The Visual Code Navigator: An Interactive Toolset for Source Code Investigation

Gerard Lommerse* Freek Nossin† Lucian Voinea‡ Alexandru Telea•

Technische Universiteit Eindhoven

ABSTRACT

We present the Visual Code Navigator, a set of three interrelated visual tools that we developed for exploring large source code software projects from three different perspectives, or views: The *syntactic view* shows the syntactic constructs in the source code. The *symbol view* shows the objects a file makes available after compilation, such as function signatures, variables, and namespaces. The *evolution view* looks at different versions of the same source file during a project lifetime. The views share one code model, which combines hierarchical syntax-based and line-based information from multiple source files versions. We render this code model using a visual model that extends the pixel-filling, space partitioning properties of shaded cushion treemaps with novel techniques. We discuss how our views allow users to interactively answer complex questions on various on various code elements by simple mouse clicks. We validate the efficiency and effectiveness of our toolset by an informal user study on the source code of VTK, a large, industry-size C++ code base.

CR categories: H.5.2[User interfaces]: evaluation/methodology; I.3.2 [Graphic systems]: stand-alone systems; J.7 [Computers in other systems]: command and control

Keywords: source code visualization, multiple views, treemaps, pixel-filling displays, source code analysis

1. INTRODUCTION

Program understanding is an important aspect of software maintenance. Current industrial projects are often based on collaborative development of millions of code lines. Industry practice studies have shown that maintainers spend 50% of their time on understanding this code [14]. In this context, software visualization addresses several facets of program and process understanding, such as reverse engineering and process recovery and execution and algorithm animation.

In this paper, we focus on the first goal. We introduce the Visual Code Navigator (VCN), a toolset that provide three different, but strongly interconnected views on software source code. VCN aims to help the developer understand the structure, and its changes, of large software systems.

*e-mail: g.l.p.m.lommerse@student.tue.nl
†e-mail: f.nossin@student.tue.nl
‡e-mail: lvoinea@win.tue.nl
•e-mail: alext@win.tue.nl

We start solely from the code base, i.e. a set of source code files, since this is often the only up-to-date, reliable source of information on a software project, and also the main material involved in the maintenance phase. Moreover, we believe that visualization tools must be tightly and non-disruptively integrated with the programmers' accepted way of working, i.e. directly reflect, and make use of, the source code itself.

VCN consists of three interrelated tools, or views, on the code base. The *syntactic view* shows the code syntactic constructs, such as classes, functions, statements, identifiers, and so on. The *symbol view* lets one see the symbols the code base makes available after compilation, e.g. functions, variables, and namespaces. Finally, the *evolution view* shows several versions of several source files during a project lifetime. Each view visualizes a different hierarchical aspect of the code base: text lines in file versions in the *evolution view*, syntactic structures in the *syntactic view*, and data/code symbols in the *symbol view*. The views use the same visual model to show hierarchical containment, based on the space partitioning properties of shaded cushion treemaps. For this, we extend the shaded cushions, as introduced in [23], with several new techniques. The three views are dense pixel displays that map up to thousands of source code artifacts on a single screen, allowing efficient zoom and pan techniques to quickly get overviews of large code bases and also details on demand.

The structure of this paper is as follows. In Section 2, we review related work on source code visualization. Section 3 outlines the code data model, i.e. how we acquire and represent source code information. Section 4 presents the three different, but correlated, views that visualize source code in VCN, and discusses several design and implementation techniques. In section 5, we illustrate our tools and techniques with the analysis of the source code of VTK, a large and complex C++ class library. Finally, section 6 discusses the contribution we bring to source code visualization and outlines future research directions.

2. RELATED WORK

We define the goal of source code visualization using the five dimensions model of Maletic *et al* [10]: task, audience, target, medium, and representation. The main **task** is to gain insight in the structure and semantics of a given code base. Specifically, we focus here on detailed, code-level understanding, as compared e.g. to getting architectural overviews of a code base [17]. The intended **audience** is mainly composed of code developers and maintainers, i.e. persons directly interested and involved in writing and manipulating source code as such, as opposed to e.g. system architects, who often manipulate code in terms of black-box components or packages rather than textual source. However, given the overview capabilities provided by the nested cushion techniques we use, our audience includes also system architects who need to quickly get condensed snapshots of a whole subsystem, and, from there, drill down to specific code details. The **target** of visualizations is the code base, i.e.

the set of source code files involved in a project. We are also interested in grasping the source code evolution, so we assume the code is stored in a version control management system, such as CVS. We focus here on code written in C/C++, so our visualizations use, and reflect, language-specific syntactic information. The intended visualization **medium** is the standard PC graphics display used by most integrated development environments (IDEs). Finally, the **representation** is formed by dense-pixel displays that use shaded cushions to convey code hierarchy.

The challenge of source code visualization has been addressed by several tools. At one end of the granularity spectrum, tools such as Rigi [18], SHriMP [15], or SoftVision [16] are used in reverse engineering to understand subsystem structure and dependencies. The code base is a hierarchy of functions, classes, components, or packages. Usually, these tools do not reflect the source code layout itself, but strive to optimize the spatial code graph layout. Moreover, such tools do now show low-level system details, such as the many, minute source code edits done during debugging or the inner control flow of modules.

At the other end of the spectrum, tools such as SeeSoft [2] and Augur [5] offer a line-oriented code representation colored by code attributes and metrics. These tools use the assumption that developers are comfortable with visualizations that present the code in the same spatial context in which it is constructed (i.e. written). While maintaining the 2D code line layout from a source file, they reduce a code line to a pixel line, thus condensing tens of thousands of lines on a single screen. This idea was first proposed by Eick et al. in SeeSoft [2] and was further refined by several tools: Augur, Aspect Browser, GSee, sv3D, Tarantula, Gammatella, Almost and CVSscan. Augur [5] combines information about artifacts and activities of a software project at a given moment. Aspect Browser [7] uses regular expressions to locate specific artifacts (e.g. keywords) and visualizes their distribution. GSee [4] tries to bridge the gap between fine and coarse-grained visualization tools, combining both elements in an orchestrated environment. sv3D [11] uses a 3D line-based code representation to compact the space needed for display. Tarantula [9] is the first tool to use color and a line-oriented display to present test line coverage overviews. A similar approach is proposed by Almost [13], a program trace visualization tool. This idea is further developed by Gammatella [12], which uses treemaps to show system-level test file coverage overviews. CVSscan [20] is the first to visualize by a line-based display the entire evolution of a file.

However, these tools mostly fall at one or the other end of the granularity spectrum. The ones that try to bridge the gap, e.g. GSee [4], Gammatella [12], bring merely fine (code lines) and coarse grained (files) elements together, and bind them through correlations. In this paper, we attempt a finer sampling of the granularity scale. We use line based displays and treemaps to depict also middle-sized artifacts such as syntactic constructs (e.g. function and class bodies) as well as artifacts resulting from compilation (e.g. namespaces and variables). We combine our multi-scale visualizations in an orchestrated environment that offers details on demand and enables code correlations across multiple versions.

3. CODE DATA MODEL

Our data source is the information stored in the CVS version control management system. This consists of several versions V_{ij} of several source files F_j, as well as, for every version, its commitment date (time when it was added to the CVS repository), and author (who committed it). To decouple CVS from the visualization itself, data extraction is done by a separate tool. Our main visualization focus is a set of related source files F_j, called a project. Once all versions V_{ij} of all files in a project are extracted, we pass them to a syntax fact extractor built by us. This is a modified GNU C/C++ compiler, with no code generation, which extracts the annotated syntax tree from the source code. This contains elements such as classes, data members, function signatures, macros, templates, for-loops, etc. [3]. Our data model contains thus, for every project P, the source code of all P's files and, for every file F_j, an annotated syntax tree with all constructs in F_j. Essentially, our code data model captures the *whole* information range from local details, such as the contents of every single code line, to global structure, such as the global syntactic constructs in files and files in project structuring. Essentially, the above resembles the DATRIX code model extracted by the CPPX C++ source code analyzer [8].

The code data model is thus a set of related hierarchies, visualized by our three views, as follows. The *evolution* view shows the file versions in a project's evolution, the files in a project, and the text lines in a file. The *syntactic* view shows all syntax constructs in a file's syntax tree, i.e. reflects an implementation perspective. The *symbol* view shows the nesting of symbols in a compiled (object code) project, e.g. function signatures, classes, and data objects in namespaces, i.e. uses an interface perspective. Given that the hierarchical code relations are closely interrelated, our three views also have tightly coupled functionality. This is presented next.

Throughout the presentation, we use code examples from the VTK code base. More details about our visualization case study are given in Section 5.

4. VISUALIZATION MODEL

We have chosen to use *shaded cushions* to represent the various hierarchies present in our data model (Section 3), for several reasons. First, shaded cushions can show up to thousands of elements in tree-like hierarchies on a single screen, as shown by various applications [23][19][20]. This is essential, since we want to visualize large, real-world code bases. Second, cushions combine best with two-dimensional spatial layouts. This serves us well, as we believe our target developer user group is most confident and comfortable with 2D layouts. Moreover, if we do not impose specific constraints on the layout to use, cushions can be combined with treemaps to make the most of the available screen space for displaying hierarchies. We use this cushion treemap combination in our symbol view (Section 4.2). In the *syntactic* and *evolution* views, we choose to use a different, code-oriented layout: the x axis maps the files visualized together, and the y axis maps the lines in a file. Here, we give up the treemap layout, but can still use cushions to effectively convey the hierarchy, by using a new cushion shape and shading (Section 4.1). Third, we implement several efficient cushion rendering methods, which allow us to perform interactive zoom and pan in our views, an essential aspect from a usability perspective. Finally, we use color encoding to display code attributes via the cushions.

We detail next the design and implementation of our three views.

4.1 Syntactic view

The *syntactic* view shows the syntactic constructs in a given file. For every construct, we render a cushion whose geometric

outline encompasses the construct's text extent. Given the way constructs nest in source code, a cushion has the dotted line shape sketched in Figure 1a, consisting of one 'body' rectangle of size (w, h-2h_l) and two 'indent' rectangles of sizes (w_s, h_l) and (w_e, h_l), where w_s and w_e are the start and end indents respectively and h_l the font height. Note, however, that this is the most complex shape cushions may have, and occurs only in presence of extreme code indentation. Usually, code cushions consist of a body rectangle and, optionally, the lower indent rectangle (see Figure 3).

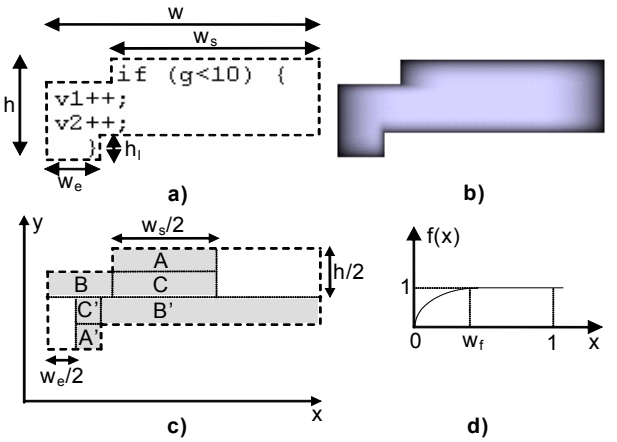

Figure 1: Code cushion design

The code cushion we propose is shown rendered in Figure 1b. To create this rendering, we extend the rectangular cushions [23], as follows. First, we define a 1D height profile f. This is a function with range [0..1] that consists of a bevel of width w_f followed by a plateau region (Figure 1d). For the bevel, we used various monotonically increasing functions with the conditions $f(0) = 0$, $f(w_f) = 1$, $f'(w_f) = 0$. In practice, we found an elliptic sector for f and a value w_f of 0.3 to give the most aesthetically pleasing results. Next, we construct a 2D profile $T(x,y) = g(x)\,g(y)$, where

$$g(x) = \begin{cases} f(x), & x \in [0,1/2] \\ f(1-x), & x \in [1/2,1] \end{cases}$$

This profile is essentially similar to the parabolic one used by the original cushion treemaps [23], with the difference that it has a plateau (flat) region in the middle of width 1 - 2w_f, as sketched by the dotted profiles in Figure 2. However, if we directly use T as height profile scaled over the rectangle [0..w,0..h] for our code cushion, we do not get the result in Figure 1b. Indeed, T does not have zero values on the concave indent borders (left, top, bottom, and right borders of rectangles A, B, B', and A' respectively in Figure 1c). To achieve this, we use a cushion height profile $H(x,y)$ defined as follows:

$$H(x,y) = \begin{cases} T & , x \notin \mathrm{A} \cup \mathrm{B} \cup \mathrm{C} \cup \mathrm{A'} \cup \mathrm{B'} \cup \mathrm{C'} \\ T\ T_x^{\mathrm{A}} & , x \in \mathrm{A} \\ T\ T_{-y}^{\mathrm{B}} & , x \in \mathrm{B} \\ T\ T_x^{\mathrm{C}} T_{-y}^{\mathrm{C}} & , x \in \mathrm{C} \\ T\ T_{-x}^{\mathrm{A'}} & , x \in \mathrm{A'} \\ T\ T_y^{\mathrm{B'}} & , x \in \mathrm{B'} \\ T\ T_{-x}^{\mathrm{C'}} T_y^{\mathrm{C'}} & , x \in \mathrm{C'} \end{cases}$$

To explain the above, consider the subdivision of the cushion shape sketched in Figure 1c. Outside the six border rectangles A, B, C, A', B', and C', the profile H coincides with the rectangular profile T. Inside these rectangles, we multiply T with four bias functions T_x, T_{-x}, T_y, and T_{-y}, in order to obtain a profile H that is zero at the outer cushion border and increases towards the inside as specified by the 1D profile f depicted in Figure 1d. The bias functions T_x, T_{-x}, T_y, and T_{-y} are simply translated, scaled, and rotated copies of f. If we think of them being luminance textures, then, given a rectangle R with origin (R_x,R_y) and sizes (R_w,R_h), then $T_x^R(x,y) = f(x\text{-}R_x)R_w$, $T_y^R(x,y) = f(y\text{-}R_y)R_h$, $T_{-x}^R(x,y) = f(R_x\text{-}x)R_h$, and $T_{-y}^R(x,y) = f(R_y\text{-}y)R_h$. Practically, we construct H by first rendering the code cushion textured with a luminance texture T stretched over the rectangle [0..w,0..h]. Next, we render the rectangles A, B, C, A', B', and C' textured with the 1D texture f, scaled, translated, and rotated as described above, using multiplicative blending. Figure 1b shows the final result.

To render a nested code structure, we must now combine the cushion height profiles H defined above. The original solution for this [23] was to sum up parabolic height profiles and illuminate the result (Figure 2a). In our case, since $w_f < 1$ (in practice, $w_f = 0.3$), the profiles of the deeper nested cushions (Figure 2b, continuous line) fall within the plateau (flat) region of the enclosing cushions (Figure 2b, dotted line). We can thus obtain the same result as [23] by simply rendering the cushions in nesting order. This has the advantage that it does not require high precision height profile summing, so can be cheaply implemented using standard fixed-point OpenGL. Summarizing, our rendering runs over all syntax constructs, in nesting order, and renders their cushions using polygons textured with the cushion profile, stored as luminance OpenGL textures. A chief advantage of this efficient rendering is that users can interactively zoom and pan tens of files of thousands of lines of code in total (see e.g. Figure 6).

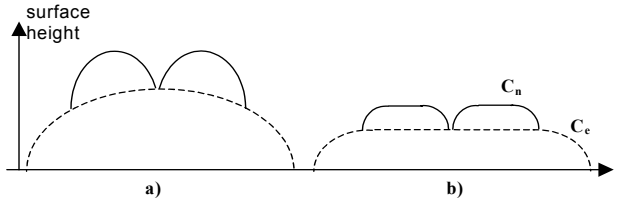

Figure 2: parabolic cushions (a) and plateau cushions (b)

Figure 3 shows our method for a code fragment of nesting depth 5. The eye perceives the height of the cushions to be proportional with the nesting depth. This result, known as the Cornsweet effect, was previously used to similar ends in visualizing nested height contours [22]. This works if the nested structures do not completely overlap, as they would in case of a treemap where parents are completely covered by their children. In the syntactic view, complete overlaps never happen. We prevent this by extending the right cushion margin with a small border. This is the reason why the cushions in Figure 3 visibly extend past the text's right border. In the other three directions, cushion margins exactly match the text, a constraint imposed by the text layout itself.

The next step in the syntactic view design is to combine the cushion hierarchy with the code text itself. Our goal is to have a tool in which programmers can smoothly and easily navigate between the familiar, trusted text view a standard editor would offer and the syntactic cushion enhancement. We achieve this by

blending the text graphics atop of the cushions. Users can control both text (α_t) and cushions (α_c) transparency via two sliders to instantly change the visualization focus from text to syntax.

Figure 3 (left) depicts this by showing clear text (α_t=1) over a faint syntactic background (α_c=0.2), and a faint text (α_t=0.3) over a strong syntactic background (α_c=0.6).

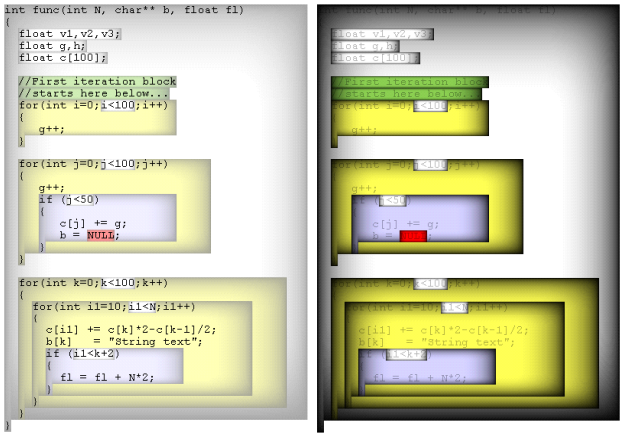

Figure 3: Cushion and text blending

Finally, we color cushions to show the type of syntax construct they display. Users can browse all C/C++ syntax constructs in a tree view widget and change their color and visibility. For example, in Figure 3 we used yellow for `for` loops, green for comments, gray for functions, light blue for `if` statements, white for declarations and conditions, and red for macros. Visibility is turned off for many constructs, such as the finer-grained declarators and expression terms, to avoid non-informative cluttering. In this way, we focus on the larger extent constructs, such as scopes, which help us grasp the overall program structure. By manually changing the colors, or choosing from predefined color schemes, we can quickly focus on various code aspects and answer queries such as "show all iterations (for, while, do)", "is the code heavily using macros?", "is the code deeply nested?" or "is the code richly commented?". More examples hereof are presented in Section 5.

Figure 4: Cushion profile choices

By tuning the range of the profile function f (Figure 4d, f=0), we can obtain effects ranging from classical 'syntax highlighting' (Figure 4 left) to a soft 3D shaded bevel effect (Figure 4 right, $f \in [0..0.5]$), or a high-contrast effect (Figure 3 right, $f \in [0..1]$). The combination of cushion display and text graphics allows a smooth level-of-detail control. By changing the font height h_l, which implicitly changes the cushion sizes, one can change the amount of code visualized on a single screen. In the extreme case when h_l=1 pixel, the syntactic view becomes very similar to a line-based source code visualization such as SeeSoft [2]. shows 11 files visualized in this way in the syntactic view. The largest file is a C++ implementation file of 635 lines, the other 10 ones are header files. In total, this screen shows over 3400 code lines. Via the color coding (same as in Figure 3), we quickly see several things, as follows. The implementation file (leftmost) contains heavy iterations (yellow), but no deep nesting. The header files are clearly much richer commented (green) than the implementation. All files share the same initial comment block, i.e. the first green block at the top. Every header contains one large class (cyan). This class is almost the first thing declared in the headers, as there is not much else atop the cyan block. Some of the headers contain also inline functions (the white blocks below the cyan class block). There is no heavy use of macros, except in one file, as shown by the immediately salient red spot (A). For comparison purposes, the same code is shown in Figure 6 (right), without the shaded syntax cushions. Obviously, it is almost impossible to grasp the code structure from this image.

Another important issue is navigation in the syntactic view. As described above, code overviews can be generated by zooming out (decreasing the font size), fading out (decreasing the text opacity), and marking certain syntactic elements as invisible. Users can navigate this code overview by scrolling the rendered code columns. Why would navigating this overview be better than scrolling text in a classical editor? First, our experience is that, when looking at some code detail, programmers often have requests such as "I want to go to the start of the third previous function in this file", "go to that deeply nested for loop somewhere below this point", or "go to that code fragment somewhere in the beginning of the file, below that richly commented code". Scrolling the syntax-colored, cushioned code overview serves exactly these requests, as one quickly sees the size, nesting, type, and distribution in file of the source code. However, often one needs details on demand as well. We provide these by displaying, at any time, the detailed code under the mouse position in a separate classical text editor view under the cushion view (Figure 6 left). However, this can disrupt the navigation process, as the user must continuously change focus between the cushion and detailed view. We address this problem by providing two details-on-demand cursor modes: the *spotlight* cursor (Figure 5 left) and the *syntax* cursor (Figure 5 right). Following the model of Furnas [6], both cursors modulate the text and cushion opacity based on the degree of interest (DoI), which is a function $f(d)$ of the geometric distance d from the point of interest located at the mouse cursor (depicted as a cross in Figure 5). For the spotlight cursor, we use $f(d) = \frac{1}{2}(\cos(\frac{Kd}{\pi}) + 1)$, where K gives the spotlight width, which is efficiently implemented by using an alpha texture. For the syntax cursor, we simply set $f(d)$ = 1 for the cushion right under the mouse.

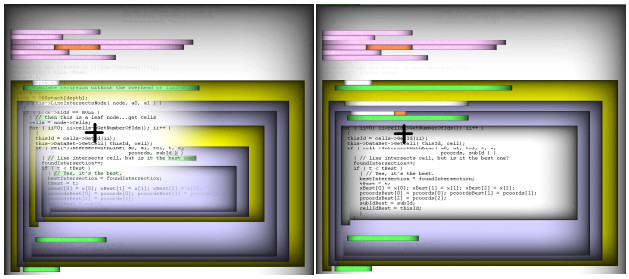

Figure 5: spotlight cursor (left) and syntax cursor (right)

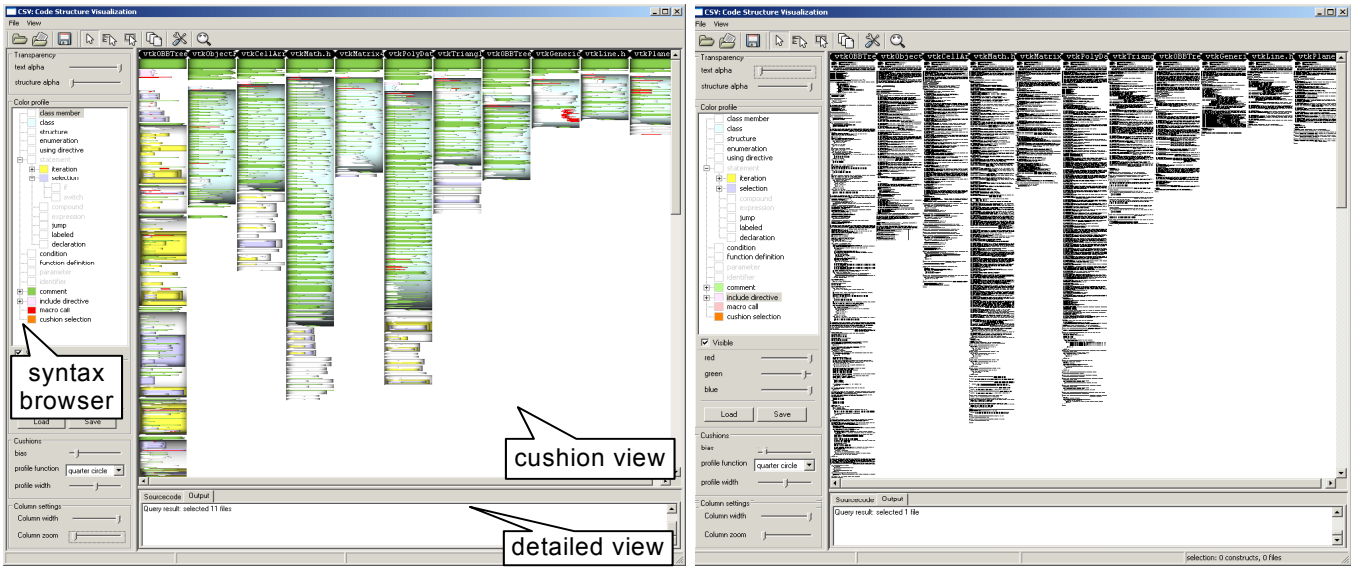

Figure 6: Syntactic view of 11 files with cushions (left) and without (right)

Summarizing, a typical use scenario for the syntactic view is as follows. First, the user opens the files of interest, selects a small font height (e.g. one pixel), and chooses a predefined cushion color scheme for the types of syntax structures he wants to see (Figure 6 left). If some area of interest pops up, e.g. a particular cushion color, size, and nesting mix, the user zooms on it increasing the font height. Next, the spotlight cursor is activated, and the area is brushed to see the concrete code at hand. Typically, an interesting area of the size of a function or class, of about 100 lines of code, gets now in focus (Figure 5 left). Next, the syntax cursor is activated to further zoom on a substructure of interest, such as the `for` loop in focus in Figure 5 right. At this point, the cushion opacity is typically decreased in favor of the text, and the user can further inquire the code using context-specific queries. Queries usually move the point of interest to some other part of the code base, where the process starts again anew.

4.2 Symbol view

The second tool in the VCN system is the *symbol* view. This is a shaded cushion treemap that displays the symbols a project contains after compilation, i.e. the symbols a linker would see at that stage. In C/C++, these are all global scope objects: function signatures, class and namespace method and data members, templates, enumerations, typedefs, and global variables [3]. The symbol view is correlated with the syntactic view: One can select a hierarchical code element (file, scope, variable, declaration and so on) in one view and see it depicted in the other too. Note, however, that the symbol view does not show any data inside function bodies. This information is inexistent from a linker perspective.

The symbol view treemap shows the C++ scope containment, as extracted by our code analyzer (Section 3). Treemap nodes are colored by their type: green for functions and methods, red for variables and data members, cyan for typedefs, and yellow for function arguments (Figure 7). Less saturated colors indicate symbols in the global scope, more saturated ones symbols in class or namespace scope. We also visualize the (included) files the symbols come from, as a separate subtree, colored orange. The size of terminal (leaf) symbols is equal to their byte count, or `sizeof` C operator, except for function bodies. Since the symbol view has no information over these, we set their size to be the number of code lines of the function signature (declaration). Non-leaf symbol sizes are the sum of their children's sizes. The above metric allows comparing objects to answer questions like "which are the 'heaviest' variables or types in a scope?" or "which are costly functions from a parameter passing perspective?" The latter is easily answered, since function arguments are children of function declaration nodes. Other questions the symbol view can answet are "where are the inlines or template declarations?", "which file contains just function prototypes?", "which classes have the largest interface (most methods)", or "which declarations are heavily commented?" The last is easily visible, since richly commented declarations have more code lines than their 'bare', code only counterparts.

Figure 7 shows the symbol view for the standard C/C++ headers (e.g. stdio.h, iostream.h, etc) included by a VTK file. The space is partitioned in three: the files (orange, below), the global C namespace (treemap left half), and the C++ `std` namespace (treemap right half). Further partitioning of the `std` namespace indicates its various classes, e.g. `iostream`, `vector`, `list`, etc. We see that the headers are dominated by function prototypes (small green and yellow cushions). There are just a few functions having more than one code line (the few larger green cushions). The global C scope has overall functions with the same argument list size (uniform distribution of small yellow cushions in the left treemap half). In contrast, the `std` namespace has much more variation in function argument size (varying-size yellow cushions, right treemap half). This is so since `std` template methods often use relatively large iterator types as arguments. There are just a few variables (red) in the headers, which matches expected good programming rules. A relative surprise is the small number of `typedef` declarations (cyan). We expected these would be more numerous in standard headers.

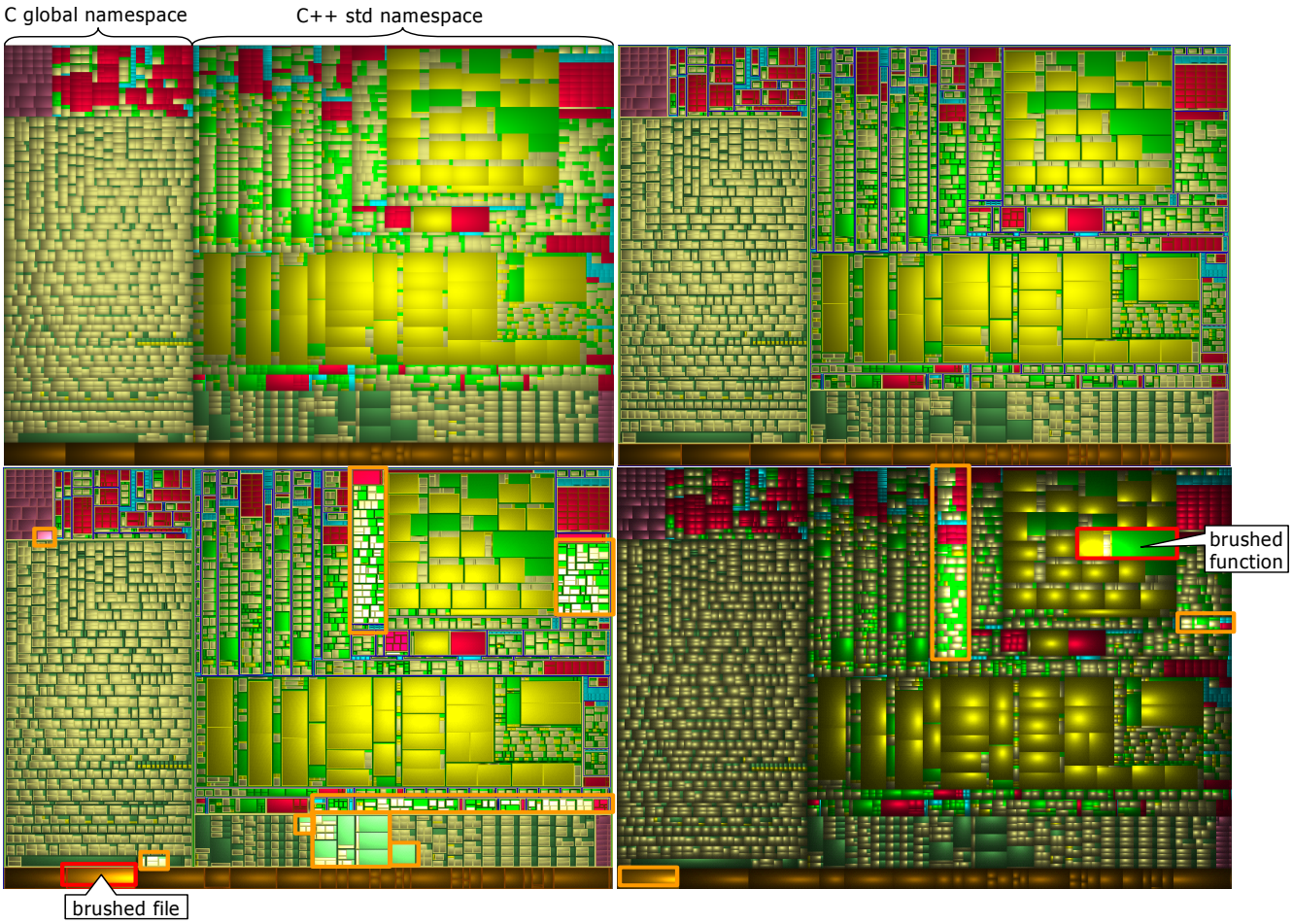

Figure 7 : Symbol view. Borderless (top left) and bordered treemap (top right) and visual brushing for queries (bottom row)

We tested two variations of the symbol view: a classical borderless cushion treemap (Figure 7 top left) and one with one-pixel borders (Figure 7 top right). Remarkably, all our users found the bordered treemap easier to understand, as the nesting is clearer. Moreover, scopes (class, structure, and namespace) appear here explicitly as thin, dark blue borders.

The symbol view offers several interactive queries. Brushing over a file (Figure 7 bottom left, red rectangle) shows all symbols defined in that file as highlighted cushions (same figure, orange rectangles). Brushing a function (Figure 7 bottom right, red rectangle) shows the file is declared in and the types of its arguments and return (same figure, orange rectangles). Finally, Figure 7 (bottom right) illustrates another cushion treemap feature. By tuning the cushion height profile, deeper nested structures become more visible, as compared e.g. to the other three images in Figure 7, which better convey the scope (shallow) nesting of code symbols.

We implemented the symbol view's shaded cushion treemap using the `ARB_fragment_program` OpenGL extension. Compared to the software rendering used in [23], this allows us to zoom and pan the view in real time, as well as quickly change the light direction to better grasp the structure nesting.

4.3 Evolution view

The *evolution* view is the last source code view in the VCN. As its name says, it displays the evolution, or change, in the source code of several files in a project's lifetime. This view uses a 2D pixel-filling display based on the file layout, similar to the syntactic view. For a file, the x axis maps the version number and the y axis maps the line number. In the data model from Section 3, the evolution view shows, for a file F_j, all its versions V_{ij} stacked along the x axis. A version V_{ij} is drawn as a vertical pixel stripe, every horizontal pixel line representing one or more source code lines in V_{ij}, colored by the line type or the line author, as described below. To separate between versions, we blend a vertical cylinder-like shaded cushion profile over each version colored stripe. In detail, the evolution view uses the pixel-filling code display technique described by Voinea *et al.* in [20] for a single file evolution. Here, as a new element, we correlate the evolution of several files F_j from a project, by displaying the visualizations of all F_j in a matrix: Every row displays a file, every column displays an attribute type (line type, author, etc), aligned on version number. Figure 8 shows an evolution view of three files F_1, F_2, and F_3 from the VTK code base. For every file (matrix row), 110 versions V_j, j=[1..110] are shown. The largest versions in this picture have about 650 lines. The left matrix column shows the source code colored by line

type: green = comments, black (dark) = function declarations, pink = strings, and blue = C/C++ code, shaded by the nesting level (darker = deeper nested). The right matrix column shows the lines' authors, i.e. the persons who committed the respective lines in the CVS repository (see Section 3).

The evolution view allows us to see several facts in the source code. First, we quickly get an *overview* of the file size evolution in time. Large changes between consecutive versions, denoting major code rewriting, are easy to spot, such as version 50 for F_3 (bottom row) and version 77 for both F_1 and F_2. Moreover, sharing the same time axis allows quickly *correlating* whether changes in different files happen at or around the same version. If so, this is a sign an important change occurred which influenced more than just one file. We noticed this sign several (about 10) times in our VTK case study.

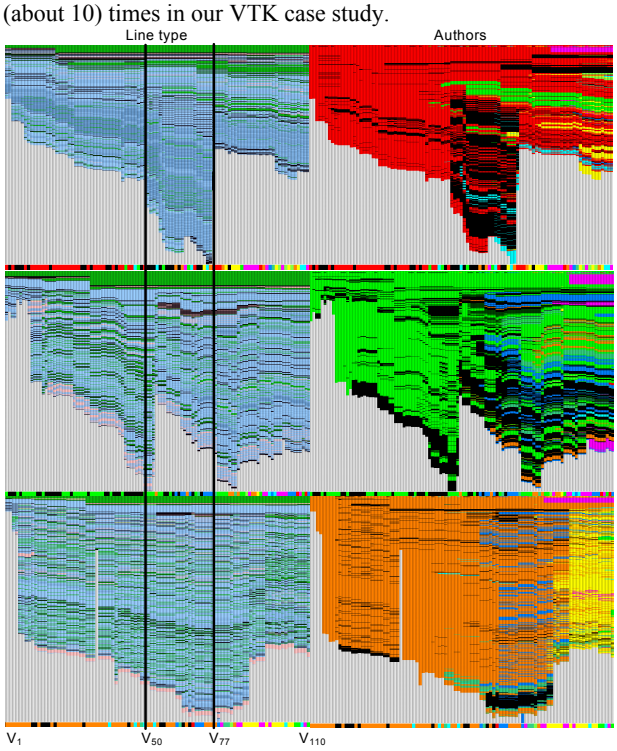

Figure 8: Evolution view of three files, one on each row

Second, we get an overview of the *impact* every developer (author) had on the source code. For example, the red author is obviously the main responsible for F_1, the orange one for F_2. The green one has done most work on F_2, but also starts contributing from around version 50 to F_1. The black author worked a little bit in all three files. Besides author impact, the right column shows also how persistent was someone's work, i.e., whether it survived to the last (rightmost) version or was deleted by someone else. For example, the top-right image in Figure 8 shows that much of the black author's work in the middle of the evolution of F_j (versions 50 to 77) is actually thrown away by the major rewriting at version 77. Code persistence can be detected by looking after continuous 'wavy' stripes having the same color in the author view, which span several contiguous versions. If the same stripe pattern is visible also in the line type view, the probability we have a persistent code fragment is strengthened even more. By correlating mouse-based brushing in the evolution view with detailed code display in the syntactic view, one can examine the actual facts to validate, or infirm, the formed hypotheses.

5. CASE STUDY: THE VTK SYSTEM

We have used VCN's three views in a study to understand VTK [21]. VTK is a complex C++ library of hundreds of classes in over 2000 files, spanning over 100 versions, developed by tens of programmers over a 10 year period. Our three users, who were experienced with C++ but never used VTK, acquired 100 versions of several VTK files with CVSgrab, analyzed them using VCN, and addressed several questions: (Q1) Are VTK files fine-grained (many small functions) or coarse-grained (few large functions)? (Q2) What is the typical structure of a public VTK class? (Q3) Can you locate and describe a few large structural changes in the VTK evolution? (Q4) What is the typical frequency and usage of macros in VTK? (Q5) Is VTK code richly commented? A fourth user, with over seven years of VTK programming experience, specified the VTK files to analyze and assessed the answers delivered by the former users after about two hours of investigation.

Several snapshots reflecting the typical use of VCN during our study have already been shown in the previous sections. Overall, our users had similar observations at the end of the study, as follows. Q1 and Q2 were quickly answered by both the syntactic view (e.g. Figure 6a) and the line metric symbol view (Section 4.2). The studied files had functions of 30..100 lines of code, except some small inlines in the headers. The VTK classes have a rich public interface and relatively fewer protected and private members. Here, the symbol view performed better than the syntactic view, as it shows classes and their members as a simple treemap, compared to the more complex shaped code cushions. The evolution view was used for Q3, the structural changes found are discussed in Section 4.3. Q4 was answered by the syntactic view by using red as macro color (Figure 6 left, third file-column from right). Finally, Q5 was answered equally well by the syntactic and line-type colored evolution views.

A prototype of the VCN including a pre-parsed VTK code base is available at **www.win.tue.nl/~lvoinea/VCN.html**

6. CONCLUSION AND FUTURE WORK

We have presented VCN, a toolset that offers three views, at different level of detail, on source code structure and evolution: The syntactic view shows code structure, the symbol view shows code from a linker's perspective, and the evolution view shows code changes during a project's lifetime.

Both the syntactic and symbol view visualize C++ syntax construct nesting. The syntactic view shows *all* constructs, including e.g. the function body implementations, and targets the code writing and debugging phase. In contrast, the symbol view shows only constructs a linker would see *after* compilation, so it offers a coarser view on the software.

The evolution view and (zoomed out) syntactic view share basically the same 2D pixel-filling, file-based layout. One may wonder whether to merge the two in one view, capable of rendering code at several levels of detail. From fine to coarse, these would be: text (classical editor), syntax cushions over variable-height font text (syntactic view), one pixel per character rendering (as in [1] and [2]), and several code lines per fixed-width pixel stripes (the evolution view and [20]). Conceptually, we see no problem with this. However, having already implemented all the above, we see several technical problems in crafting one single, efficient implementation that would address all above visualizations and their underlying code data models. Another interesting note from our case studies is that code perception varies discontinuously when we continuously vary

the text scale (font size). The evolution view targets a perception scale at which individual code structures (loops, declarations, etc) become unimportant, and only the file size and overall code change patterns throughout the file are relevant. The next discrete perceptual scale is the syntactic view, where code structure itself becomes relevant. The next scale is the usual text editor view where one focuses on the meaning of individual words. Hence, users like to have these views separate, next to each other, rather than combined in one.

Apparently similar, the syntactic view fundamentally differs from the so-called 'syntax highlighting' built in many code editors. The latter only emphasizes individual lexical tokens and, at most, comment blocks. Our syntactic view visualizes the full syntax construct range present in the language, so it generalizes syntax highlighting, which, thus, would better deserve the name 'lexical highlighting'.

Our tools and techniques are immediately applicable to other programming languages besides C/C++, if appropriate syntax extractors (parsers) are available. Our choice for C/C++ was motivated by their widespread use in industry-size projects. Moreover, C++ code is well known for its complexity, so visualization tools have here their best chance to prove themselves.

We plan to extend VCN to explore new dimensions of source code, such as visualizing change at file group (directory) and project level, and computing and visualizing structural, instead of line-based, code change. Our goal is to smoothly integrate VCN in the code development and maintenance cycle to validate and promote the use of visualization in software engineering.

REFERENCES

[1] Ball, T., and Eick, S., "Software visualization in the large", *IEEE Computer*, 29(4), 1996, pp. 33-43

[2] Eick, S. G., Steffen, J. L., and Sumner, E. E., "SeeSoft --A Tool for Visualizing Line Oriented Software Statistics", *IEEE Trans. on Software Engineering*, 18(11), 1992, pp. 957-968

[3] Ellis, M.A., and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[4] Favre, J.M., "GSEE: A Generic Software Exploration Environment", *Proc. IWPC'01*, IEEE CS Press, 2001, pp. 233

[5] Froehlich, J., and Dourish, P., "Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams", *Proc. ICSE '04*, IEEE CS Press, 2004, pp. 387 – 396

[6] Furnas, G., "Generalized Fisheye Views", *Proc. CHI '96*, ACM Press, 1996, pp. 16-23

[7] Griswold, W.G., Yuan, J.J., and Kato, Y., "Exploiting the Map Metaphor in a Tool for Software Evolution", *Proc. ICSE '01*, IEEE CS Press, 2001, pp. 265 – 274

[8] Holt, R., A. Hassan, B. Laguë, S. Lapierre, and C. Leduc, *E/R Schema for the Datrix C/C++/Java Exchange Format,* Proc. WCRE '00, IEEE CS Press, 2000, pp. 284-287, See also: swag.uwaterloo.ca/~cppx

[9] Jones, J.A., Harrold, M.J., and Stasko, J., "Visualization of Test Information to Assist Fault Localization", *Proc. ICSE '02*, ACM Press, 2002, pp. 467 – 477.

[10] Maletic, J.I., Marcus, A., and Collard, M.L., "A Task Oriented View of Software Visualization", *Proc. VISSOFT '02*, IEEE CS Press 2002, pp. 32-40

[11] Marcus, A., Feng, L., and Maletic, J.I., "3D Representations for Software Visualization", *Proc. ACM SoftVis '03*, ACM Press, 2003, pp. 27 – 36

[12] Orso, A., Jones, J, and Harrold, M.J., "Gammatella: Visualization of program-Execution data for deployed software", *Proc. ACM SoftVis '03*, ACM Press, 2003, pp. 173 - 188

[13] Renieris, M., and Reiss , S.P., "ALMOST: exploring program traces", *Proc. NPIVM'99*, ACM Press, 1999, pp. 70-77

[14] Standish, T.A., "An Essay on Software Reuse", *IEEE Trans. on Software Engineering*, 10 (5), Sep. 1984, pp. 494 — 497

[15] Storey, M.A., Best, C., Michaud, J., Rayside, D., Litoiu, M., and Musen, M., "SHriMP Views: an Interactive Environment for Information Visualization and Navigation", *Proc. CHI '02*, ACM Press, 2002, pp. 520 – 521

[16] Telea, A., Maccari, A., and Riva, C., "An Open Toolkit for Prototyping Reverse Engineering Visualization", *Proc. IEEE VisSym '02*, The Eurographics Association, 2002, pp. 241 – 251.

[17] Tilley, S.R., Wong, K., Storey, M., and Müller, H.A., *Programmable Reverse Engineering*, Intl. Journal of Software Engineering and Knowledge Engineering, vol. 4, no. 4, World Scientific, 1994, pp. 501-520

[18] Tilley, S.R., Wong, K., Storey M., and Muller, H.A., "Rigi: A visual tool for understanding legacy systems", *International Journal of Software Engineering and Knowledge Engineering*, December 1994

[19] Voinea, L., Telea, A., and Van Wijk, J.J., "A Visual Assessment Tool for P2P File Sharing Networks", *Proc InfoVis'04*, IEEE CS Press, 2004, pp. 41-48

[20] Voinea, L., Telea, A., and Van Wijk, J.J., "CVSscan: Visualization of Code Evolution", to appear in *Proc SoftVis '05*, ACM Press, 2005

[21] VTK Web Repository: http://www.vtk.org/get-software.php#cvs

[22] Van Wijk, J.J., and Telea, A., "Enridged Contour Maps", *Proc. IEEE Visualization '01*, IEEE CS Press, 2001, pp. 69-74

[23] Van Wijk, J.J., and van de Wetering, H., "Cushion treemaps: visualization of hierarchical information", *Proc. InfoVis '99*, IEEE CS Press, 1999, pp. 73-78