Interactive Software Maps for Web-Based Source Code Analysis

Daniel Limberger [a] Benjamin Wasty [b] Jonas Trümper [a] Jürgen Döllner [a]

[a] Hasso-Plattner-Institut, University of Potsdam, Germany; e-mail: {daniel.limberger, jonas.truemper, juergen.doellner}@hpi.uni-potsdam.de
[b] Software Diagnostics GmbH, Potsdam, Germany; e-mail: benjamin.wasty@gmail.com

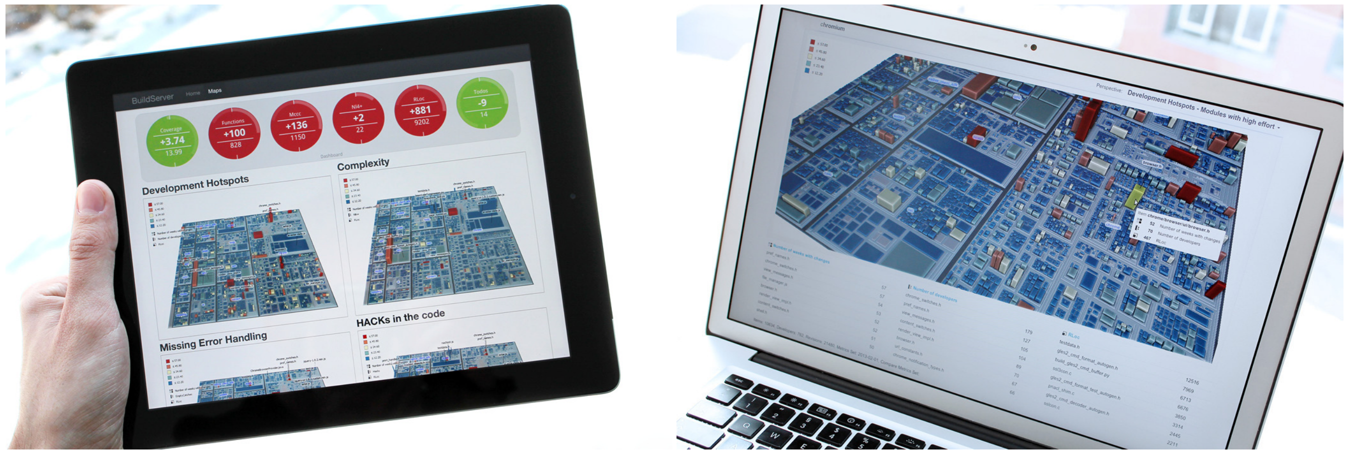


Figure 1: *Left: Dashboard accessed from a tablet computer, which summarizes project specific performance indicators (metrics) and links them to software maps. Right: Software map with code metrics mapped to ground area, height, and color, rendered in real-time in the browser.*

Abstract

Software maps – linking rectangular 3D-Treemaps, software system structure, and performance indicators – are commonly used to support informed decision making in software-engineering processes. A key aspect for this decision making is that software maps provide the structural context required for correct interpretation of these performance indicators. In parallel, source code repositories and collaboration platforms are an integral part of today's software-engineering tool set, but cannot properly incorporate software maps since implementations are only available as stand-alone applications. Hence, software maps are 'disconnected' from the main body of this tool set, rendering their use and provisioning overly complicated, which is one of the main reasons against regular use. We thus present a web-based rendering system for software maps that achieves both fast client-side page load time and interactive frame rates even with large software maps. We significantly reduce page load time by efficiently encoding hierarchy and geometry data for the net transport. Apart from that, appropriate interaction, layouting, and labeling techniques as well as common image enhancements aid evaluation of project-related quality aspects. Metrics provisioning can further be implemented by predefined attribute mappings to simplify communication of project specific quality aspects. The system is integrated into dashboards to demonstrate how our web-based approach makes software maps more accessible to many different stakeholders in software-engineering projects.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality D.2.2 [Software Engineering]: Design Tools and Techniques—Computer-aided software engineering; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Documentation;

Keywords: 3D-Treemaps, Software Visualization, Computer Aided Analysis, Decision Making, Web Integration, WebGL

1 Introduction

For a sustainable software engineering process in large software projects, a number of prerequisites exist. These include good resource management, collaboration, code management, prioritization, reviews, and decision making as well as architecture- and feature planning. Up-to-date knowledge of the software project, its processes, and, in particular, its source code are further key elements [Eisenbarth et al. 2003]. It is imperative to use additional services and platforms to support and ease this collaborative process [Trümper and Döllner 2012]. In fact, most available collaboration platforms provide visualization for various aspects related to development, e.g., branching, summary of development activities, status related charts, source code diffs and many more.

With increasing projects size, however, management overhead increases and an additional issue arises: Companies struggle with essential information gaps between management and development, which hinder informed decision making [Poole 2003]. Such gaps include clearly communicating quality standards (management to development), and, vice versa, reasoning why a specific amount of resources is required for a given development task. Visualizing these facts using traditional plain lists and charts is hard because of the exploratory nature of the underlying questions and problems: It enforces an iterative refinement of and re-focusing within a massive amount of facts. Fortunately, powerful various visualization

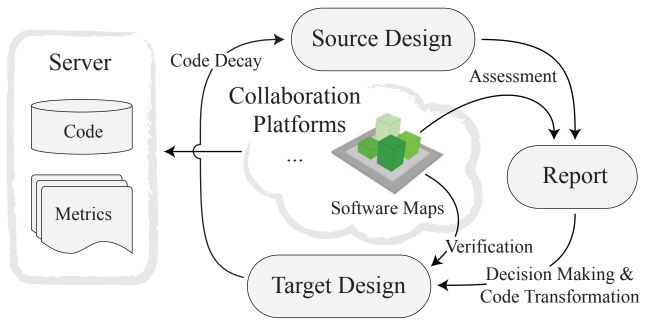


Figure 2: *Exemplary development cycle within larger software engineering processes. Software maps as integrated part of collaboration platforms, which the development process relies on, aid assessment and verification.*

techniques are available to help bridge this gap and to provide additional up-to-date information, which jointly aid exploring code quality, estimating benefits, and decisions making. Among others, code cities [Wettel and Lanza 2007] or software maps [Bohnet and Döllner 2011] are frequently used in this context. We here focus on software maps, which are 3D-Treemaps that provide a hierarchy-preserving layout and spatial context for code artifacts (*code units*) and metrics. Each treemap item represents a code unit (at varying abstraction levels) with at least three visual variables (height, ground area, color) that metrics can be mapped to.

While stand-alone applications for software maps, which allow arbitrary metric mappings, exist, their daily use in software engineering projects is significantly limited by several circumstances: First and foremost, being stand-alone applications, they are *disconnected* from the remainder of the preexistent tool set, so they impose an unnecessary usage overhead (e.g., context switches, data synchronization) on its users. Second, provisioning such application as per-computer stand-alone installation is complex, and, especially in industry contexts, requires the application to comply to company-specific security regulations etc. Third, to enable any analysis benefits, specific knowledge not only about metrics is required, but also about how to express quality aspects as metric mappings [Kitchenham and Pfleeger 1996], which currently have to be configured per installation. This can (partially) be alleviated by providing managers and developers with (initial) training that focus on the company's individual engineering process(es), its requirements, and related quality issues [Broy et al. 2006; Deissenboeck et al. 2009]. Last, stand-alone applications cannot be integrated in web mashups, which further limits their use.

To overcome these drawbacks, we suggest integrating software-map related use cases into the (web-based) software engineering tool set, including tools such as collaboration platforms, frameworks, or web-interfaces of source code repositories (Figure 2). Hence, we *connect* software maps to existing collaboration platforms by a web-based implementation that also centralizes metrics configuration. Our system delivers fast loading times and interactive frame rates to ensure that the visualization does not interfere with the exploratory nature of the targeted use cases. We achieve this by (1) introducing a method for efficient data provisioning, using JSON and Typed Arrays, for interactive visualization of software maps and 3D-Treemaps in general, including color encoding, normal generation, and geometry retrieval; (2) an image-based rendering pipeline implemented in JavaScript and WebGL that uses stylization techniques specialized on the visual exploration of code metrics through software maps, (3) an importance-based LOD-filtering for labeling, and (4) a set of interaction techniques specifically combined to support the given use case. The system is accessible through WebGL-enabled [Marrin 2013] browsers, available on stationary and various types of mobile computers (e.g., tablets, notebooks, light desktop or office computers, etc. – Figure 1). We demonstrate that our system scales up to real-world data, and evaluate its performance (provisioning, rendering) for integrated consumer graphics hardware.

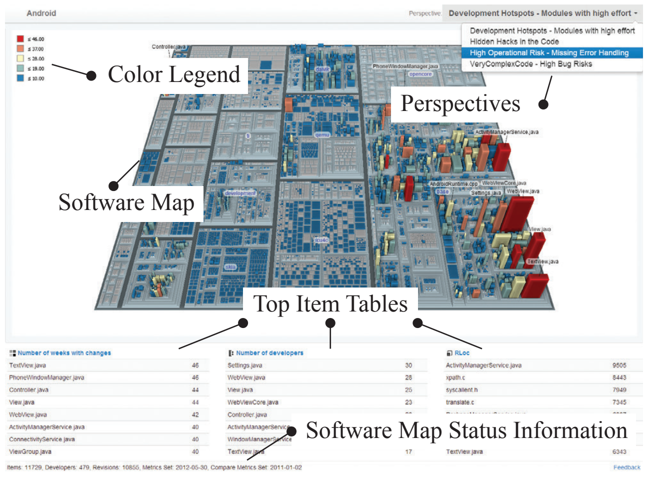
2 Related Work

Visualization of data is in many aspects of high importance, providing meaningful insights for diverse groups of information consumers [Langelier et al. 2005; Bürger and Hauser 2007]. As such, attributed 3D-Treemaps were suggested for software analysis, in the form of Code Cities [Wettel and Lanza 2007] or more recently as software maps [Bohnet and Döllner 2011]. The later focusing on aiding software engineering processes by monitoring certain source code aspects through metrics. Similarly, integrating software maps with the common software-engineering tool set was suggested [Trümper and Döllner 2012], but focuses on using software maps as additional front-end for recommendation systems.

The generation of treemap layouts, and thereby the provision of a spatial context to source code artifacts, various algorithms have emerged (e.g., [Johnson and Shneiderman 1991]) for which an excellent resources for further exploration is available [Schulz 2011]. For software maps we build on the strip treemap algorithm [Bederson et al. 2002] that among others, was recently evaluated and improved [Tak and Cockburn 2012]. While designing our web-based visualization and its data encoding, we evaluated state-of-the-art technologies [Behr et al. 2010; Behr et al. 2012], but neither fitted our needs for exclusively rendering software maps; E.g., the additional latency introduced by requiring multiple requests to access distributed precision data over multiple textures seems inappropriate, and requires further investigation for various use cases. Similarly, progressive (or LOD based) data provisioning [Alliez and Desbrun 2001] is impractical, unless it is desired to provide geometry data per hierarchy level and thereby allowing stepwise software map refinement.

Concerning efficient rendering of software maps or 3D-Treemaps in general, a technique utilizing geometry shaders was recently proposed [Trapp et al. 2013]. In WebGL, however, geometry shaders are not yet available, so we use basic rendering via attribute arrays. The suggested techniques as unsharp masking [Luft et al. 2006], edge-enhancement [Nienhaus and Döllner 2003], or use of camera lights, are directly integrated into our deferred rendering. In addition, screen space ambient occlusion techniques, e.g., [McGuire 2010], increase the perceived software maps plasticity, aiding perception through better item and module differentiation and depth cues. Due to the limitations of WebGL, several tricks for data encoding into attribute arrays or G-Buffers (e.g., for texture encoding ids and depths), similar to [Jung et al. 2013], are necessary.

The problem space of labeling 3D-Treemaps is similar to the labeling of geovirtual 3D environments, hence, several works can be found. We especially relate to the labeling approach introduced by [Hagedorn et al. 2007]. Labeling of empty spaces (i.e., due to padding) between modules as in [Maass and Döllner 2007] is not used, since a meaningful orientation and unambiguous placement of labels seems not possible. Though not implemented, the layouting by [Lü and Fogarty 2008] seems promising for better label placement in software maps. Recently suggested approaches to overcome occlusion of labels in 3D navigation maps [Vaaraniemi et al. 2012], might similarly enhance perceptibility of labels in 3D-Treemaps.


Android
Perspective: Development Hotspots - Modules with high effort
Development Hotspots - Modules with high effort
Hidden Hacks in the Code
High Operational Risk - Missing Error Handling
VeryComplexCode - High Bug Risks
🟥 ≤ 46.00
🟧 ≤ 37.00
🟨 ≤ 28.00
🟩 ≤ 19.00
🟦 ≤ 10.00
Color Legend
Perspectives
Software Map
Top Item Tables
Software Map Status Information
Feedback

Number of weeks with changes	
TextView.java	46
PhoneWindowManager.java	46
Controller.java	44
View.java	44
WebView.java	42
ActivityManagerService.java	40
ConnectivityService.java	40
ViewGroup.java	40

Number of developers	
Settings.java	30
WebView.java	28
View.java	25
WebViewCore.java	23
Controller.java	[illegible]
ActivityManagerServic	[illegible]
WindowManagerService	[illegible]
TextView.java	17

RLoc	
ActivityManagerService.java	9505
xpath.c	8443
syscallent.h	7949
translate.c	7345
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
TextView.java	6343

Items: 11729, Developers: 479, Revisions: 10855, Metrics Set: 2012-05-30, Compare Metrics Set: 2011-01-02

Figure 3: *The user interface of our web front-end, composed using HTML5 and CSS, comprises the following UI elements: The software map visualization itself as centerpiece offers a large area for exploration, a color legend for providing a context to the software-map's color mapping, and in the upper right the selection of predefined perspectives. For each of the three metrics some of the most important code units are listed below the visualization, enabling batch-highlighting individual code units in the software map.*

3 Concept and Design

For our system, two instruments build orthogonal entry points for any subsequent exploratory analysis: (1) *perspectives* and (2) *dashboards*, which are outlined next.

1. A perspective is a named, predefined set of metric mappings for software maps, by which for each of the three available visual attributes of the treemap items (resp. *code units*) – color, height, and ground area – a metric can be selected. Such perspectives are defined in the context of a software project and allow for analyzing different aspects of the code base. Additionally, time or revision parameters, labeling configuration, and further parameters are specified. Other perspectives can be selected at run-time, triggering a reconfiguration of the software map.

   Since each perspective focuses on a specific set of metrics (or code aspect), expert users (i.e., lead developers and architects) typically reconfigure the software map multiple times (by selecting other perspectives) during an exploration session to correlate those aspects. Expert analysis with software maps is a highly iterative and exploratory process that is executed for building up and verifying hypotheses by examining orthogonal aspects of the code base [Trümper and Döllner 2012]. Hence, fast perspective switching is an essential non-functional requirement for any system implementing software maps to ensure good user experience.

2. A dashboard provides an overview of best or worst performers for a number of preselected metrics. Our implementation displays (1) lists of important code units, whose importance is given by predefined metrics (or perspectives) that are relevant to the individual software engineering process, and (2) up-to-date snapshots of all available perspectives using a default camera position. The snapshots enable direct jumps to related perspectives for further exploration. Further, (3) circular status indicators (inspired by a bootstrap template [creativeLabs 2013]) give additional visual cues for specific metrics and further improve fast inspection and software map selection.

   In contrast to perspectives, dashboards typically function as entry point for managers to gain quick overview of key performance indicators (KPIs), and occasionally lead to in-depth analysis of those aspects by means of individual perspectives.

In this paper, we only briefly outline the usage of entry point 2, dashboards, and focus on the use of entry point 1, perspectives. Hence, users begin their interactive exploration session by selecting a perspective. Subsequently, the software map is updated by handling layout changes, rendering the 3D-Treemap, coloring and labeling relevant code units.

A perspective is generally defined by selecting three metrics from a multitude of code metrics. Here, metrics can be either common code metrics (e.g., lines of code) or project-specific metrics (e.g., aggregated metrics or style guide specifics). When a metric is assigned to one of the treemap attributes, specific mapping parameters are configured. For color mapping either a discrete or continuous color scheme is required. Height mapping is accomplished through a scale function with a predefined maximum height. Area mapping, is also specified by an arbitrary scale function, controlling the code units relative base area. In addition to defining the mapping, various kinds of filters can be applied, e.g., for viewing all items touched by a certain author or team or all revisions referencing a certain bug. All in all, defining perspectives is an expert task that requires a good understanding of available metrics and the respective aspect of the code base that is to be modeled by such perspective.

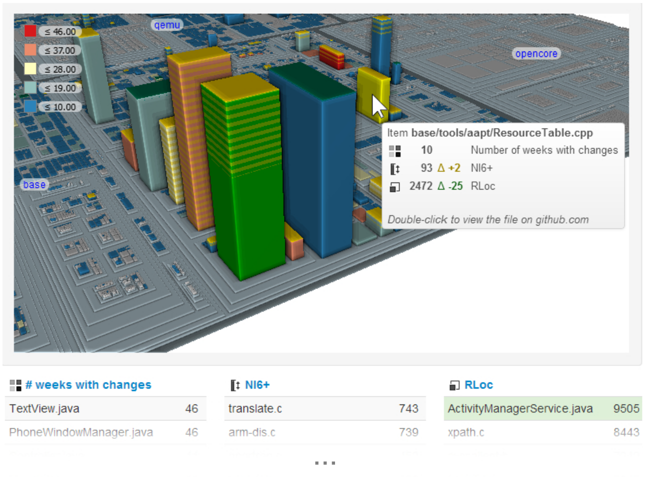
Once a perspective is defined and applied, we continue by rendering the software map and handling user interaction with it. For our first prototypes we evaluated several WebGL Engines: ThreeJS, SceneJS and PhiloGL. ThreeJS and SceneJS would've needed much customization, and matching our performance needs in their rendering pipeline would've caused similar effort as our custom implementation. Due to SceneJS's scene graph and draw list implementation, no interactive frame rates where achieved for rather small software maps. Finally, PhiloGL, as a easily extensible, fast, and lightweight engine, seems to fit best for this task.

The presented system, however, utilizes no third parties for rendering nor for generalized scene graph management. The WebGL canvas is integrated into HTML and also supports alpha blending, therefore allowing to place other, arbitrary HTML elements before or behind the visualization. This is used to display a color legend and text labels for code units without the need to render them within the software map's virtual 3D space.

In the following sections we focus on server-side data provisioning, through efficient geometry and hierarchy data encoding, for browser based rendering. For this OpenCTM [Geelnard 2013] was evaluated as a carrier format, but, especially for our data, a more efficient data encoding is obtainable. The visualization process is presented in more detail, G-Buffers encoding schemes are provided, and adjustments made to treemap layouting and labeling, as well as interaction specifics are described.

4 Software-Map Data Provisioning

A software map, as a representation of a tree, consists of modules as nodes and items as leafs, both represented as boxes. Modules represent directories or similar project structuring entities. Items represent individual code units. The height of an item is defined by the relative value of its assigned metric value whereas the height of a module is constant. The generic terms module and item are preferred to avoid confusion with project specific language and struc-



Item **base/tools/aapt/ResourceTable.cpp**

10		Number of weeks with changes
93	Δ +2	NI6+
2472	Δ -25	RLoc

Double-click to view the file on github.com

# weeks with changes	
TextView.java	46
PhoneWindowManager.java	46

NI6+	
translate.c	743
arm-dis.c	739

RLoc	
ActivityManagerService.java	9505
xpath.c	8443

...

Figure 4: *This section shows various features of our visualization within the browser. In the upper part, importance-based, separate labeling of code units and modules is applied. In the center, a tooltip containing detailed information like exact metric values and differences, metric names, and the code units path or name. In addition, highlighting of user picked code units, visible as green code units both either item tables and visualization, is shown. In this case, the android project is shown, for which attributes like, e.g., McCabe's complexity, real lines of code, coverage, or number of developers can be visualized. For example, the ActivityManagerService stands with its high nesting level and RLoc count. It turns out that this item is regularly manipulated by many different developers and thus, could be considered as a potential quality risk.*

ture conventions. Software projects based on ABAP[1] for example, have no file structure but are based on tables and development objects or classes. Most C++ projects in contrast, are file based projects with classes; commonly one principle class per header and source pair.

For the representation of the software map two data structures are used: a JSON-based [JSON 2006] hierarchy and binary geometry data represented by typed arrays [Khronos Group 2012]. Both are transmitted via AJAX requests in a compressed way (HTTP compression, i.e., GZIP [GZIP 1996]). This reduces the transmitted data by up to 65% (Table 1).

4.1 Hierarchy Data

The hierarchy is a JSON structure, where the keys are item ids and the values arrays with information about these items (Figure 5). The first entry is the *box geometry index*; A pointer into the render geometry that is used for highlighting (i.e., to look up where in the color array the color for this specific item is stored so it can be exchanged for the highlight color) and interactive exploration. The third entry is the item name. To assemble the full item path (e.g., for the tooltip), the second entry, parent id, is used to recursively query the hierarchy and concatenate the item/module names.

Entries 4 to 9 are only present for items. They are the metric values and differences (between the last and current metric set) for the three mapping dimensions and are shown in tooltips (Figure 4).

The complete hierarchy can become quite big, especially item/module names take up much space (Table 1). To be able to show something quickly, we transmit the hierarchy in two parts. The first one is sent together with the so-called report. The report contains meta information like project name, number of items, authors, revisions, the name of the perspective, metric names and explanations, and the ids of all labeled items and modules. The hierarchy part of the report contains information about the n (i.e., 20) top items per metric and all labeled items and modules. The second part contains the whole hierarchy, the already transmitted items excluded. Once report and geometry are loaded, *top item tables* (Figure 3) and labels on the software map can be displayed.

[1]Advanced Business Application Programming language by SAP

4.2 Geometry Data

The camera navigation is restricted to the upper hemisphere, making bottom faces unnecessary. Even without the camera restriction, a single bottom face for the software map's root module would be sufficient. Concerning attribution, the following attribute-geometry relations need to be taken into account: per-vertex, per-face, and per-box. They need to be provided for the indexed vertices, yielding 30 indices per box and 20 vertices with attributes. For rendering, we primarily need color, diff threshold, and id per box, normal per face, height per vertex, as well as a top vertex classification flag per vertex for diff rendering (*binary height*).

In WebGL, vertex index buffers are currently limited to 16-bit indices. Thus, a maximum of 65k vertices or 3 276 modules and items can be rendered per draw call. Because software maps are usually significantly larger, they are rendered in chunks with a maximum of 65 520 vertices each. Chunks do not need to have the same number of boxes, and a predefined ordering, providing more regular, continuous chunks as, e.g., rectangular shaped patches could be beneficial for frustum culling. Contrarily, chunks are filled up in sequence as emerged by the hierarchy, keeping the number of required draw calls to a minimum and forego time consuming server or client-side reorganization of geometry data.

Since each chunk represents a sequence of boxes, the indexing remains the same for each chunk; Generating only one chunk and using it for all draw calls saves memory on the GPU. During rendering, normals, binary heights, and indices once, and re-bind the arrays for vertices, colors, and ids are bound for every draw call with a different offset (a multiple of the chunk size). The various attribute relations demand a quad based triangle definition, resulting in a large redundancy for per box and per face attributes (Figure 6). Furthermore, indices need to be provided for drawing of single triangles, resulting in 6 indices per quad.

```
{
  ...
  "4814":[      // item id
    4631,       // box geometry index
    4756,       // parent id
    "java"      // item/module name
  ],
  "4848":[
    4710,
    4814,
    "PackageManagerService.java",
      13,  0,   // color value, difference
     162, -4,   // height value, difference
    6807,  1    // area value, difference
  ],
  ...
}
```

Figure 5: *An exemplary excerpt of a JSON file, providing hierarchy and metrics information for the software map visualization.*

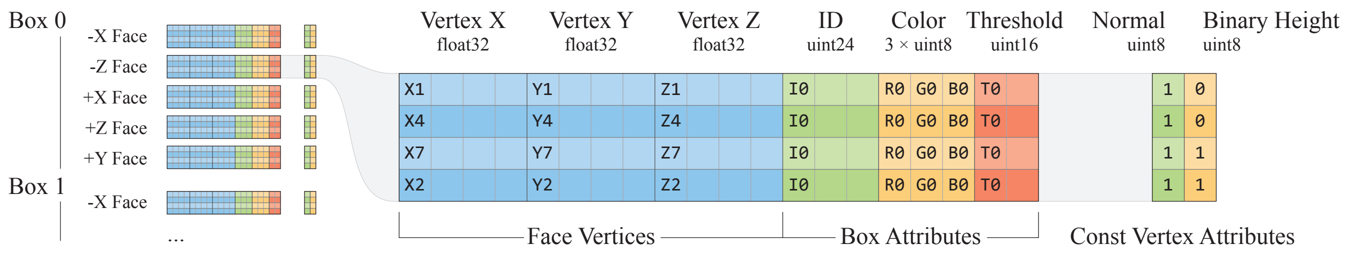


Figure 6: *Drawing a software map chunk accesses a constant index array, two static arrays containing constant vertex attributes (normal as face attribute and binary height as vertex attribute), and chunk specific attribute arrays that contain vertex data and other box attributes. The illustration shows the memory size (one byte per block) and exemplary content of the arrays for the first box's second face of a chunk.*

For the geometry, we transmit compact versions of the arrays used for rendering by a binary AJAX request (response type *ArrayBuffer*). Due to the regular structure of the axis-aligned boxes, indices, normals and binary heights, need not to be transmitted, but can be generated on the client instead. There are four arrays remaining: vertices, colors, ids, diff thresholds. They are stored consecutively and are each preceded by their length (int). During loading, typed views into the received array buffer are created for processing the data. Vertices are transferred as *Float32Array*, containing bounding box information (llf and urb) only. From this the full box geometry of 20 vertices (4 vertices per face, 5 faces) is created. Colors and IDs are transferred as *Uint8Arrays*, diff thresholds as *Int16Array*. They contain the values for each box in the same order as in the vertex array. Since one box has 20 vertices, the data has to be replicated to new arrays, duplicating each date 20 times consecutively. Figure 6 illustrates the content and memory footprint of all generated attribute arrays.

5 Software-Map Rendering

With image-based rendering and deferred stylization, imagery is generated by rendering attributed geometry into textures using render-to-texture while concurrently filling G-Buffers [Saito and Takahashi 1990]. Rendering to multiple targets, however, is not available with WebGL yet (though an extension is in the works: WebGL_draw_buffers), limiting rendering of a single pass to a single G-Buffer. Another limitation in WebGL is, that the support for floating point textures is optional. Therefore float values (like depth) have to be encoded in four 8-bit channels (RGBA). This section describes how G-Buffer data is encoded and used for stylization and exploration of large software maps.

5.1 Efficient Data Encoding

To enable the application of various image-based stylizations, a basic set of two common G-Buffers is provided. The first contains colors and normals (*rgbn-buffer*), the second linearized depths (*depth-buffer*). Box identities are rendered to a third G-Buffer (*id-buffer*), that is mandatory for image-based interaction (i.e., hovering).

The rgbn-buffer contains color and normal information, all of which is encoded by four 4 unsigned bytes. The color calculation uses various vertex attributes and uniforms. Color, diff threshold, binary height, highlight status, highlight and mouse over colors, as well as a procedural texturing are exclusively processed in this pass. Factors that are considered for color calculation are described in more detail in the following subsection. Since only axis aligned, five-sided boxes are rendered for the software map, only five distinct normals are required. These are encoded by indices as follows: $\pm 2 \rightarrow \pm x$, $\pm 1 \rightarrow \pm z$, and $0 \rightarrow +y$. The normal index is stored as an unsigned byte, which is provided as normalized float in $[0;1]$ within shaders. Although no dynamic access to constant arrays within shaders is possible in WebGL, this scheme allows for efficient decoding without branching:

```
vec3 decodeNormal(in float i)
{
    i -= 2;
    float a = sign(i);
    return vec3(i - a, 1 - a * a, 2 * a - i);
}
```

The parameter i is expected in $[0;4]$ and encodes the normal of a single software-map face via signed indices $\{-2,-1,0,+1,+2\}$. Its value specifies the vertex component and its sign the normal direction.

The id-buffer provides an unsigned 24-bit integer identity per box, encoded by three bytes. It is used for fast object picking and further enables enhancement of various stylizations (i.e, edge enhancement). The depth-buffer stores linearized depth values within the range $[0;1]$ and is encoded by three bytes as well. Encoding of 32-bit floats to three single-byte channels, and vise versa, can be done within the shader as follows:

```
vec3 float2vec3i(const in float f)
{
  return vec3(
    floor(      f *    256.0)           / 255.0,
    floor(fract(f *    256.0) * 256.0) / 255.0,
    floor(fract(f * 65536.0) * 256.0) / 255.0);
}
float vec3i2float(const in vec4 v)
{
  return v[0] * 255.0 / (256.0)
       + v[1] * 255.0 / (256.0 * 256.0)
       + v[2] * 255.0 / (256.0 * 256.0 * 256.0);
}
```

To enhance human perception of the software-map hierarchy and to facilitate visually distinction of individual code units, a composition of various stylization techniques can be applied (Figure 7): Static lighting, edge enhancements through contour lines [Nienhaus and Döllner 2003], and either unsharp masking [Luft et al. 2006] or any efficient screen space ambient occlusion (SSAO).

Static lighting, often uses a distance light at the cameras position [Trapp et al. 2013]. For software maps, we suggest to fixate the camera at a constant altitude. This yields a constant, view independent coloring of the software-map's top faces (i.e., all horizontal faces), and further reduces the overall light variation for vertical faces due to camera movement. Colors remain stable for top

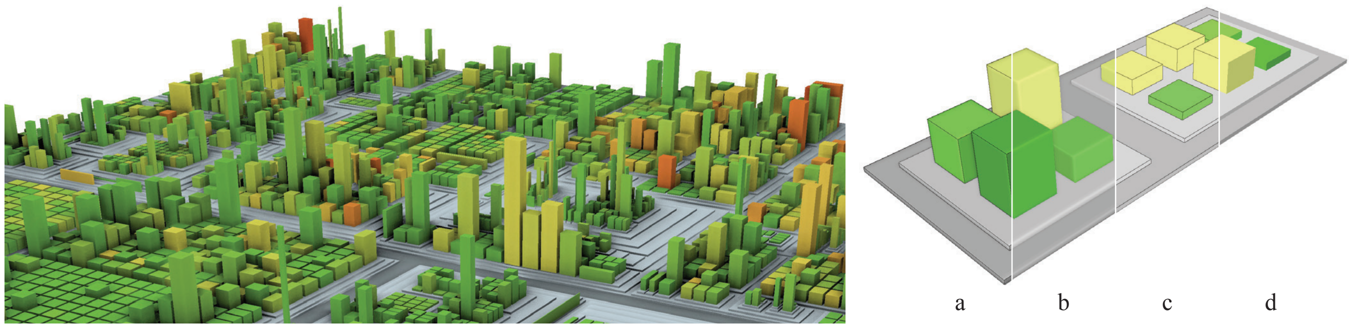


Figure 7: *A composition of various stylization techniques supported by our rendering pipeline. The segments on the right show treemaps rendered with unsharp masking (b), edge enhancement (c), and static lighting (d). All three techniques are combined in the first segment (a).*

faces and can be adjusted to exactly match the color scheme (visual matching of colors of vertical faces becomes also more accurate).

For edge enhancement, contours generated from normal, color, and id buffers are used to add colored contour lines to all visible edges of each box. Depth-buffer unsharp masking increases depth cues and the overall visual quality. Alternatively, SSAO can be used, which further increases the perceived plasticity of the software map. In Figure 4, unsharp masking was used, compared with a more expensive SSAO used in Figure 7. Illumination that affects saturation or hue (e.g., color bleeding as result of global illumination) is not recommended, since they obviously distort the perceptual color mapping and thereby hinder accurately reading metrics encoded in color.

It is often difficult to discern the basic structure of the code base (i.e., the root modules) even with basic padding applied to the treemap layout. To alleviate this, we increase the border between root modules and color it differently (i.e., darker, with neither saturation nor hue change - Figure 7). Since the interesting modules may not be at the top hierarchy level, we can adjust the hierarchy level that is considered the root level.

5.2 Interactive Exploration

A meaningful exploration of software maps requires interactions and displaying of information to be aligned to one another and highly adjusted to the targeted use cases. This section discusses technical details of labeling, highlighting and hovering, tooltips provisioning, and difference visualization, that is the visual emphasis of metric changes.

Small textual annotations of software map artifacts are commonly referred to as *labels*. The process of controlled label placement is *labeling*. We provide multiple ways to find or identify a code unit within the software map: Labeling of top items, labeling of root (or similarly discernible) modules, highlighting via batch selection in *top item lists*, and provision of tooltips for individual items (Figure 4, with black/blue labels in the figure referring to item and module labels). The labeling of software maps is similar to the labeling of geovirtual 3D-environments, since both feature a two dimensional spatial context enriched with height information.

Item labels are connected to their boxes via poles; if they overlap, labels farther away from the camera are moved upwards. We therefore rely on a technique that places labels such that they avoid occluding one another [Hagedorn et al. 2007]. Module labels work slightly different as they are not connected via a pole. Instead, the mental connection to the labeled module is achieved by placing it in the center of the module. Also, we hide module labels if they would be moved more than half of its height since the mental identification with the labeled module can easily get lost if it is not centered properly. To ensure that labels of large modules are not hidden by labels of smaller modules, we sort the labels by module area in descending order before rendering. While the number of shown item labels is a system setting, the number of shown module labels is view-dependent, i.e., labels for smaller modules can be made visible by zooming in.

The camera navigation for software maps is based on and thus similar to common terrain navigation (e.g., Google Earth). It supports panning, zooming, and rotation, wherein the first two require depth buffer access for fulfilling individual constraints.

When a user starts panning, the initial intersection point is used as a reference point and required to remain exactly below the mouse cursor. This is contrary to intersecting with the ground plane, since panning can be initiated by picking a side face at arbitrary height, necessitate a correct panning speed. The first click of the interaction is used to access the depth-buffer and to retrieve the related object-space position p_0. This point is our constraint for panning, which means that for every mouse position there has to be an appropriate positioning for the software map, so that the point under the cursor remains p_0. With this point and the up normal we build a plane, that defines the panning space. For panning, a ray is created, pointing from the screen pixel into the view frustum. This ray then is converted to object space and used to intersect with the plane at p. The delta of p_0 and p is the translation required for our constrained panning. We further require at least one single fragment of the software map to be at the viewport's center by constraining the software-map's center to its own bounding box.

Rotation is applied around the software map central y-axis, since we found users to be disturbed when rotation is around y-aligned axes centered at the initial intersection points. Furthermore, vertical rotation is restricted by a minimum angle to a range above the horizon, and a minimum angular distance to the up vector. This prevents the software map to be seen from below or upside down.

For zooming, the camera should move in respect to the pointed software-map position. This is done by scaling the distance between the pointed position in the scene (again via depth-buffer) and the camera position. Since the software-map's center needs to be constrained to the ground plane, a temporary center is calculated based on the initial intersection with the ground plane and then used to obtain the new viewray-groundplane intersection as new center. With that, zooming can be deliberately used to pan and scale towards a module or items top or side face. This approach allows fast transitions to arbitrary places of interest by zooming only, thus increasing navigation efficiency. For mobile use, i.e., devices with touch enabled displays, a simple mapping of common touch gestures is used (pinch is simultaneously used for rotation and zoom).

Two other aspects of software map exploration are hovering and highlighting. The first causes a recolouring of the hovered item or module for the time being hovered. The second can be applied for multiple items, by batch-selecting items in the *top item lists*. The item's basic color is retrieved from the color attribute and replaced by the hover color, if the item is hovered. Additionally, if the compare threshold attribute contains a valid value, a procedural texture (i.e., horizontal stripes) is applied to mark the difference between the current and the old metric value (yellow stripes as in Figure 4). Finally, double clicking items browses the linked source code and tooltips are used to provide more detailed information for an item.

6 Discussion and Results

The presented visualization is applicable to similar techniques like CodeCities [Wettel and Lanza 2007] and scales up to real-world software projects containing up to 90k code units (Table 1). Although large code bases might contain more code units, applying a one-to-one mapping from code units to treemap items is not useful then: Visual clutter (e.g., too small individual items, occlusion) begins to increase drastically, which generally renders software maps unusable. Hence, pre-filtering of input data – to reduce the number of code units to approx. 100k – has to take place anyway, so that the scalability of our system is sufficient even for larger code bases.

The precision of the depth buffer varies through out different hardware platforms, on some systems resulting in tiny discontinuities, causing visual artifacts on depth-buffer-based post processing (unsharp masking, SSAO).

To ensure the highest possible compatibility while providing good performance on powerful machines like desktop computers with dedicated graphics cards, it will become necessary to provide a baseline functionality based on WebGL 1.0 with optional optimized code paths that make use of the available extensions if they are available. For example, with WebGL_depth_texture we could skip the depth pass and use the depth buffer from the color pass for SSAO. Reducing the number of geometry passes from two to one is very useful, since for large projects ($>$ 50k items) the rendering speed is geometry-bound. With OES_element_index_uint we could increase the chunk size to a few million vertices. The chunked rendering code would remain active (for systems without this extension), but effectively only one chunk would be rendered.

	Android	Customer Project
Code Units #	30 853	90 089
Vertices #	617 060	1 801 780
Triangles #	308 530	900 890
geometry.bin	964 KiB	2 815 KiB
compressed*	404 KiB / 19 ms / 3 ms	1 175 KiB / 56 ms / 11 ms
compression ratio	0.42	0.42
report.json	4 KiB	16 KiB
compressed*	2 KiB / 0 ms / 0 ms	7 KiB / 0 ms / 0 ms
compression ratio	0.51	0.45
hierarchy.json	1 513 KiB	5 429 KiB
compressed*	480 KiB / 21 ms / 6 ms	1 587 KiB / 70 ms / 19 ms
compression ratio	0.32	0.29
GPU Memory	12.082 MiB	34.679 MiB

*gzip (zlib) compression level 1, with compress and decompress times.

Table 1: *Enlisted are file sizes of all files required for the visualization. System: Intel Xeon CPU E5-1620 0 at 3.6 GHz.*

	Android	Customer Project
Transmission at 1.5 Mbps (synthetic):		
Geometry + Report	5.04 s	14.75 s
Hierarchy	7.88 s	28.28 s
Transmission at 7.6 Mbps (synthetic):		
Geometry + Report	1.00 s	2.91 s
Hierarchy	1.55 s	5.58 s
Local initialization on i5-3337U:		
Interface	0.24 s	0.24 s
Geometry + Report	0.48 s	2.13 s
Hierarchy	1.11 s	4.27 s
Averaged frame rendering time:		
Intel HD Graphics 4000	5.62 ms	28.06 ms
Intel Core i5-3337U at 1.80 GHz (WIN)		
NVIDIA GeForce GT 650M	1.92 ms	13.62 ms
Intel Core i7 at 2.80 GHz (OS X)		
NVidia GTX 680	2.20 ms	16.04 ms
Intel Xeon W3530 at 2.80 GHz (WIN)		

Table 2: *Enlisted are transmission, initialization, and average times per frame for rendering software maps of the two previous projects. Transmission and initialization times were measured separately, since the software map is available for interaction with geometry and report data provided, except for tool tips. Local initialization times were measured with Google Chrome's tracing capability, the influence of the tracing itself are ignored. The frame times were averaged across 1000 frames of continuous redraw with no interaction (by default, redraw is only triggered on interaction). Chrome 26.0.1410.64 was used as browser (WebKit 537.31, JavaScript V8 3.16.14.11). The resolution of the website* 1024×1280 *and for the OpenGL canvas* 976×732 *pixel. For Windows, ANGLE was utilized by chrome, which surprisingly performed faster than direct OpenGL rendering.*

Concerning rendering performance, we measured transmission, initialization and frame rendering times on three platforms: one ultrabook, one mac book pro, and one workstation. For all devices, interactive frame rates have been met and transmission and initialization delays are reasonable fast within our use-cases (Table 2).

7 Conclusion and Future Work

We have presented a web-based system for the interactive display and analysis of large software maps. We achieve this technical scalability by efficiently encoding the data transferred to the client and by optimized usage of vertex attribute arrays. We further address visual scalability of the software-map rendering by applying an importance-based LOD-filtering for labels that annotate treemap items. A combination of interaction techniques for easy manipulation of the camera as well as details-on-demand features for querying additional properties of individual treemap items are provided.

Introducing filtering for modules, metrics, developers, code units etc. on the client side would be beneficial for the visualization's exploratory nature. Layouting and configurable attribute mapping allowing for arbitrary metric-mapping parametrization from within the browser could increase the scope of web-based software-map analysis further. Concerning data provisioning, we found that general-purpose storage containers (e.g., [Behr et al. 2010], [Behr et al. 2012], [Jung et al. 2013]) can be useful in most rendering scenarios, however they do not provide the efficiency of our approach (two vertices per box, various attributes, decoupled hierarchy, etc.).

The availability of a web-based software-map implementation now allows for connecting software maps to collaboration platforms, such as Github[2], which improves user experience and lowers the access barrier. With the centralized provisioning of (predefined) perspectives, users can now collaboratively re-use perspectives without being forced to define any themselves. We hope, more APIs promoting interactive web-based software maps or similar visualizations become available, making software visualization more accessible and supporting today's software engineering processes.

Acknowledgements

We thank Software Diagnostics GmbH[3], for sharing their ideas and providing us access to their Software Diagnostics Transparency Platform. This work was partially funded by the Federal Ministry of Education and Research (BMBF), Germany within the InnoProfile Transfer research group "4DnD-Vis".

References

ALLIEZ, P., AND DESBRUN, M. 2001. Progressive compression for lossless transmission of triangle meshes. In *Proc. of ACM SIGGRAPH*, 195–202.

BEDERSON, B. B., SHNEIDERMAN, B., AND WATTENBERG, M. 2002. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Trans. Graph. 21*, 4, 833–854.

BEHR, J., JUNG, Y., KEIL, J., DREVENSEK, T., ZOELLNER, M., ESCHLER, P., AND FELLNER, D. 2010. A scalable architecture for the html5/x3d integration model x3dom. In *Proc. of ACM Web3D*, 185–194.

BEHR, J., JUNG, Y., FRANKE, T., AND STURM, T. 2012. Using images and explicit binary container for efficient and incremental delivery of declarative 3d scenes on the web. In *Proc. of ACM Web3D*, 17–25.

BOHNET, J., AND DÖLLNER, J. 2011. Monitoring code quality and development activity by software maps. In *Proc. of ACM MTD*, 9–16.

BROY, M., DEISSENBOECK, F., AND PIZKA, M. 2006. Demystifying maintainability. In *Proc. of ACM WoSQ*, 21–26.

BÜRGER, R., AND HAUSER, H. 2007. Visualization of multivariate scientific data. In *Proc. of EG STARs)*, 117–134.

CREATIVELABS, 2013. Perfectum dashboard. http://wbpreview.com/previews/WB0PHMG9K/ui.html.

DEISSENBOECK, F., JUERGENS, E., LOCHMANN, K., AND WAGNER, S. 2009. Software quality models: purposes, usage scenarios and requirements. In *IEEE WOSQ*, 9–14.

EISENBARTH, T., KOSCHKE, R., AND SIMON, D. 2003. Locating features in source code. *IEEE Transactions on Software Engineering 29*, 3, 210–224.

GEELNARD, M., 2013. Openctm. http://openctm.sourceforge.net/.

GZIP, 1996. Gzip file format specification. http://tools.ietf.org/html/rfc1952.

HAGEDORN, B., MAASS, S., AND DÖLLNER, J. 2007. Chaining geoinformation services for the visualization and annotation of 3d geovirtual environments. In *4th International Symposium on LBS and Telecartography*.

JOHNSON, B., AND SHNEIDERMAN, B. 1991. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proc. of IEEE VIS*, 284–291.

JSON, 2006. The application/json media type for javascript object notation (json). http://tools.ietf.org/html/rfc4627.

JUNG, Y., LIMPER, M., HERZIG, P., SCHWENK, K., AND BEHR, J. 2013. Fast and efficient vertex data representations for the web. In *Proc. of IVAPP*, SCITEPRESS Science and Technology Publications, 601–606.

KHRONOS GROUP, 2012. Typed array specification. http://www.khronos.org/registry/typedarray/specs/latest/.

KITCHENHAM, B., AND PFLEEGER, S. L. 1996. Software quality: The elusive target. *IEEE Softw. 13*, 12–21.

LANGELIER, G., SAHRAOUI, H., AND POULIN, P. 2005. Visualization-based analysis of quality for large-scale software systems. In *Proc. of ACM ASE*, 214–223.

LÜ, H., AND FOGARTY, J. 2008. Cascaded treemaps: examining the visibility and stability of structure in treemaps. In *Proc. of GI*, 259–266.

LUFT, T., COLDITZ, C., AND DEUSSEN, O. 2006. Image enhancement by unsharp masking the depth buffer. In *Proc. of ACM SIGGRAPH*, 1206–1213.

MAASS, S., AND DÖLLNER, J. 2007. Embedded labels for line features in interactive 3d virtual environments. In *Proc. of ACM AFRIGRAPH*, 53–59.

MARRIN, C., 2013. Webgl specification. https://www.khronos.org/registry/webgl/specs/latest/.

MCGUIRE, M. 2010. Ambient occlusion volumes. In *Proc. of High Performance Graphics 2010*.

NIENHAUS, M., AND DÖLLNER, J. 2003. Edge-enhancement - an algorithm for real-time non-photorealistic rendering. *Journal of WSCG*, 346–353.

POOLE, W. 2003. The softer side of custom software development: working with the other players. In *Proc. of CSEE&T*, 14–21.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. In *Proc. of ACM SIGGRAPH*, 197–206.

SCHULZ, H.-J. 2011. Treevis.net: A tree visualization reference. *IEEE Comput. Graph. Appl. 31*, 6, 11–15.

TAK, S., AND COCKBURN, A. 2012. Enhanced spatial stability with hilbert and moore treemaps. *IEEE Trans. Vis. Comput. Graph. 99*.

TRAPP, M., SCHMECHEL, S., AND DÖLLNER, J. 2013. Interactive rendering of complex 3d-treemaps. In *Proc. of GRAPP*, SCITEPRESS Science and Technology Publications, 165–175.

TRÜMPER, J., AND DÖLLNER, J. 2012. Extending recommendation systems with software maps. In *IEEE RSSE*, 92–96.

VAARANIEMI, M., FREIDANK, M., AND WESTERMANN, R. 2012. Enhancing the visibility of labels in 3d navigation maps. *Lecture Notes in Geoinformation and Cartography (LNG&C)*.

WETTEL, R., AND LANZA, M. 2007. Visualizing software systems as cities. In *IEEE VISSOFT*, 92–99.

[2] `http://www.github.com`

[3] `http://www.softwarediagnostics.com`