughout time, we can build a set of related probabilities that show if one file is related to another. This graph is directional, and allows us to predict if we change one file, the percentage likelihood that another file will change.

This metric is interesting for a number of reasons. Firstly, it allows us to correlate changes to sets of files over time, rather than individual files. We can group files together and use this to assist with visualisation. Secondly, it allows a high level analysis of changes that might be problematic. If a new programmer commits file A, but not B, and there is a high correlation between $A \rightarrow B$, this might be flagged as a potential issue. Finally, it allows us to build a global graph of relationships between files and this could conceivably be a visualisation in its own right.

I implemented this feature by using a proximity function $P(A, B) = \frac{|\Delta A| \rightarrow |\Delta B|}{|\Delta A|}$. This reads, the proximity of A to B relates to the number of times B has changed when A has changed.

The results this function provided were interesting, but also very noisy. Implementing a low cut filter helped to reduce noise, but further work in this direction was not pursued.

3.3 Visualisation #2: Heat Maps

After thinking about my initial 3D visualisation and its problems, I realised that the information could be used to create a heat map. We use a similar set of axes as in visualisation #1, but rather than using 3D height to show impact, we use colour instead. A heat map is a well known format and thus should be fairly intuitive. Building on the experiences of the previous visualisation, several decisions were made:

Visualisation #1	Visualisation #2	Motivation
VRML	HTML	Availability, usability and flexibility.
Non-linear time	Linear time periods	Scalability and reliable data correlation.
File list	Filtered n-level directories	Scalability and context/focus.
Lines added/removed	Impact calculation	Better modelling of input data.

My second visualisation tool (see figure 4) processes a repository of information and produces HTML. It has the following primary features:

- A comprehensive summary of data at the top of the graph.
- A list of filtered sub-directories are placed along the vertical axis.
- A periodic linear time scale along horizontal axis at the top of the graph.
- At each data point, an appropriately coloured box to represent the aggregate amount of change.

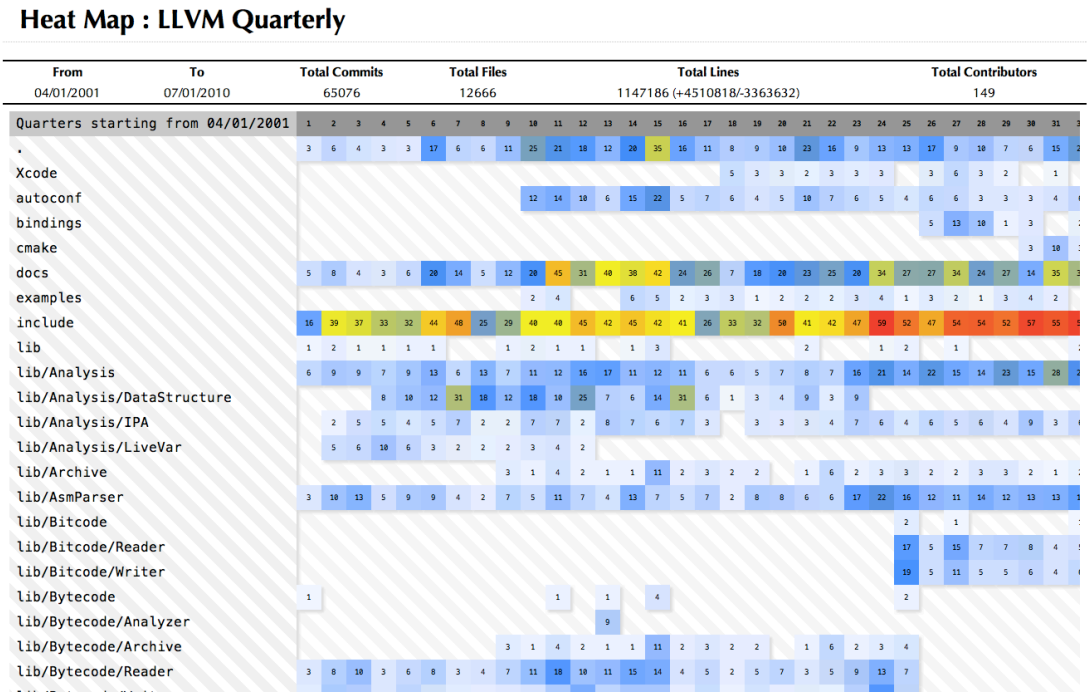

Figure 4: A quarterly visualisation of LLVM (a project with over 9 years of history and 65k+ commits).

3.3.1 Improvements

HTML provides a much more mature platform for visualisation. Features such as row highlighting and tooltips became feasible, and data navigation was reduced to 2D scrolling, which improved usability a great deal. Performance was improved, and because the platform is far more accessible, the visualisations could be easily published and viewed online easily with any modern computer[2].

To improve navigation for larger data sets, drag-to-scroll was implemented. This allows the user to drag the graph horizontally without using a scroll bar. However, despite these improvements, large data sets will still be difficult to visualise completely without some concept of zooming in and out[3].

Because the visualisation included a lot of text, the 2D nature of the web helped improve the readability and visibility of this text. The left hand panel which lists sub-directories, is always visible despite scrolling the graph left and right through time. Other important details such as a summary of the data are presented at the top of the graph, and are easily accessible.

[2]This is in comparison with VRML, which is not widely and consistently available.
[3]Some browsers support this operation by increasing and decreasing page size.

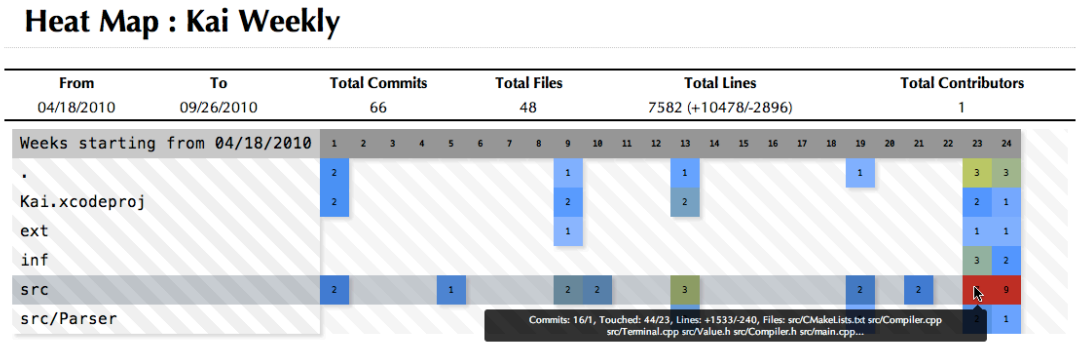

Figure 5: A weekly visualisation of Kai (a project with only 1 year of history). Notice the horizontal row highlight, and the tooltip underneath the currently selected box.

The heat map visualisation presents a linear view of time, divided into sequences such a days, weeks, quarters or years (part of the initial configuration). This is especially useful because different projects have a wide variety of time scales, as can be seen with the comparison of LLVM over 9 years (see figure 4) and Kai over 1 year (see figure 5).

Rather than listing every file as in the first visualisation, directories are aggregated beyond a specified depth. This avoids problems with huge lists of files and reduces the visual clutter. Filters can be specified which allow data to be specifically indexed to a defined depth, and this allows detail to be exposed when it is useful to do so.

Revisions are subdivided into an appropriate sub-directory and time period, and an appropriate colour is displayed (from cold blue to hot red) to indicate the size of the aggregate changes. Colour is calculated using a logarithmic scale which allows data to be compressed appropriately. Careful tuning was done to ensure that the distribution of colours maps appropriately to the work done, even between small or large projects.

Tooltips are displayed when the user hovers the mouse over a particular revisions box (see figure 5). These tooltips contain detailed information such as the number of commits, the number of files changed, and an ordered summary of the hottest files (i.e. the most heavily edited).

When the user hovers over a box, the path name and row is dimmed. This is to help the user process related content.

The background of the visualisation is a carefully selected diagonal pattern which helps to provide a point of reference when navigating the graph. The left hand panel and revisions boxes have a slight drop shadow to improve composition and improve the user's spatial awareness.

3.3.2 Usability Issues

As a user scrolls down the graph, the date bar stays at the top. This makes it difficult to co-relate time periods on larger graphs. To fix this, the date header should be fixed to the window rather than the page.

Another problem with the date header is the use of numbers to refer to incremental periods. If the user hovers the mouse over the date box, it shows the date for the start of the period, but it doesn't give a global

view of the progression of time. A solution to this problem would be to add an additional header grouping periods into logical segments. As an example, for periods of weeks, this could be months and for periods of quarters, this could be years.

Colour presents usability issues for people who are perceptually unaware of it, or don't have the same intrinsic appreciation for the metric it represents. The colour gradient was based off the most common set of colours for heat maps and carefully chosen, however despite this, it still presents a major problem for human accessibility.

There is currently no way disclose further details about the data. It is impossible to drill down into a sub-directory as the visualisation is generated statically with everything showing by default. To augment the visualisation with dynamic features would require a restructuring of program so that all the statistics are available at run time.

The tooltips pop up in a different direction depending on where the mouse cursor is on the screen. If the cursor is on the left hand side, the tool tip extends to the right, and vice versa. At the middle of the screen, this 'jump' can disturb eye tracking of the tooltip content. It should be possible to fix this by linearly adjusting the position rather than having a 2-case switch.

3.3.3 Summary

Existing ideas from the previous visualisation were refined and several new ideas were explored. The accessibility of HTML has meant that several thousand people have been able to look at the visualisation.

There are many opportunities for further improvements; however, one of the key points about this visualisation is simplicity. It has been deliberately designed to be unobtrusive and clutter free.

One possibility towards this goal would be to remove the numbers in the boxes, or reduce their visibility (they could be disclosed when the mouse hovers over a specific box for instance). The overall purpose of this would be to increase the users reliance on colour which is the main medium of communication in this visualisation.

Another place where the visualisation could be improved is the addition of a vertical row marker. As the mouse moves over a revisions box, the horizontal row is dimmed to assist the correlation of data with a particular sub-directory. It may also be possible and useful to do the same vertically.

Adding support for tagged revisions would also improve the viability of the visualisation. Many version control systems expose a system for tagging particular commits with a message. Marking this on the graph (by a vertical line) would add an additional dimension to the progression of time, and improve the ability for a user to understand data.

4 Software Overview

The heat map generation tool is written in Ruby and JavaScript and uses a wide range of technologies. It is accessed using a command line interface:

```
$ ./generate.rb --help
Usage: generate.rb [options] [git-path]
This script can be used to generate visualisations of git repositories.

  -t template.erb                 Specify the template to use.
  -o output.html                  Specify the output file.
  -p period                       Specify the period of aggregation:
                                Hourly, Daily, Weekly, Monthly, Quarterly Yearly.
  -d depth                        Specify the depth of aggregation
                                  (depth of top level directories).
  -f filter                       Specify the depth of specific branches
                                  (0 means exclude).

Help and Copyright information
      --copy                      Display copyright information
  -h, --help                      Show this help message.
```

The result is an HTML file which contains the visualisation. Several additional assets are required including images, stylesheets and JavaScript source code.

4.1 Period Aggregation

Period aggregation is done per unit length of time. Time is aligned to natural boundaries such as years starting on the 1st of January, months starting on the 1st of the relevant month, etc.

Listing 1: Daily Period Implementation

```
class Daily < Period
    def name
        "Days"
    end

    def key(t, p = 0)
        mktime(t.year, t.month, t.day) + (3600 * 24 * p)
    end
end
```

The main class **Period** provides two key functions.

The function **aggregate** takes a set of values with associated dates, and returns a set of discrete buckets. Each bucket represents a specific time period and contains any values that fell within that period:

Listing 2: Aggregate Implementation

```
def aggregate(values, options = {})
    slots = {}

    values.each do |value|
        time = value[:date]
```

```
        k = key(time)

        slots[k] ||= []
        slots[k] << value
    end

    return slots
end
```

The function **between** takes a start and end date, and returns an array of every period date covering that duration. This is used to generate the main list of periods in the visualisation.

4.2 Trie Filter

The filter provided allows various depths to be explored. In order to efficiently map the filter to the incoming data, a trie is used. The function **trie_depth** returns the depth that a given path should be truncated to.

Listing 3: Trie Depth Implementation

```
def trie_depth(top, path)
    depth = nil

    path.each do |p|
        break if top == nil
        depth = top[:depth] if top[:depth]

        top = top[p]
    end

    return depth
end
```

The initial scan through the source code revisions uses this to determine which bin to place the revision into.

4.3 Partitioning

Revisions are partitioned by dividing the set of files that are changed into two sets - one that is in a particular subdirectory and all others. This is used to efficiently process the binned revisions into subsets. This is required because a revision may contain changes in separate project sub-directories which are to be visualised separately.

Listing 4: Bin Partitioning Implementation

```
        bins.to_a.sort{|a,b| b[0] <=> a[0]}.each do |dir, revisions|
            slice = @slices[dir] = []

            revisions.each do |rev|
                next if rev.changes.size == 0

                a, b = rev.partition_in_directory(dir)
                rev.update_from(b)
```

```
                slice << a
            end
        end
```

4.4 Magnitude Calculation

Once the data has been sliced into both a sub-directory and time period, we then need to calculate the magnitude of the set of data. This data can include multiple revisions, each containing multiple changes to various files. The magnitude function takes two factors into account, the number of revisions, and the average number of lines changed. This is important because a change might be a single line in many files (not so important), or many lines in a single file (very important).

Listing 5: Bin Partitioning Implementation

```
    def magnitude(revisions)
        # Number of commits during this period
        rs = revisions.size

        # Number of files changed during this period
        fs = revisions.inject(0){|total,commit| total + commit.changes.size }

        # Net number of lines changed
        ls = revisions.inject(0){|total,commit| total + ((commit.lines_added + commit.
            lines_removed) >> 1) }

        # Weight together the changes
        m = rs
        m += Math::log10((ls.to_f / fs.to_f) + 1) if fs > 0

        # This is the expansion factor, it changes how compressed the data is.
        # This number is a bit magic, it will change the entire visualisation.
        # The lower this number, the more compressed the data will be.
        f = 60.0

        return Math::log10((m.to_f / f) + 1) * f + 1.0
    end
```

4.5 Colour Interpolation

Colours are interpolated from a sequence of fixed colours.

Listing 6: Colour Interpolation Implementation

```
def temperature(t, colors)
    p = (t.to_f * colors.size)

    if (p <= 0)
        return colors.first
    end

    if (p+1 >= colors.size)
        return colors.last
    end

    return interpolate(p - p.floor, colors[p.floor], colors[p.floor+1])
```

```
end

def background_color(size)
    temp = temperature(ramp(size.to_f), [
        [0xFF, 0xFF, 0xFF],
        [0x4D, 0x85, 0xFF],
        [0xF4, 0xFB, 0x13],
        [255, 0x05, 0x05]
    ])

    return hexcolor(temp)
end
```

4.6 Weighted Map

In order to generate a list of files for the tooltip, a WeightedMap is used. This is simply a set that keeps track of how many times the same element has been inserted. We use this to show the most 'popular' files in the tooltip.

Listing 7: Weighted Maps Implementation

```
class WeightedMap
    def initialize
        @values = {}
    end

    def add(key, weight = 1)
        @values[key] ||= 0
        @values[key] += 1
    end

    def insert(others)
        others.each do |other|
            add(other)
        end
    end

    def sorted
        @values.to_a.sort{|a,b| b[1] <=> a[1]}
    end
end
```

4.7 HTML Generation using Erubis

Erubis[7] is used for generating HTML. An appropriately modern and well designed template is supplied in a separate file (by default **template.erb**). The raw data is provided to the template which then produces the output data. The output format is not limited to HTML; in the first visualisation attempt, the same technique was used to produce VRML.

5 Feedback

A basic qualitative assessment was done by publishing the visualisations for LLVM and Kai online. Feedback was received from several users. The feedback was used to adjust various elements such as colour curves and improve several features such as filtering.

> It's neat, but I think it needs be more flexible in its selection of how deep to drill down – make it a function of the change velocity of each directory. That would likely correspond to what's interesting. So for example lib/Targets is one of the hottest directories, and we know anyway that would be interesting to look another level into (to see which targets are the most active). While some of the others, you might not need to drill beyond the first level.
>
> Looks great, that is very useful to see the different targets broken out. The obsessive in me sees that most of the heat is now on llvm/test can those be broken out a level? llvm/test will be interesting because there are a few (relatively hot directories like llvm/test/Transforms, but also a bunch of code ones. Is it possible to lump the coldest ones into a llvm/test/<other> category or something? This is really cool stuff, but I'm not exactly sure what to do with it.
>
> Overall, excellent. I immediately was able to take away interesting information, like when the work on ARM started in earnest, how it compares to x86, etc. Excellent use of color. The blues and grays are very neutral, and the dark reds are used very sparingly, only to call attention to exceptional data.
>
> Too big! Even in full screen, I can scroll up-down 4 times. I can't keep all that information in my head; there's no way to visually compare the folders at the top with the folders at the bottom. No idea how to address this: perhaps allowing subfolders (e.g. CodeGen, Target) to be folded up together into a summary row?

6 Conclusion

Several methods of visualising information change over time have been explored. The initial attempt encountered a range of technical, format, and implementation limitations. After further effort, many issues were overcome and a successful and distinctive visualisation has been produced. The software can be tuned for projects both of small and large scale effectively, and the implementation is efficient. A basic evaluation was performed, and feedback was used to improve the visualisation.

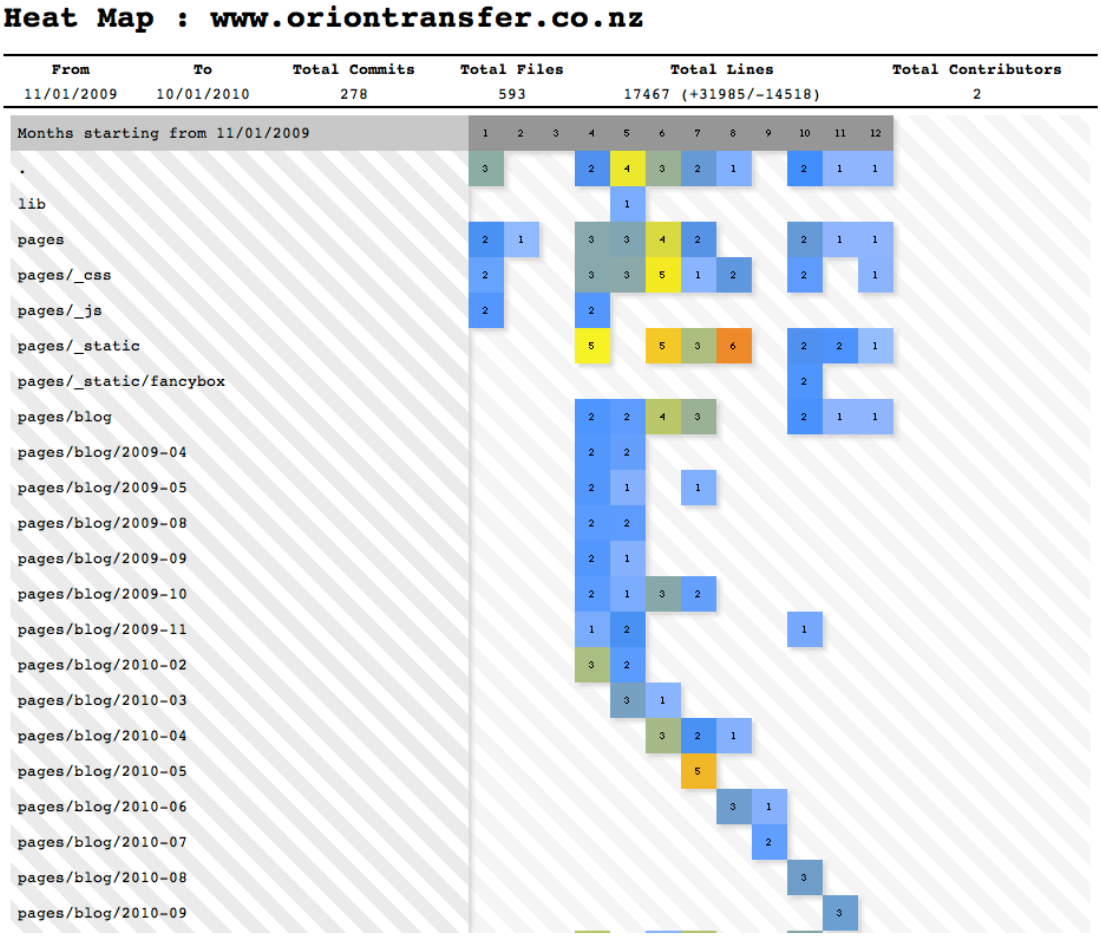

Figure 6: A visualisation of the changes to the source of a website. Note the trend in the changes to the blog.

References

[1] A. Hunt and D. Thomas. Software archaeology. *Software, IEEE*, 19(2):20 –22, mar. 2002.

[2] Gource. Available from: `http://code.google.com/p/gource/`.

[3] Extravis. Available from: `http://www.win.tue.nl/~dholten/extravis/`.

[4] Doxygen. Available from: `http://www.doxygen.org/`.

[5] SLOCcount. Available from: `http://www.dwheeler.com/sloccount/`.

[6] git. Available from: `http://git-scm.com/`.

[7] Erubis. Available from: `http://www.kuwata-lab.com/erubis/`.