Code Canvas: Zooming towards Better Development Environments

Robert DeLine
Microsoft Research
One Microsoft Way
Redmond WA USA 98052
rob.deline@microsoft.com

Kael Rowan
Microsoft Research
One Microsoft Way
Redmond WA USA 98052
kael.rowan@microsoft.com

ABSTRACT

The user interfaces of today's development environments have a "bento box" design that partitions information into separate areas. This design makes it difficult to stay oriented in the open documents and to synthesize information shown in different areas. Code Canvas takes a new approach by providing an infinite zoomable surface for software development. A canvas both houses editable forms of all of a project's documents and allows multiple layers of visualization over those documents. By uniting the content of a project and information about it onto a single surface, Code Canvas is designed to leverage spatial memory to keep developers oriented and to make it easy to synthesize information.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments

General Terms

Design, Human Factors.

Keywords

Integrated development environments, software visualization, zoomable user interfaces.

1. INTRODUCTION

Integrated development environments (IDEs) were introduced thirty years ago, with the goal of increasing developer productivity by uniting then-separate development tools, like editors, compilers, debuggers, and analyzers, under a common user interface. This idea has been extremely successful, and many programmers today use IDEs, like Eclipse, Apple Xcode, and Microsoft Visual Studio. While many aspects of IDEs have improved over time, their user interfaces have remained largely the same. Today's IDEs have a "bento box" design: the screen is partitioned into rectangular areas that contain editors (e.g., code editors, user interface designers), navigators (e.g., project viewers, class viewers), and tool output (e.g., search results, compilation errors).

ICSE '10, May 2–8 2010, Cape Town, South Africa

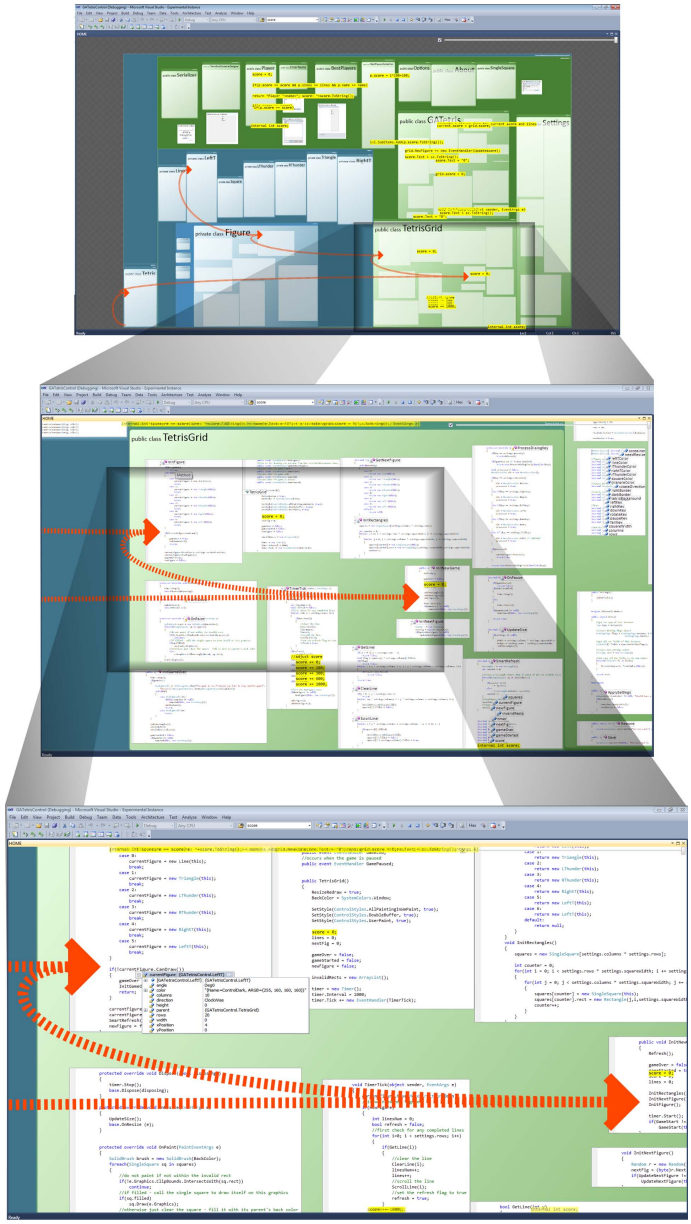

Figure 1. Code Canvas displaying the contents of a game project, at three levels of zoom, along with the debugger stack trace (curved dashed arrows) and search results (yellow boxes).

areas of the screen, the bento box design makes frequent use of symbol cross-referencing and hyperlinks. A method name, for example, could appear in class viewers, search results, and analysis results; clicking on the method name acts as a hyperlink to jump to the methods' definition.

While this bento box design has proved robust over the years, it nonetheless has several shortcomings.

- First, the frequent use of hyperlinks to jump around the project causes both disorientation and a frustrating number of open documents. A programmer can "get lost in the code," that is, struggle to find a given definition among the open documents. This disorientation can be frequent, given that programmers spend a large fraction of their time seeking answers to questions about their code. [13]
- Second, researchers are increasingly data mining a project's artifacts to provide more information about the project and its history (for example, the Mining Software Repositories conference). When the output of these analyses are shown in separate areas of the display, synthesizing the information is difficult. As one example, answering the question *Which methods on the call stack have changed recently?* would require mentally synthesizing information across a debugger window and a revision system window.
- Finally, many developer's desktop machines have both multiple monitors and high-performance graphics cards, with even more pixels and processing power likely in the future. The bento box design does not exploit these technical trends.

In this paper, we present a new style of user interface for IDEs, implemented as a prototype called Code Canvas. Code Canvas replaces all of the bento box's rectangular areas with a single zoomable surface, called a canvas, that houses all of a project's documents – source code, user interface designs, images, etc. A user zooms in to edit a document and zooms out to get an overview. To help keep the user oriented, Code Canvas serves as a map of the project, allowing the user to form and exploit spatial memory to find items. Code Canvas also serves as a visualization surface, displaying layers of information about the project, including search results, test coverage, and execution traces.

2. THE DESIGN OF CODE CANVAS

Figure 1 shows a small game, written in C#, at three levels of zoom in Code Canvas. Zooming in on the indicated portion of the top screenshot produces the middle screenshot; zooming in on the indicated portion of the middle screenshot produces the bottom screenshot. Code Canvas can display any document in any language that Visual Studio supports: C# code, XML data, user interface designs, images, etc. As an example of this heterogeneity, the top screenshot shows C# code side by side with user interfaces designs. The screenshots also show two active visualizations: the debugger call stack is shown as dash, curved arrows; and the results of searching for the term "new" are shown in yellow boxes.

2.1 Semantic Zoom and Navigation

Code Canvas uses a semantic zoom technique to show different levels of detail at different levels of zoom. When the user views a C# file at 100% zoom (bottom screenshot), she sees the typical code editor and can browse, edit and debug the code in the normal fashion. As she zooms out and the code becomes less readable, Code Canvas introduces a set of labels (middle screenshot) with the names of types and members, whose text is always kept at a readable size, regardless of zoom level. There is a pecking order on the labels. As the user zooms out and less screen space is available, lower priority labels (e.g. private methods, field names) are dropped to leave room for higher priority labels (e.g. public methods). At the outermost level of zoom, the canvas shows a diagram of the system's structure, both the directory and file structure and the class structure.

All navigation in Code Canvas is through pan and zoom of the whole canvas and all display transitions are animated, reinforcing a physical sense of space. Even when the user follows a hyperlink (for example with the go-to-definition command), Code Canvas pans and zooms the canvas to reach the hyperlink target. Code Canvas displays the full content of all documents on the canvas to avoid having two competing kinds of scrolling.

2.2 Code Layout

Code Canvas uses a mixed initiative strategy for layout. We use the MSAGL[1] graph layout engine to create an initial layout of the project documents. The user can then modify that layout in three ways. First, many of the items on the screen, including directories, files and editors, have drag handles by which the user can directly position the items. Code Canvas invokes the graph layout engine during these drag interactions to maintain containment and edge relationships in the diagram. Similarly, if a user adds text to a document and thereby increases its size, Code Canvas invokes the layout engine to push away the neighboring documents and to maintain containment and edge relationships.

The second way that the user can affect layout is to introduce new containers to represent concepts that are not syntactically explicit in the code, such as cross-cutting concerns. We added this feature based on previous research showing that developers represent such concepts when drawing diagrams of their code [3].

As the final way to affect layout, the user can "tear off" an individual method or a set of consecutive methods in a code file, which splits the file into fragments. For example, the bottom of Figure 1 shows the method InitFigure in its own fragment; its class TetrisGrid is drawn as a rounded green rectangle around the method fragments. Each fragment is simply a different editor view on the same underlying file. No changes to the compiler nor source revision system are needed for this feature.

The ability to lay out source code in units smaller than a file offers several advantages. First, when layout is based on whole files (as with Seesoft [9] or Code Thumbnails [6]), files appear as awkwardly long "filmstrips," which fit poorly on a screen with the opposite aspect ratio. Second, users can take advantage of the two-dimensional layout to express design intent, e.g. either placing methods idiosyncratically based on code content (e.g. all visitor methods side by side) or systematically (e.g. à la Class Blueprints [8]).

[1] research.microsoft.com/projects/msagl

2.3 Layers of Visualizations

In addition to the project's documents, Code Canvas also shows visualizations of information about the project. Code Canvas organizes the visualizations into layers, as are commonly found in tools for graphic designers, like Adobe Photoshop. The graphics drawn in a given layer all appear in the same plane, ordered along the Z axis. Code Canvas currently has the following layers, listed in Z order from back to front: directory structure; file structure (i.e. boundaries around code fragments); class diagrams; code test coverage; document editors; definition labels; code annotations (sticky notes); execution traces; search results; and reference edges (type/subtype and caller/callee relationships). Layers above the editor layer act as code overlays, while those below act as underlays.

The set of layers is extensible: new visualizations can be added, each in its own layer. The user can independently show or hide each layer, depending on the information needs of her task. The ability to show multiple layers at the same time makes it easier to synthesize information across multiple analyses. For example, Figure 1 shows both the current call stack in the debugger (the curved arrows) and search results (yellow boxes, showing term "new"), making it easy to find those methods on the call stack that perform allocations.

2.4 Multiple canvases

Code Canvas can create multiple canvases simultaneously, each one its own view onto the same underlying space. Code Canvas automatically creates the first canvas, called the "home" canvas. Additional canvases can be dragged onto other monitors (e.g. for comparative tasks) or can be docked with other canvases in a tabbed browser (clicking tabs switches between canvases). Each canvas has its own viewport, its own level of zoom, its own set of active layers, and its own filtered set of items shown (described below).

With multiple canvases, the user can perform detailed work simultaneously in two distant parts of the canvas, without the need to pan and zoom repeatedly back and forth. To do this, the user opens a second canvas and navigates to the distant location in the new canvas. The user can either place the two canvas on different monitors or dock them together and use the tabs to flip between them.

Another reason for multiple canvases is to support multitasking. When a user has a new task, she can create a new canvas for it, preventing the new task's navigations and visualizations from polluting the previous task state. (This is similar to how users of a tabbed web browser typically create new tabs for new information-seeking tasks.) To support interrupted and deferred tasks, Code Canvas persists a canvas' content in a relational database. Hence a user can return to a task's context, even if the task has not been active for weeks.

In addition to creating new canvases that are copies of the home canvas, Code Canvas also supports filtered canvases, which show a subset of the project's documents. The user can create a filtered canvas by multi-selecting items on the current canvas, then launching a new canvas, which will then contain only the selected items. Code Canvas uses the graph layout engine to gravitate the filtered items toward one another, compacting the area they cover while preserving their relative spatial positioning. The user can also create a filtered canvas based on a layer. When a new canvas is launched from a layer, it contains only those items that are involved in that layer. As an example, Figure 2 shows a canvas that was launched from the execution trace (call stack) layer in Figure 1. The filtered canvas contains only those fragments whose methods are part of the call stack. As another example, a user could launch a canvas from the search results layer to see only those fragments that contain the search term. Like any new canvas, a filtered canvas shows all the layers that were turned on in the canvas from which it was launched. Hence, Figure 2 shows search results as well as the stack trace.

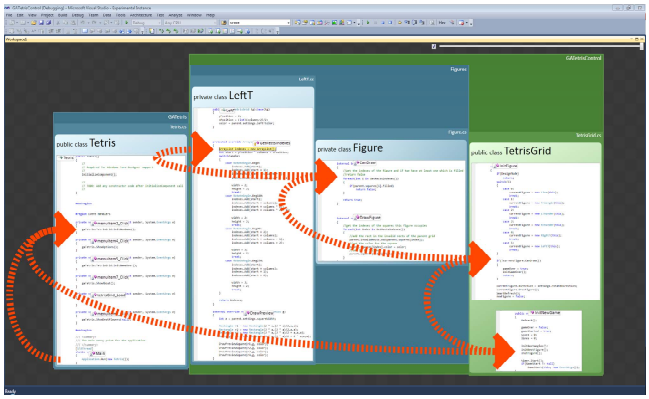

Figure 2. A filtered canvas showing only those fragments involved in the debugger call stack.

3. RELATED WORK

The Code Canvas project lies in the intersection of three research areas: software visualization, visual programming languages, and zoomable user interfaces. Each of these has a substantial history, which can only be briefly mentioned here. Code Canvas is mostly closely related to software structure visualizations intended to support program comprehension, like Shrimp/Creole [11]. Several of these previous visualizations have been based on thumbnail versions of the source code files, starting with Seesoft [9], and more recently Code Thumbnails [6] and Enhance [12]. These previous visualizations were intended as supplements to the development environment, either implemented as standalone tools or embedded as windows in the IDE. Unlike these previous tools, Code Canvas is designed to replace the IDE's user interface, rather than supplement it.

Visual programming languages (VPLs) provide both a programming notation and a two-dimensional spatial representation of programs. The programming notation might be object-based, as in Self [14] and Boxer [7], or functional, as in Prograph [4]. Like a VPL, Code Canvas also provides a two-dimensional spatial representation of programs and therefore uses similar representation conventions (e.g. containment for inclusion relationships, edges for pairwise relationships) and similar interaction techniques (direct manipulation, with a layout engine maintaining relationships). However, Code Canvas is not a programming notation and intentionally reuses the existing languages, compilers and debuggers implemented in the IDE.

Code Canvas is, to our knowledge, the first Zoomable User Interface (ZUI) designed as the front-end to an IDE. The first ZUI, Pad++ [1], allowed both infinite pan and infinite zoom. More recent examples of ZUIs, like Google Earth and Photosynth, restrict the levels of zoom to prevent disorientation, as does Code Canvas. Many ZUIs use space-distorting techniques, like fisheye views [10], to show details within context. The current design of

Code Canvas avoids these techniques to promote spatial stability to avoid interfering with the formation and recall of spatial memory. Code Canvas is the latest in a series of designs of code representations to exploit spatial memory, including Software Terrain Maps [5] and Code Thumbnails [6].

Code Canvas' design has many similarities to Code Bubbles, in these same proceedings [2]. The main difference is that Code Bubbles provides a spatial layout of the user's working context, which unfolds as the user explores, while Code Canvas provides a spatially stable overview of the entire project. Code Canvas' filtered canvases are an alternative approach to supporting working contexts.

4. OPEN RESEARCH QUESTIONS

Our next step will be to test the Code Canvas prototype with professional developers in the usability lab. There are several research questions we intend to evaluate:

- *Can two-dimensional layout capture design intent?* The ability to spatially arrange code fragments is a new type of secondary notation, akin to the use of whitespace and comments in existing textual notations (that is, the layout has meaning to programmers, but not to the compiler). What intentions will developers want to express with this secondary notation, and will others be able to read that intent?
- *How well does a spatial layout avoid disorientation and support multitasking?* Previous research showed that programmers quickly form a spatial memory of code files laid out on a two-dimensional surface [6]. In theory, this spatial memory should help prevent disorientation and allow better recall of interrupted or deferred tasks, but this has not yet been shown empirically.
- *To what extent should Code Canvas be a collaborative space versus a personal space?* Development is a collaborative effort, and shared diagrams are often a vehicle for spreading knowledge among team members. On the other hand, many individual development tasks would be better supported by a personalized view. In short, some aspects of Code Canvas should be common across team members, while other aspects should be individual – an area for future design exploration. Furthermore, a programmer's team mates create code churn that causes spatial instability. Another open design issue is to incorporate others' work into one's own Code Canvas with the least disturbance to spatial memory.

5. SUMMARY

Code Canvas demonstrates a new approach in the design of user interfaces for development environments. Rather than balkanizing information in disjoint display areas, Code Canvas provides a single, zoomable surface on which a programmer's work can be conducted and information needs can be met through visualizations. This design is intended to reduce disorientation, support an increasing number of analyses, and allow the programmer to benefit from modern displays and graphics processing.

6. REFERENCES

[1] B. Bederson and J.D. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *Proc. ACM Symp. on User Interface Software and Technology*, 1994.

[2] A. Bragdon, S.P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J.J. LaViola Jr. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proc. International Conference on Software Engineering*, 2010.

[3] M. Cherubini, G. Venolia, and R. DeLine. Building an ecologically-valid, large-scale diagram to help developers stay oriented in their code. In *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing*, 2007.

[4] P.T. Cox, F.R. Giles, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *Proc. IEEE Workshop on Visual Languages*, 1989.

[5] R. DeLine. Staying oriented with Software Terrain Maps. In *Proc. of the Workshop on Visual Languages and Computation*, 2005.

[6] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. 2006. Code Thumbnails: Using Spatial Memory to Navigate Source Code. In *Proc. IEEE Conf. on Visual Languages and Human-Centric Computing*, 2006.

[7] A.A. diSessa and H. Abelson. Boxer: A reconstructible computational medium. *Comm. of the ACM* 29(9): 859–868, Sept. 1986.

[8] S Ducasse, M Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Trans. on Software Engineering* 31(1):1–16, 2005.

[9] S.C. Eick, J.L. Steffen, E.E. Sumner Jr. Seesoft: A tool for visualizing link-oriented software statistics. *IEEE Trans. on Software Engineering*, Nov. 1992.

[10] G.W. Furnas. Generalized fisheye views. *ACM SIGCHI Bulletin* 17(4): 16–23, Apr. 1986.

[11] R. Lintern, J. Michaud, M.A. Storey, and X. Wu. Plugging-in visualization: Experiences integrating a visualization tool with Eclipse. In *Proc. ACM Symp. on Software Visualization*. 2003.

[12] H. Shah, C. Görg, M.J. Harrold. Visualization of exception handling constructs to support program understanding. In *Proc. ACM Symp. on Software Visualization*, 2008.

[13] J. Sillito, G.C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proc. ACM SIGSOFT Intl. Symp. On Foundations of Software Eng*. 2006.

[14] R.B. Smith, J. Maloney, D. Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness uniformity, and flexibility. ACM SIGPLAN Notices 30(10): 47–60, Oct. 1995