SkyscrapAR: An Augmented Reality Visualization for Software Evolution

Rodrigo Souza[1], Bruno Silva[1], Thiago Mendes[1,2], Manoel Mendonça[1]

[1]Computer Science Department, Federal University of Bahia (UFBA), Bahia, Brazil

[2]Information Technology Dept, Federal Institute of Bahia - Santo Amaro, Bahia, Brazil

{rodrigo, brunocs, thiagomendes, mgmendonca}@dcc.ufba.br

***Abstract**. Many software visualizations have been proposed to help stakeholders identify potential threats in the design of evolving software systems, often employing metaphors such as trees, graphs, constellations, maps, and cities, among others. In this scenario, three-dimensional visualizations have also been adopted. They can represent visual metaphors realistically and efficiently engage users. However, the interaction with such visualizations is either cumbersome or requires expensive equipment. This paper presents SkyscrapAR, an augmented reality visualization that employs the city metaphor to represent evolving software, along with its potential applications in the practice. The use of augmented reality allows the user to interact with the city in an intuitive way, by manipulating a marker, while requiring nothing but a computer and an inexpensive camera to make it work.*

1. Introduction

Software systems evolve by nature [Lehman 1980]. The activity of software development becomes more complex as the system evolves, which leads to the need for preventive and corrective maintenance. This complexity is reflected in the fact that about 90% of the total costs of software systems are associated with their maintenance. Therefore, the area represents a promising field for improvement [Erlikh 2000].

Visualization techniques have been used in software engineering as a viable solution to the difficult task of understanding, maintaining and evolving software systems. The visual metaphors are very useful in this context because sight is the most developed sense in human being [Diehl 2007].

During the last decade, tools such as SeeSoft and CodeCity have used 3D visualizations to represent, and hopefully provide a better understanding of software. 3D visualization is sometimes more intuitive and adds an extra dimension that allows the visualization of a larger amounts of information in relation to 2D visualization [Teyseyre 2009]. However, it also has an important limitation. The computer screen is bi-dimensional and always occludes something that is represented in three dimensions. This requires the user to continually interact with the visual scenarios to rotate the visual scene. Current interface devices, such the computer mouse, make it cumbersome for users to manipulate the six degrees of freedom (three translational axes and three rotational axes) of a 3D scene.

A solution to the problem of interaction between the user and 3D software is the use of augmented reality. This technology makes virtual objects visible in real environments, facilitating virtual interaction with the user, since the communication can be performed through gestures and movements [Azuma 1997].

This paper presents SkyscrapAR, a new software evolution visualization approach that uses a city graphical metaphor combined with augmented reality techniques. SkyscrapAR represents software classes and packages in a particular project revision as 3D buildings. These buildings are laid over on a terrain baseline blueprint derived from all versions of the software project. Buildings are sized up and colored by their size and number of changes. The shown project revision can be dynamically chosen and buildings can be dynamically filtered for better evolution analysis. The augmented reality resource allows for easy manipulation of the produced visual scenarios. To our best knowledge, there is no other work in the literature that uses this type of approach.

This paper is structured as follows. Section 2 introduces relevant related work. Section 3 presents SkyscrapAR and its features. Then, Section 4 discusses some applications of the tool. Finally, Section 5 concludes this article.

2. Related Work

According to Steinbrückner and Lewerentz (2010) and Teyseyre (2009), several tools have been proposed in the literature that use 3D visualizations to support software maintenance and evolution activities.

Without implementing it, Parnas (2003) proposed a 3D approach that depicts software production cost related program information to support software maintenance. The idea is to help maintainers and managers to decide which components in a system are superfluous, cause high cost due to frequent changes, and needs to be revised.

Wettel and Lanza (2008) introduced an approach called CodeCity, which can help to detect software design problems by presenting software in the form of a 3D city where the artifacts are mapped to the city's neighborhoods and buildings. Buildings represent classes sized accordingly to metrics such as number of methods and number of attributes. SkyscrapAR is inspired by CodeCity, but it uses different metrics, such as lines of code and code churn (the total number of lines changed in the evolving classes).

Maletic et al. (2001) proposed a virtual reality environment, called Imsovision, to investigate how immersive visualization environments can assist in software development and maintenance. Their environment requires CAVE, an expensive virtual reality system composed of projectors, special glasses, and an electromagnetic tracking system. Although SkyscrapAR does not create the same level of immersion, it provides an intuitive way to manipulate 3D models while requiring only an inexpensive camera.

3. SkyscrapAR

SkyscrapAR is an augmented reality (AR) software evolution visualization based on the metaphor of an evolving city, see Figure 1. Similarly to CodeCity, SkyscrapAR represents packages (or folders in the source code file structure) as rectangular city lots, with sub-packages being stacked on top of the package containing them. Classes (or source code implementation files) are represented by buildings (boxes with different areas and heights) located on top of their respective packages, with the area they occupy being proportional to the size of the class, measured in lines of code. The user can browse over revisions of the software, and see buildings appearing, disappearing, and changing their shape to reflect successive modifications that they suffered over time.

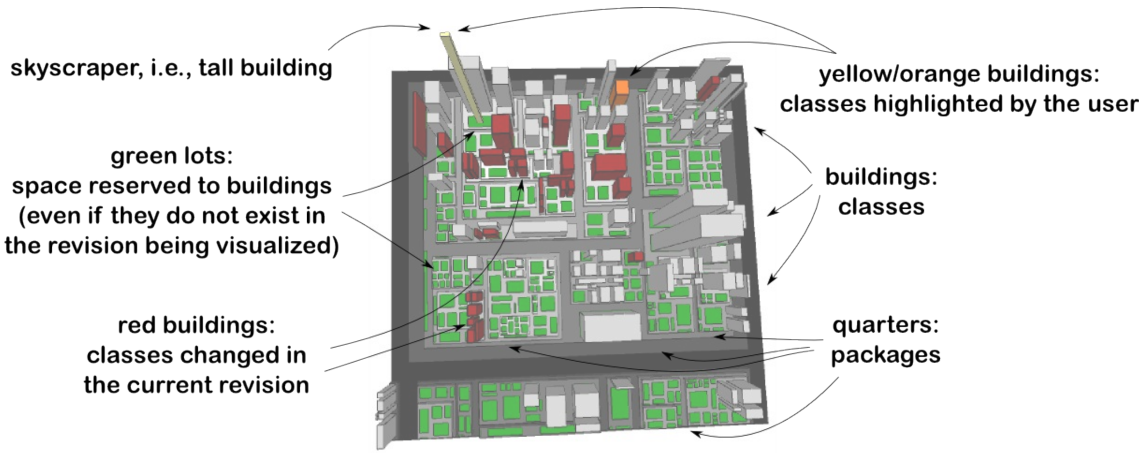
skyscraper, i.e., tall building
green lots:
space reserved to buildings
(even if they do not exist in
the revision being visualized)
red buildings:
classes changed in
the current revision
yellow/orange buildings:
classes highlighted by the user
buildings:
classes
quarters:
packages

Figure 1. The JUnit framework visualized as a city using SkyscrapAR.

Because the city (i.e., the software) evolves over time, it must have space to accommodate all buildings that existed in some period of its history. That is why the city terrain is divided in green lots, each one belonging to a building (i.e., a class). Each green lot is large enough to fit its respective building when it reaches maximum area. So, for example, buildings which present some green area in the last revision represent classes that used to be larger but had their size reduced.

The city visualization is projected onto the user's real-world environment, as captured by a camera. The user interacts with the visualization as he would do with a mock-up city, manipulating it with the hands so it can be seen from the desired viewpoint.

A demonstration video[1] of the tool can be found online. Readers are urged to look at it before reading further on. The full source code[2] is also publicly available under an open source license. The rest of this section describes SkyscrapAR in terms of its architecture, used source code metrics, and its user interface.

Architecture

SkyscrapAR is composed of two executables: the extractor and the viewer. The extractor, written in Java, reads a local Git repository containing Java source code and outputs a XML file describing its revisions, along with information for each source code file that was modified in each revision. The viewer, written in Processing, takes the XML file and presents its data as a city that the user can view and interact with.

The hardware setup needed to use the visualization consists of a computer with a camera and a printed marker. A marker is a piece of paper or any other object with a predefined black and white square pattern printed on it, as shown in Figure 2a. The marker signals where the city should appear within the user's environment.

Each frame captured by the camera is converted to a binary, black and white image, according to a configurable brightness threshold. After that, an algorithm detects black square contours and then compares its inner region with a predefined pattern. If they match, then it builds a local 3D coordinate system for the marker, as shown in Figure 2b. The 3D model of the city is then aligned with the local coordinate axes, as shown in Figure 2c, so it moves together with the marker, giving the effect that it is a part of the real world.

[1] Available at http://www.youtube.com/watch?v=VVRjihr-40U.

[2] Available at https://github.com/rodrigorgs/SkyscrapAR.

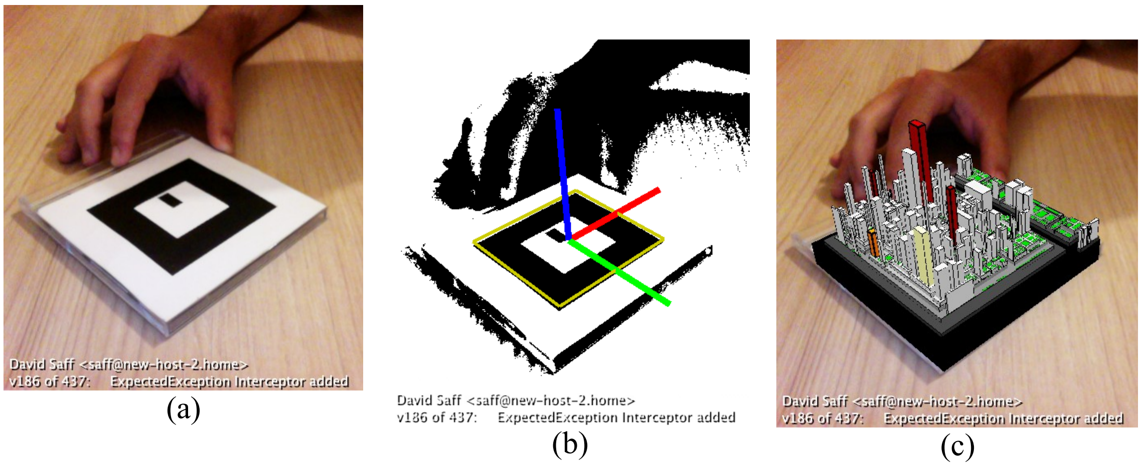
David Saff <saff@new-host-2.home>
v186 of 437: ExpectedException Interceptor added
(a)
David Saff <saff@new-host-2.home>
v186 of 437: ExpectedException Interceptor added
(b)
David Saff <saff@new-host-2.home>
v186 of 437: ExpectedException Interceptor added
(c)

Figure 2. Augmented reality: (a) the camera captures a scene containing a marker; (b) after the image is transformed to black and white, the marker is detected and its local coordinate system is computed; (c) the 3D model is aligned with the coordinate axes.

Mapping Metrics to Visual Elements

A software system is composed of consecutive revisions. In each revision, two metrics are computed for all the classes: number of lines of code (LOC) and code churn.

The LOC metric is the total number of lines in the respective source code file, including comments and blank lines. This metric, although simple to measure, correlates with many traditional complexity metrics that are used to predict development effort and fault proneness [El Elmam et al. 2001].

Code churn refers to how much a particular file has been modified in its current and previous revisions in the version control system. The code churn of a class in its initial revision is equal to its LOC. Then, each time the class is modified, the code churn is incremented by the number of lines of code added or removed in the revision, as computed by Unix's *diff* tool. That way, code churn always increases or is kept constant over time. The rationale for code churn is that source code files modified frequently are less stable and tend to be more prone to faults [Nagappan and Ball 2005].

The two metrics are mapped to visual attributes of the visualization. The LOC metric for a class determines the area of the base of the respective building, so that larger classes are represented by buildings that occupy a greater part of the city's territory.

All buildings start with a fixed, minimum height. In the subsequent revisions, the height of a building is determined by how much the code churn of its respective class increased since its initial revision. Therefore, classes that undergone many changes since their creation are represented by tall buildings, or *skyscrapers*.

It should be noted that code churn, unlike LOC, is a historical metric, i.e., its value for one class in one revision reflects the evolution of the class in the previous revisions. Its use in the visualization, therefore, enables the user to gather historical information about the software by looking at a single static view.

User Interface

The user interface for SkyscrapAR displays the scene being captured by the camera and, if a marker is on the scene, the 3D model of the city is superimposed on it. In the bottom part of the screen, information about the current revision being visualized is displayed. It includes the sequential number of the revision, together with the name of the developer responsible for the change and the message describing the change.

Contrary to what happens in most 3D software visualizations, in which users change their viewpoint by using the mouse or the keyboard, in SkyscrapAR, the user can view the city from different angles by manipulating either the 3D marker or the camera. With such interaction mechanism, the user can explore all six degrees of freedom of the 3D space (translation and rotation along three axes) in a natural way. For instance, the city can be seen from the top, in an aerial view, evidencing the structure of the software, or sideways, in a skyline view, evidencing skyscrapers, i.e., classes with high code churn.

In its current state, SkyscrapAR still depends on mouse and keyboard input, although we plan to reduce this dependency. The mouse pointer is used to highlight classes the user may want to focus on. The keyboard is used to navigate through revisions (left and right arrow keys), filter out buildings ("H" key), zoom in ("Z" key), and zoom out ("Shift-Z" key combination).

When the user advances the visualization to the next revision, buildings representing classes that were modified in that revision appear in red, so they can be easily spotted among the default gray buildings. It is expected that the buildings in red changed its shape compared to the previous revision[1] to reflect changes in the number of lines of code (area of the base) or in its code churn (height). Such changes in shape are presented as a smooth animation, by interpolating the dimensions of the buildings over a short period of time (less than one second). That way, it is easier to track the changes in the classes over time, enhancing the perception of evolution.

When the mouse pointer is over a building, the interface shows details on demand for the respective class: name, package containing it, and the values for the LOC and code churn metrics. The user can also highlight a set of buildings, which get painted in yellow, by clicking on them. If a class is highlighted and has changed in the current revision, it gets painted in orange instead (i.e., a red and yellow mix).

The purpose of the highlighting is twofold: it helps the user track the position of some buildings while exploring the visualization, and enables the user to focus on the changes of the highlighted buildings. When at least one building is selected, the navigation among revisions is restricted to revisions in which at least one highlighted class is changed. This is useful to study the evolution of a set of classes.

By pressing the "H" key on the keyboard, gray buildings are filtered out, i.e., only highlighted classes and classes that were changed in the current revision are displayed. This filter helps the user to analyze the scattering of changes throughout the packages. When combined with the restricted navigation that takes place when classes are highlighted, it also enables the user to visualize classes that evolve together within a given set of classes.

4. Applications in Software Development Practice

Normally, developers spend more time understanding the code than modifying it [LaToza 2006], partly because, as time goes by, software systems get harder to understand and to maintain unless effort is invested to simplify the design [Lehman 1980]. Code review meetings and module inspections are examples of team effort for identifying design flaws that could be mitigated to improve the software comprehension and maintenance.

[1] One notable exception is the case when only the file permissions or ownership were changed, although its contents were kept unchanged.

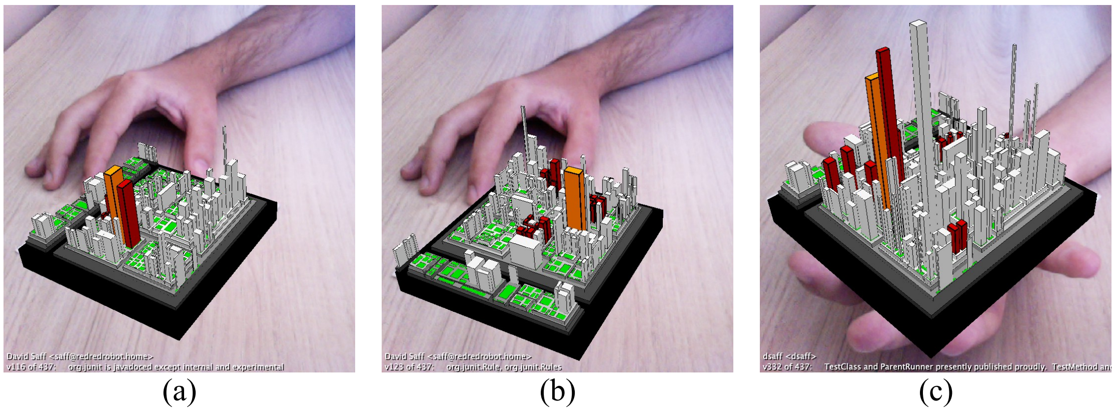
David Saff <saff@redredrobot.home>
v116 of 437: org.junit is javadoced except internal and experimental
(a)
David Saff <saff@redredrobot.home>
v123 of 437: org.junit.Rule, org.junit.Rules
(b)
dsaff <dsaff>
v332 of 437: TestClass and ParentRunner presently published proudly. TestMethod an
(c)

Figure 3. Three distinct versions of the JUnit framework in which the class BlockJUnit4ClassRunner (highlighted in orange) was modified.

Moreover, researchers have found demand for *agile assessments*, that is, a kind of software product assessment that may provide valuable information rapidly and cheaply to support program understanding and maintenance [Nierstrasz 2012]. We envision several applications of SkycrapAR in this context. This section describes some of them and shows screenshots of the tool to illustrate typical situations (see Figure 3).

Finding Skyscrapers

In a software project, it is common to find classes which change a lot—the so called change-prone classes. In SkyscrapAR, they can be easily found as *skyscrapers* (see the orange building in Figure 3). Analog to the real concept, skyscrapers in our visualization are classes that concentrate a lot of the development effort. These classes deserve special attention, since they are more likely to be fault-prone [Nagappan and Ball 2005].

Visualizing Scattering of Changes

Source code management systems usually list packages and classes that were changed in each revision, but there is no visual information about the scattering of each commit transaction that generates a revision. This is easily done in SkycrapAR (see red buildings in Figure 3). Identifying scattered changes is important, as they represent modifications that impacted several systems modules, pointing out system-wide maintenance activities, such as refactorings or move operations among folders, or indicating that there are too many dependencies between software modules. By focusing on this second issue, developers can detect candidates for the Shotgun Surgery, identifying design flaws that cause modifications in one module and trigger modifications in several other modules.

Visualizing Classes Co-change

Sometimes, classes have logical couplings even if they are not explicitly coupled to each other by their internal structural elements [Gall 1998]. Classes may be connected by means of implicit abstract elements related to the domain. For example, classes that are not structurally coupled but implement concerns that are strongly dependent at the requirements level. SkycrapAR provides information about the classes co-change by highlighting in red the classes that changed together in each revision (see Figure 3).

Finding God Classes

A god class is a well-known code smell which refers to classes that perform too much work on its own. It breaks one of the basic principles of object-oriented design which states that a class should have one single responsibility. Also, god classes are more likely to undergo changes as much as the evolution history targets the responsibilities implemented by the class [Silva 2012]. SkyscrapAR helps developers reason about god classes using the *High Occupation of Territory* metaphor—tall buildings occupying large areas. An analogy can be made to a real city. While, in real world, buildings that occupy a larger area require their dwellers to pay higher taxes, in the visualization it is the project developers who have to pay the price of maintaining large classes.

Identifying Populous Districts and City Centers

Normally *systems as cities* have one or more neighborhoods with tall buildings and packages with no or very few green area—those areas are *Populous Districts*. This observation can be interpreted as the most touched parts of the system. If there is only a single well-defined populous district, we call it the *City Center*. Finding the city center (or other populous districts) may be useful to focus code inspection and testing on the most modified and possibly the most fault-prone parts of the system. It may also help to find refactoring opportunities on specific packages that suffered too many changes.

Identifying Rural Houses

Rural houses are classes that lost their functionality along with the software evolution, so they are now presented as large green areas with a small building in the last revision of the software. Classes like that should be examined for signs of dead code, i.e. they may be obsolete and therefore be removed.

5. Final Remarks

This paper presented SkyscrapAR, an augmented reality visualization that employs the city metaphor to display information about the evolution of a software system. To the best of our knowledge, this is the first software visualization tool to employ augmented reality. In addition to describing SkyscrapAR, the paper shows how it can be used to support software comprehension tasks.

Currently, the tool has some limitations. In its current stage, only Java source code is supported, although it should be easy to add support for other languages. Also, it is limited to display one software system at a time, even if multiple markers are used. Finally, the user interaction still relies on the use of mouse and keyboard, which reduces the sense of immersion that characterizes augmented reality applications.

As future work, we intend to investigate interaction modes that dispense the use of mouse and keyboard. Our goal is to create a user interface that is intuitive, while discarding approaches that require expensive equipment such as virtual reality helmets and gloves. We also plan to make better use of colors in the visualization in order to display richer information. For instance, buildings can be painted in different colors to represent distinct crosscutting concerns implemented by them or, still, the developers who have contributed to the class. We plan to evaluate the tool with developers in academic or industrial settings. The evaluation should focus on usability and the effectiveness of the tool when used to support specific software comprehension tasks.

Acknowledgements

The authors wish to thank Michele Lanza for encouraging this publication.

This work was partially supported by CNPq/INES under grant 573964/2008-4.

References

Azuma, R. (1997) “A Survey of Augmented Reality”. Presence: Teleoperators and Virtual Environments, v. 6, n. 4, August, p. 355–385.

Diehl, S. (2007) “Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software”, Springer-Verlag New York, Inc.

El Emam, K., Benlarbi, S., Goel N., and Rai S. N. (2001) “The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics”. IEEE Transasctions on Software Engineering, 27(7).

Erlikh, L.(2000) “Leveraging legacy system dollars for e-business”. IEEE IT Pro, pp. 17–23.

Gall, H., Hajek, K. and Jazayeri, M.. (1998) “Detection of Logical Coupling Based on Product Release History”. In Proceedings of the International Conference on Software Maintenance (ICSM '98). IEEE Computer Society, Washington, DC, USA.

Lanza, M., Marinescu, R, and Ducasse, S. (2005) “Object-Oriented Metrics in Practice”. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

LaToza, T., Venolia, G. and DeLine, R.. (2006) “Maintaining mental models: a study of developer work habits". In ICSE '06: Proceedings of the 28th international conference on Software engineering, pages 492–501, New York, NY, USA.

Lehman, M. M., (1980) "Program Life Cycles and Laws of Software Evolution," Proceedings of IEEE, Special Issue on Software Engineering, September, pp. 1060–1076.

Maletic J. I., Marcus A., Dunlap G., and Leigh J., (2001) “Visualizing Object-Oriented Software in Virtual Reality,” Proc. Ninth Int’l Workshop Program Comprehension (IWPC ’01), p. 26.

Nagappan, N., and Ball, T., (2005) "Use of Relative Code Churn Measures to Predict System Defect Density", Proc. the 27th International Conference on Software Engineering (ICSE '05), St. Louis, MO, USA, May 15-21, 2005, pp. 284–292.

Nierstrasz, O. (2012) “Agile software assessment with Moose”. SIGSOFT Softw. Eng. Notes 37, 3 (May 2012), 1-5.

Panas, T., Berrigan, R. and Grundy, J. (2003) “A 3D metaphor for software production visualization”. International Conference on Information Visualization, page 314.

Silva, B., Sant’Anna C., Chavez, C. and Garcia, A. (2012) “Concern-based Cohesion: Unveiling a Hidden Dimension of Cohesion Measurement”. In: 20th IEEE International Conference on Program Comprehension, ICPC’12, Passau, Germany.

Steinbrückner, Frank, Lewerentz, Claus (2010) “Representing development history in software cities”. SoftVis '10, Proceedings of the 5th international symposium on software visualization. ACM, NY, USA.

Teyseyre, A. R., Campo, M. R., (2009) "An Overview of 3D Software Visualization", IEEE Transactions on Visualization and Computer Graphics, pp. 87–105.