Interactive Visual Mechanisms for Exploring Source Code Evolution

Alexandru Telea   Lucian Voinea

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science
Den Dolech 2, 5600 MB Eindhoven, the Netherlands
alext@win.tue.nl, l.voinea@tue.nl

Abstract

The Visual Code Navigator (VCN) is an ongoing effort to build a visual environment for interactive visualization of large source code bases. We present two techniques that extend the previous work done on the VCN. We propose an efficient and effective mechanism for specifying and visualizing queries on the source code. Next, we show a new project evolution view that offers global insight in change correlations that span several files, and thus lets users sport possible inconsistencies, problems, or undesired project structuring. We illustrate both mechanisms using a real-life C++ source code base.

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 [8].

In this paper, we present our ongoing effort to construct a Visual Code Navigator (VCN). VCN uses solely the code base, i.e. a set of source code files, as 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. VCN consists of several interrelated tools, or views. Every view focuses on a separate code aspect and uses potentially different visualization and interaction techniques to bring that aspect to the user. We present two extensions we designed and built in the VCN toolset. First, we describe a generic *query system* added to VCN's syntactic view. This view generalizes the 'syntax highlighting' provided by integrated development environments (IDEs) by drawing syntactic structures (e.g. for loops, classes, scopes) in custom ways [5]. Just as the syntactic view itself generalizes syntax highlighting present in IDEs, our query system generalizes the point-and-click code queries offered by the same IDEs. Users can program new queries as separate plug-ins and easily integrate them in VCN. Next, they can apply these queries on the code shown in the syntax view, by an easy point-and-click interface. In contrast to most IDEs, which open new (text-based) views to show query results, we display these results in-place in the syntactic view. This diminishes the cognitive disruption caused by view switching, and also allows query cascading, i.e. building complex queries from simple ones. Secondly, we add a new *project view* to the VCN. While the existing VCN file view shows the evolution of a single file in time [5], the project view displays the evolution of a whole project, seen as a set of files, in time. The file view focuses on low-level changes, at code line level. The project view covers higher-level changes, at file level.

In Section 2, we review related work on queryable, interactive source code visualization. Section 3 presents the VCN and its syntactic, file evolution, and project evolution views, and the query mechanism. Section 4 concludes the paper.

2. Related Work

Several tools address the challenge of source code visualization. SeeSoft [1], Augur [3], Aspect Browser [4], GSee [2], sv3D [6], and Almost [7] offer a line-oriented code visualization. Files are reduced to a 'zoomed out' image where the code layout is preserved but every textual code line becomes a pixel line, colored by code attributes and metrics, thus condensing tens of thousands of lines on one screen. This technique uses the assumption that developers are comfortable with viewing their code in the same spatial context in which it was constructed (written). A recent effort in this area is the Visual Code Navigator (VCN) [5]. VCN proposes an open architecture in which new (complementary) code visualization techniques can be added, to provide different insights in a code base. We extend VCN's capabilities with two new techniques: a project evolution view showing high-level changes during a project's lifetime, and a generic query architecture that allows users to construct complex queries and interactively apply them on the visualized code.

3. Code Visualization Views

We first outline the architecture of the VCN toolset (Figure 1). For detailed information, see [5].

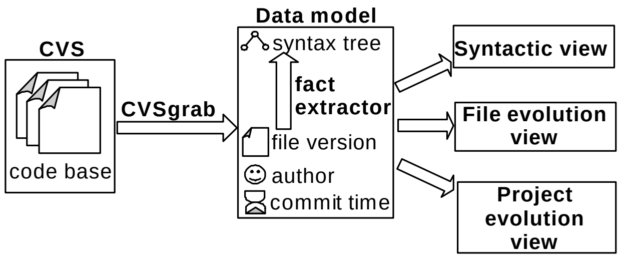


Figure 1: VCN system architecture

Our data source is the CVS version control management system. CVS holds several versions V_{ij} of several source files F_j, called a project. For every version, the commitment date (time when added to the repository), and author (who added it). To decouple CVS from VCN and allow for other data sources, extraction is done by a separate tool, called CVSgrab. The extracted versions V_{ij} of all files F_j are passed to a syntax fact extractor built by us. This is a modified GNU C/C++ compiler, with no code generation, which extracts all syntactic facts from the source code, e.g. classes, data members, function signatures, macros, templates, for-loops, etc, but also complete lexical information, such as the line and column positions where every code construct starts and ends. We obtain an 'enhanced' syntax tree from which we can render the source code at character level. This is essential for building the syntactic view and query mechanism described later in this section.

This data model, essentially a hierarchy ranging from high-level file-in-project data to low-level lexical details, is visualized by several views. The *syntactic* view shows all syntax constructs in a file (Section 3.1). We extend this view with a generic way to interactively define complex queries on the code (Section 3.2). The *file evolution* view shows the evolution of a single file [5]. We extend this by a *project evolution* view that shows the evolution of a whole project (Sec. 3.3).

3.1. Syntactic view

The *syntactic* view is essentially a classical text editor with three main changes. First, for every syntactic construct extracted by the parser, we render a *shaded cushion* whose outline matches the construct's text extent. Shaded cushions were introduced first by Van Wijk *et al.* to enhance treemap visualization techniques [11] by adding a block-nesting visual cue via the shading effect. Figure 2 shows the idea: given the `if` block (a), we render a shaded cushion to show its extent (b). In detail, we draw a polygon shaped as the stippled outline (a) textured with the cushion image (b). Merging cushions and text yields a 'generalized' syntax highlighting (c). Figure **4** shows our method for a code fragment of nesting depth 5. The eye perceives the cushions' height to be proportional with the nesting depth.

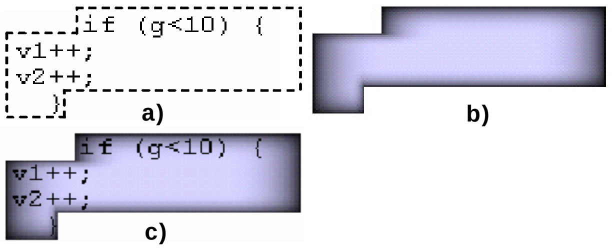


Figure 2: Code cushion design

Cushions allow drawing tree-like hierarchies of thousands of elements on a single screen, as shown in various applications, e.g. [11]. This allows us to visualize large, real-world code bases. Cushions combine best with 2D spatial layouts. This serves us well, as our syntactic view uses a 2D line-based layout: the *x*-axis maps the files visualized together, and the *y*-axis maps the lines in a file (Figure 3). Cushions are efficiently rendered with hardware-accelerated OpenGL, allowing interactive zoom and pan in the views, an essential usability aspect.

Secondly, we allow programmers to smoothly navigate between the familiar, trusted text editor view and the syntactic cushion view. We do this by blending the text over the cushions (Figure 2c). Sliders allow tuning both text (α_t) and cushion (α_c) transparencies to instantly change the visual focus from text (Figure 4 left, α_t=1, α_c=0.2) to syntax. (Figure 4 right, α_t=0.3, α_c=0.6).

Third, we color cushions to show the type of syntax construct they display. Users can browse all C/C++ constructs in a tree widget and change their color and visibility (Figure 3). In Figure 4 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. Turning off visibility for finer-grained constructs (e.g. identifiers) avoids visual cluttering and focuses on larger extent constructs, such as scopes, which help us grasp overall program structure.

By choosing from predefined color schemes, we can 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?"

Fourth, by changing the font height, and thus the text block and cushion sizes, we tune the amount of code shown on a single screen. For one-pixel tall fonts, the syntactic view becomes very similar to line-based visualizations [5]. Figure 3 shows this for 11 files. The largest is a C++ implementation file of 635 lines, the other 10 ones are header files. In total, this view shows over 3400 code lines. Although the actual text is not visible due to the small font size, the cushion view shows the

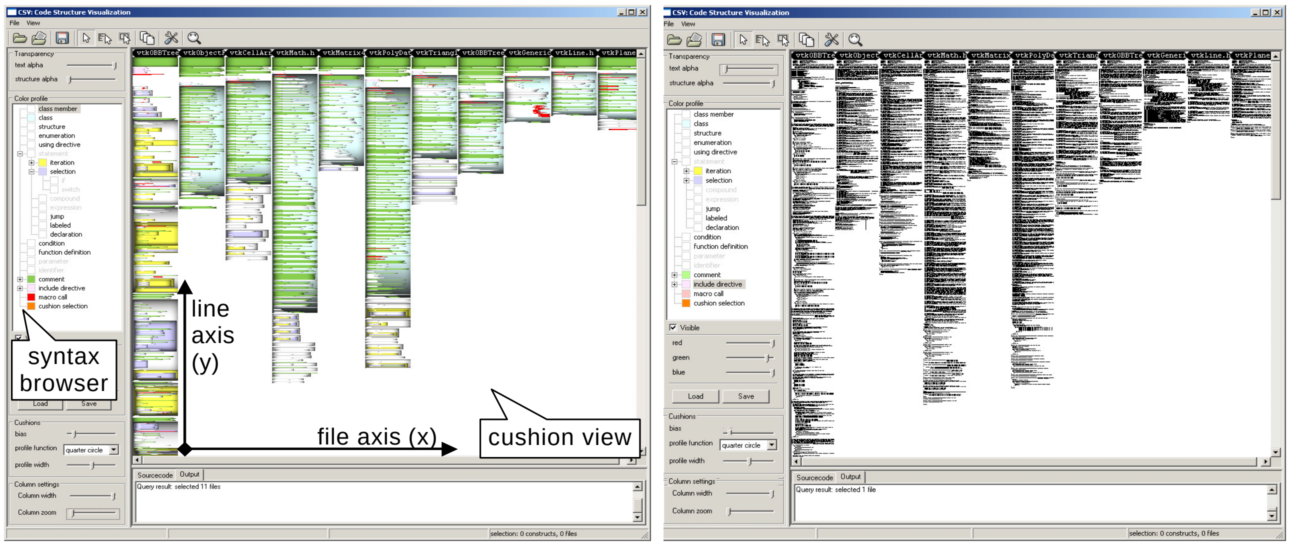
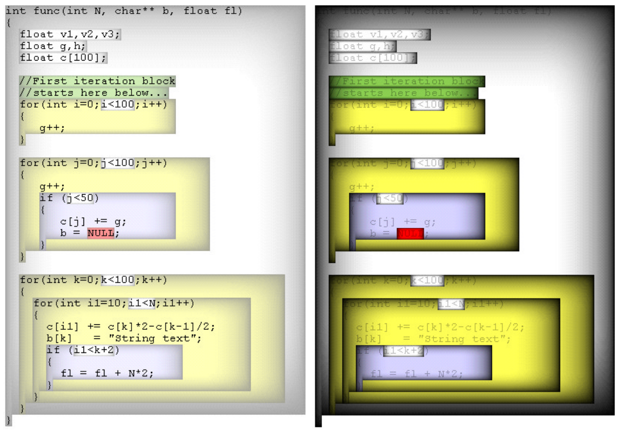


Figure 3: Syntactic view of 11 files with cushions (left) and without (right)

code structure, which is not visible in the classical text view.



```
int func(int N, char** b, float fl)
{
  float v1,v2,v3;
  float g,h;
  float c[100];

  //First iteration block
  //starts here below...
  for(int i=0;i<100;i++)
  {
     g++;
  }

  for(int j=0;j<100;j++)
  {
     g++;
     if (j<50)
     {
        c[j] += g;
        b = NULL;
     }
  }

  for(int k=0;k<100;k++)
  {
    for(int i1=10;i1<N;i1++)
    {
      c[i1] += c[k]*2-c[k-1]/2;
      b[k]   = "String text";
      if (i1<k+2)
      {
        fl = fl + N*2;
      }
    }
  }
}
```

Figure 4: Cushion and text blending

3.2. A generic visual query mechanism

The syntactic view presented above allows interactive display and navigation in tens of files of hundreds of lines each on a single screen. Code overviews can be done by zooming out (decreasing font size), fading out (decreasing text opacity), and making desired syntactic elements invisible. When working on code details, programmers often think in terms of: "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 the implementation stuff below those class declarations". The cushioned code view serves precisely these requests, as one quickly sees the source code size, nesting, type, and structure. However, we often require more complex queries. IDEs such as Eclipse or Visual Studio offer predefined queries, e.g. "show where a symbol is defined or used" or "show type of an object". Three problems exist with the implementation of such queries. First, query output is often displayed in a different way, or place, than query input. For example, the user clicks on a function in the editor and gets its call locations listed in a *separate* window, from where he can go, one by one, to *other* windows showing the actual call locations. This causes users to get confused and lose orientation in code space. Second, different queries have different ways to specify their input and display their output, causing more confusion. Finally, it is hard to cascade simple queries into more complex ones, or to add one's own queries.

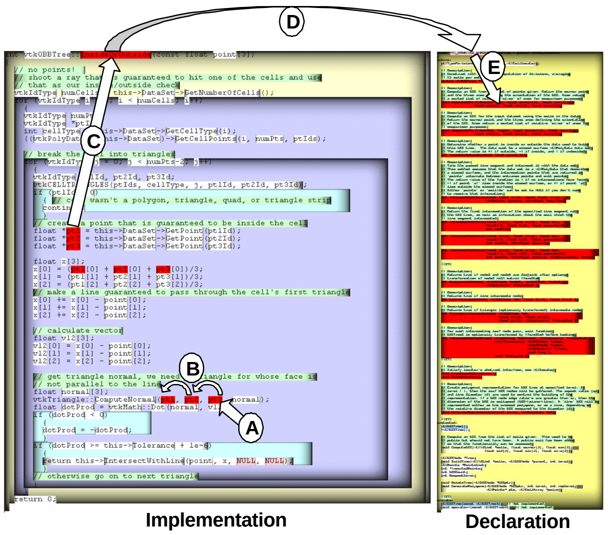
We address the above by a generic mechanism consisting of a query model, visual representation, and implementation model. We model a query as a function $q : S^I$? S^O, where $S^I = \{ e_i^I \}$, $S^O = \{ e_i^O \}$ are selections, i.e. sets of syntactic elements e_i from the syntax tree. A query maps from a set of syntactic elements e_i^I (the query input) to a similar set e_i^O (the query output). We call $T(S) = \{ t_i \}$ the signature of a selection set $S = \{ e_i^I \}$, where t_i^I is the type of e_i, e.g., class, method, function, variable, etc. We use this model as follows:

1. The user builds a *working selection* S by shift-clicking the syntactic elements in the cushion view he wants to query. S is displayed in a special selection color (e.g. red).
2. Right-clicking on any element of S shows a menu with all queries that can be applied on S, i.e. all queries whose input signature $T(S^I)$ matches $T(S)$. Two signatures $T(U) = \{ u_i \}$ and $T(V) = \{ v_i \}$ match if types u_i , v_i match for all i. Type

matching follows the standard C/C++ language rules [9].

3. The user selects a query *q* from the menu and applies it on the selection *S*.
4. The output of *q* replaces the working selection. It becomes immediately visible, since drawn in the selection color.

This method has several advantages. Queries are applied by a few mouse clicks. Both query input and output are displayed in the same way and in the same view, directly on the code. Queries are context-sensitive by default. The context is the working selection built by the user. For example, if one selects classes, then the query menu shows only queries that can be applied on classes. This considerably simplifies the use of the queries, since the user sees only the 'valid' options from the possibly large total query set. Cascading simple queries to perform more complex ones is also trivial. Since the query output replaces the query input (step 4), we can simply keep right clicking on the working selection to query it. Formally, this corresponds to a composition of the query functions *q*. Finally, adding custom queries is easy. These are C++ functions with a fixed signature $q : S^I ? \quad S^O$. One has only to fill in the body of *q* to describe how to build S^O from S^I.



```
int vtkOBBTree::InsideOrOutside(const float point[3])
{
  // no points!
  // shoot a ray tha[illegible]s guaranteed to hit one of the cells and use
  // that as our ins[illegible]/outside check
  vtkIdType numCells[illegible] this->DataSet->GetNumberOfCells();
  for (vtkIdType i [illegible]; i < numCells; i++)
    {
    vtkIdType numP[illegible]
    vtkIdType *ptI[illegible]
    int cellType [illegible]is->DataSet->GetCellType(i);
    ((vtkPolyDat[illegible]his->DataSet)->GetCellPoints(i, numPts, ptIds);

    // break the [illegible] into triangles
    for (vtkIdTy[illegible] j = 0; j < numPts-2; j++)
      {
      vtkIdType [illegible]1Id, pt2Id, pt3Id;
      vtkCELLTR[illegible]GLES(ptIds, cellType, j, pt1Id, pt2Id, pt3Id);
      if (pt1Id[illegible] 0)
        { // ce[illegible] wasn't a polygon, triangle, quad, or triangle strip
        contin[illegible]
        }
      // crea[illegible] a point that is guaranteed to be inside the cell
      float *pt1 = this->DataSet->GetPoint(pt1Id);
      float *pt2 = this->DataSet->GetPoint(pt2Id);
      float *pt3 = this->DataSet->GetPoint(pt3Id);

      float x[3];
      x[0] = (pt1[0] + pt2[0] + pt3[0])/3;
      x[1] = (pt1[1] + pt2[1] + pt3[1])/3;
      x[2] = (pt1[2] + pt2[2] + pt3[2])/3;
      // make a line guaranteed to pass through the cell's first triangle
      x[0] += x[0] - point[0];
      x[1] += x[1] - point[1];
      x[2] += x[2] - point[2];

      // calculate vector
      float v12[3];
      v12[0] = x[0] - point[0];
      v12[1] = x[1] - point[1];
      v12[2] = x[2] - point[2];

      // get triangle normal, we need [illegible]riangle for whose face is
      // not parallel to the line
      float normal[3];
      vtkTriangle::ComputeNormal(pt1, pt2, pt3, normal);
      float dotProd = vtkMath::Dot(normal, v12[illegible]
      if (dotProd < 0)
        {
        dotProd = -dotProd;
        }
      if (dotProd >= this->Tolerance + 1e-6)
        {
        return this->IntersectWithLine(point, x, NULL, NULL);
        }
      // otherwise go on to next triangle
      }
    }
  return 0;
```

Figure 5: Visual cascading of queries

Figure 5 illustrates the query mechanism. First, the user clicks on a local variable in a method in an implementation file (A). Next, he applies the query "show all instances of this variable" to get all occurrences thereof in the method's body, 9 in total (B). Next, applying the query "show enclosing scope" on one of these instances, leads us to the method name (C). The query "show class" on the method name leads us to the class owning the method, in a header file (E). Finally, the query "show all public methods" displays the public interface of this class, shown as the red (dark) selected area in the right file in Figure 5.

3.3. File and project evolution views

The second code view in the VCN is the *file evolution* view. This shows the evolution, or change, in a file's source during a project's lifetime. It uses a 2D layout similar to the syntactic view (Section 3.1): the *x*-axis maps the file version number and the *y*-axis maps the line number.

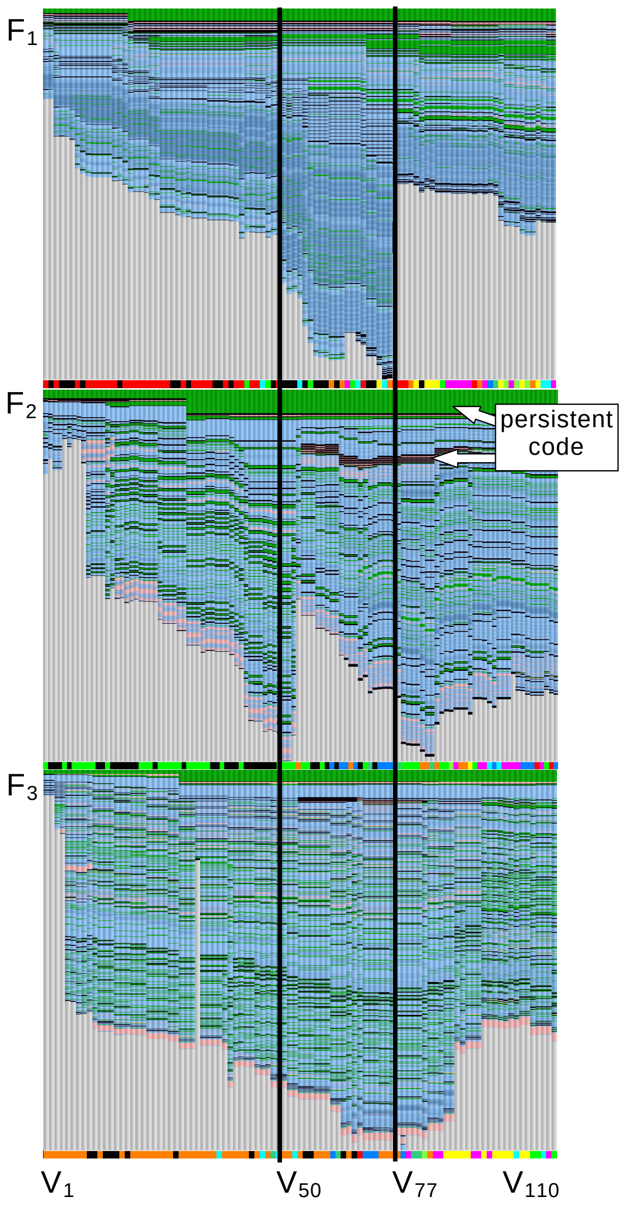


Figure 6: Evolution view

The file 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 mapping a source code line in V_{ij}, colored by line type or line author, as detailed in [5]. The idea is to reduce file versions to (thin) pixel stripes. Drawing these stripes along each other allows correlating code changes across several versions. Drawing several file evolution views atop of each other allows correlating changes across several files. Figure 6 shows the evolution of three files F_1, F_2, and F_3 across 110 versions. The largest version (V_{50}) has 650 lines. Code is 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).

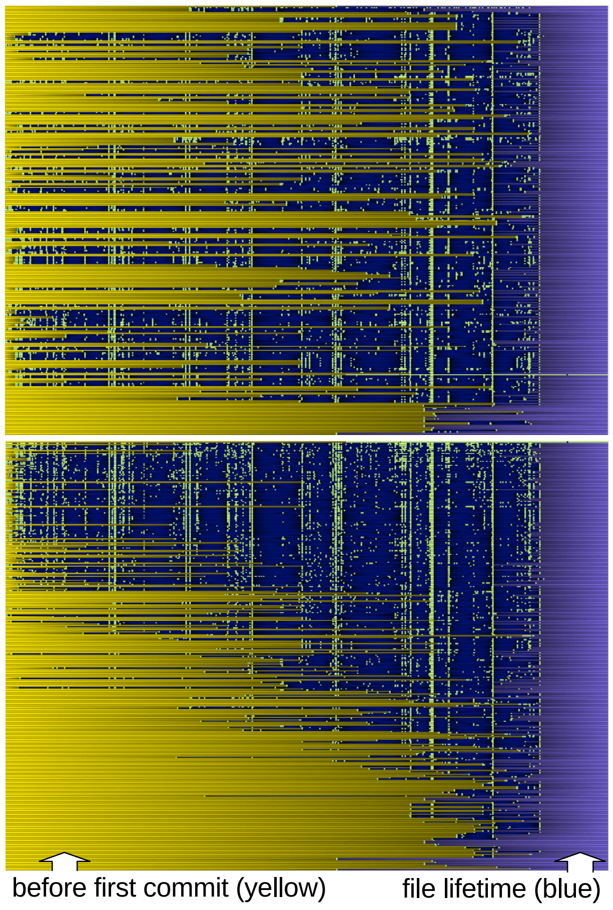


Figure 7: Project evolution view

The evolution view shows several aspects of 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, e.g. the code insertions around V_{50} for F_1 and F_2 (top two images) and at V_{77} (code deletion in F_1 versus code insertion in F_2). Sharing the time axis allows correlating changes in different files that happen at or around the same version, such as at V_{77} for F_1 and F_2. This signals an important change that affected more than one file. Code persistence over several versions can be detected by looking for horizontal color bands spanning several versions (Figure 6, middle image). The band waviness shows code insertion or deletion.

We present now a new *project evolution* view that works at a higher, more abstract, level. In this view, different files are mapped to the *y*-axis and time (file versions) is mapped to the *x*-axis (Figure 7). A file is thus mapped to a thin horizontal pixel stripe.

Color shows commitment events. Files are yellow (bright) before their first commit, and blue (dark) after. White (brightest) shows the commit moments. In the upper image in Figure 7, files are sorted in alphabetical order. In the lower image, they are sorted on decreasing order of the number of commit events. This clusters the most frequently changed files at the top of the image. Finally, we notice that the commit events (bright dots) group themselves in several vertical bright dotted stripes. These show commit events corresponding to *several* files, which help us detecting those changes that spanned (influenced) more than one file. To get insight into the exact changes that took place at this time, the user can select the affected file versions at that time and open them in the syntactic view, for the finest level of detail.

Next, we notice a blue (dark) vertical stripe-like region at the right side of the visualization, where there are no commits (bright dots), which spans about 15% of the project time axis. This final phase where no commits are done is typical for closed, matured software projects.

We used VCN in a study to understand VTK, a C++ library of hundreds of classes in over 2000 files, spanning over 100 versions, developed by tens of programmers over 10 years [10]. Our three users, who were experienced with C++ but never used VTK, acquired 100 versions of a few VTK files with CVSgrab from the public VTK web repository, analyzed them in VCN, and addressed several questions. A fourth user, with over seven years of VTK experience, specified the files to analyze, the questions, and assessed the answers delivered after two hours of investigation. The limited space here precludes us from detailing all results of this study. However, we must say that all users had very similar conclusions, validated as correct by the experienced user. Also, VCN was found to be very helpful in getting quick insight in a large, unknown code base. A VCN prototype and example datasets are available at:

http://www.win.tue.nl/ ~lvoinea/VCN.html

4. Conclusions and future work

We presented two extensions to the Visual Code Navigator (VCN), as previously introduced in [5]: a project evolution view and a generalized query mechanism. The project view displays file-level changes during a project evolution, i.e. file entry, exit, and commit times in a project lifetime. The query mechanism allows constructing complex queries on the syntactic code view. Cascading simpler queries to build more complex ones takes just a few mouse clicks. Custom queries can be built by implementing a function with a fixed, simple signature.

VCN supports, so far, only C/C++ code. Still, its techniques are applicable to any programming language for which syntax extractors (parsers) are available. An interesting challenge is to add these in the VCN architecture by some kind of generic plug-in mechanism. This is by no means an easy task. A major challenge here is to define a 'generic grammar' that is able to cover several programming languages and make them all 'available' to the visualization in an uniform manner. Secondly, we plan to extend VCN by enhancing the visual query mechanism beyond 'plain' query cascading and by building new source code views, e.g. to visualize syntax-based code changes. To do the latter, we must be able to compare source code fragments beyond the basic capabilities of the purely lexical `diff` operator. An option is to design a syntactic `diff` operator which is able to compare entire parse trees. Again, this is no easy task.

Concluding, we see a large number of challenges and prospects for extending the Visual Code Navigator with several techniques, in order to make it even more effective in helping developers getting insight in their source code and, hence, prove the value of visual tools in supporting software engineering.

References

[1] Eick, S. G., Steffen, J. L., Sumner, E. E., "SeeSoft: A Tool for Visualizing Line Oriented Software Statistics", *IEEE Trans. on Soft. Eng.*, 18(11), 1992, IEEE CS Press, pp. 957 – 968.

[2] Favre, J.M., "GSEE: A Generic Software Exploration Environment", *Proc. IWPC'01*, IEEE CS Press, 2001, pp. 233 – 244

[3] Froehlich, J., 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.

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

[5] Lommerse, G., Nossin, F., Voinea, S.L., Telea, A., "The Visual Code Navigator: An Interactive Toolset for Source Code Investigation", *accepted for IEEE InfoVis 2005*, IEEE CS Press. See also http://www.win.tue.nl/~lvoinea/vcn.pdf

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

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

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

[9] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Professional, 2000

[10] VTK Web Repository: http://www.kitware.com/

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