Eurographics/ IEEE-VGTC Symposium on Visualization (2007)
Ken Museth, Torsten Möller, and Anders Ynnerman (Editors)

Multiscale Visualization of Dynamic Software Logs

Sergio Moreta and Alexandru Telea

Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, the Netherlands
s.moreta@student.tue.nl, alext@win.tue.nl

Abstract

We present a set of techniques and design principles for the visualization of large dynamic software logs consisting of attributed change events, such as obtained from instrumenting programs or mining software repositories. We enhance the visualization scalability with importance-based antialiasing techniques that guarantee visibility of several types of events. We present a hierarchical clustering method that uncovers several patterns of interest in the event logs, such as same-lifetime memory allocations and software releases. We visualize the clusters using a new type of technique called interleaved cushions. We demonstrate our methods on two real-world problems: the monitoring of a dynamic memory allocator and the analysis of a software repository.

1. Introduction

In this paper, we present an approach for the visual analysis of time-dependent data obtained from software logs. We consider two types of logs: profiling logs and source code evolution logs. These are typically weakly structured datasets containing hundreds of thousands of low-level events. Developers often need to ask questions at a higher level than reflected by the log data. Hence, we need ways to create *and* show structure from these unstructured logs.

We propose a set of techniques and design elements to construct scalable visualizations of dynamic log data. First, we introduce several anti-aliasing techniques to render large log datasets which guarantee different visibility types addressing different user queries, even for subpixel items. Second, we present a clustering method for creating hierarchical visualizations which answer several structure-related questions. Finally, we introduce interleaved shaded cushions, a new visualization technique for rendering the data hierarchy which allows one to easily follow and discover structures. We demonstrate our techniques on two different problems: The testing of a dynamic memory allocator and the evolution analysis of a large source code repository.

This paper is structured as follows. Section 2 overviews efforts in visualizing software logs. Section 3 describes our data model. Section 4 presents our visualization design. Section 5 describes our new clustering techniques that reveal global evolution patterns. Section 5.2 presents the interleaved cushions used to visualize clusters. In section 6 we apply our techniques on two real-life applications. Finally, section 8 summarizes our findings and outlines future research directions.

2. Related work

Several methods exist for visualizing time-dependent software log data. For the study of memory allocators, logs consist of allocation and deallocation events, and various metrics, e.g. memory fragmentation, occupancy, and block size distribution [GT89]. Logs are created by code instrumenting and profiling tools [ACS90, JG94, WKT04] and visualized by applications such as Rivet [BST*00, Bos01], LynxInsure [Lyn06], Polka [SK93], and the more general TANGO animation framework [Sta92]. However, time-dependent metrics show only aggregated facts and little structural insight. Our second application is visualizing source code evolution logs from code repositories. These record events such as code lines added, modified, or deleted, author names, file types, and modification date [VT06b]. Repository logs have been visualized using 2D orthogonal layouts of sets of axis-aligned, attribute-colored, densely packed, rectangles [VT06a]. Such methods suffer from serious aliasing when the drawn rectangles are smaller than one pixel line. Extra structure can be generated, e.g. to emphasize groups of files having similar evolutions, using hierarchical clustering and luminance cushions [VT06b, VT06a]. However, this technique considers the evolution of *entire* files only. Moreover, the luminance cushions, as proposed in [VT06a], are quite hard to interpret visually.

3. Data model

Our generic software log data is a set of artifacts $S = \{e^i\}$. An artifact has an own identity (i) but can change at several moments t_j yielding the a set of artifact versions $\{e^i_j\}_j$. With each event $\{e^i_j\}$, data attributes a^i_j can be associated. In this paper, we consider two artifact types from two different applications. In the first case, an artifact $e^i = (a^i, b^i)$ is a memory address range $[a, b]$. An element e^i_j is an allocation or deallocation of the memory block $[a^i, b^i]$ taking place at moment t_j, and has several attributes: the memory address range, calling process ID, and allocator-specific attributes (detailed in Sec. 6.1). We create a log file S containing all (de)allocation events recorded by the memory allocator, by instrumenting the C library functions `malloc` and `free`. In our second example, artifacts e^i are files stored in a CVS or Subversion repository. The repository log S records the changes e^i_j of all files i at moments t_j. An event e^i_j has several data attributes: the commit author, a commit message, and the amounts of added and removed code lines with respect to the previous change. The contents of a file between two successive changes is called a file version. We extract such data using the `CVSgrab` repository analysis tool [VT06a].

4. Visualization model

We visualize the artifact space S described in Sec. 3 as follows. First, we use a 2D Cartesian layout which maps discrete event time j and artifact identifier i to the x and y axes respectively. For the memory log data, the artifacts' y axis order is implicitly given by their memory addresses. For the repository log, artifacts (files) are ordered on the y axis following a depth-first traversal of the repository root, so files in the same directory get laid out close to each other on the y axis. Every element $\{e^i_j\}_j$ is an axis-aligned rectangle. This layout has several advantages. It is *compact* or dense, so hundreds of thousands of elements can fit on a single screen. No screen space is wasted. Empty areas convey actual information, e.g. they indicate memory fragmentation for the allocator log data (Sec. 4.1). Every artifact $e^i \in S$ maps to a distinct horizontal strip, so the visualization is intuitive. Encoding time on the x axis is a natural choice. Finally, this layout is simple and fast to compute. After layout, we color every rectangle to show a data attribute a^i_j via a suitable color mapping scheme. To separate same-color neighbor elements, we overlay each element with a luminance cushion, dark at the element border and bright at its center. We use both parabolic cushions [vWvdW99] and plateau cushions [LNVT05].

Figure 4 illustrates the basic idea for a memory allocation log dataset containing 119932 allocations spanning a period of 4 minutes done by 54 concurrent processes. Color shows the allocating process ID. This image already reveals several aspects: The "blue" process allocates the most memory. Since the y axis maps one-to-one to the memory address space, the long rectangles at the image bottom show that the "blue" process allocates memory early and frees it as last in this scenario. After start, almost no extra memory is allocated in the first third of the monitored period. Next, the "green" process rapidly allocates many equal-sized blocks, all at one moment, and frees them at the same time too, as shown by the thin vertical green stripes. We discovered that this pattern of same-lifetime blocks is typical for *array* objects. These arrays use about half of the free memory (y axis), so they are quite important. The second third of the period shows a dynamic allocation-freeing pattern which almost fills up the entire memory at some moments. In the last third, there are few allocations. At the end, all memory is freed.

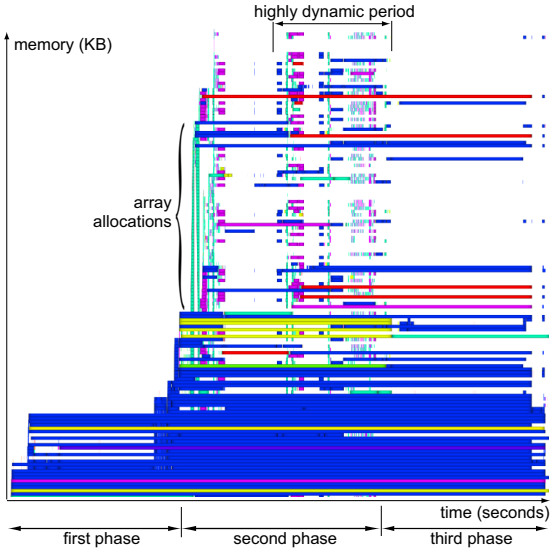

Figure 1: *Dynamic memory allocation visualization*

4.1. Importance-Based Anti-Aliasing

However useful, the visualization in Fig. 4 has some problems. Some memory logs have hundreds of thousands of allocations. Similarly, repository logs contain hundreds of versions of thousands of files [LNVT05, VT06b, VT06a]. Rendered on a typical screen, this yields rectangles smaller than one pixel in one or both dimensions. Increasing the screen size is not a solution, as the axes sampling can be highly non-uniform. For both memory and repository logs, the time (x axis) sampling can be extremely dense compared to the time range. The same happens on the y axis e.g. when the memory log contains blocks of a few bytes on a total range of e.g. megabytes.

The question is: How to color a pixel covered by K segments $e_1, \ldots, e_K$ of which we want to show the attribute values $a_1, \ldots, a_K$? Figure 2 a shows allocated blocks colored by process ID. Figure 3 a shows the evolution of 2792 files over 850 versions spanning over 10 development years from the Visualization Toolkit (VTK) repository. Color shows the amount of changed lines (blue=unchanged, red=maximal change over the whole project life). This image addresses the

question "which files have changed least/most, and when?" Yet, figures 2 a and 3 a are misleading when displayed in color e.g. on a computer screen. In both cases, the drawn rectangles are of subpixel size. Simply drawing all rectangles shows only the color C of the last drawn element $C = c(a_K)$, where c is the scalar-to-color mapping function, which amounts to a regular dataset undersampling. More gravely, subpixel-width rectangles, e.g. allocations quickly followed by deallocations, become invisible. This is actually what happens in area A in Fig. 2 a. Seeing this image, we first thought we found an allocator bug, as it indicates a free memory range (white space) between two occupied ranges (colored space).

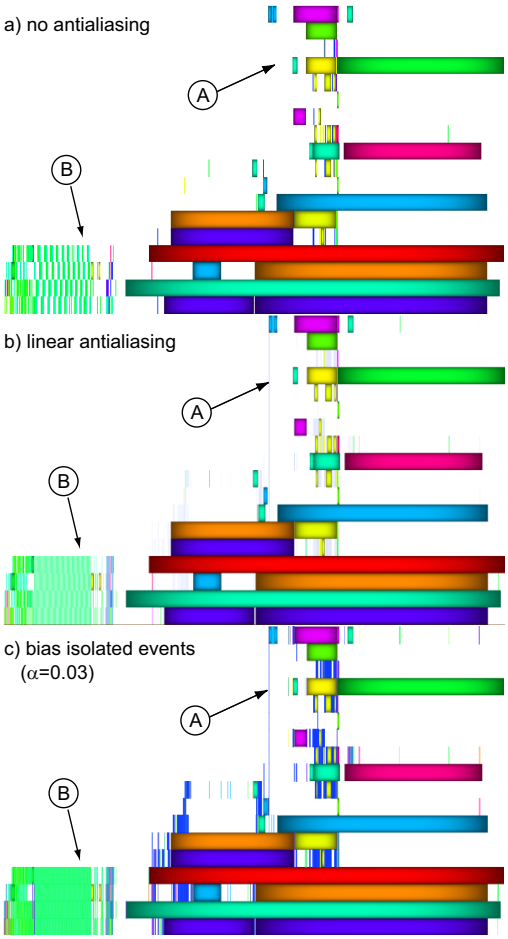

Figure 2: *Importance-based antialiasing for memory log*

A better solution is to use anti-aliasing. If each element e_i covers a fraction $f_i \leq 1$ of a pixel, we compute its color C by blending the color mapping the attributes' weighted average with the background color c_B

$$C = \sum_{i=1}^{K} f_i c \left(\sum_{i=1}^{K} f_i a_i \right) + \left(1 - \sum_{i=1}^{K} f_i \right) c_B \qquad (1)$$

This makes previously invisible, thin segments visible (Fig. 2 b). We also restrict rectangles that fully cover one or more pixels to the fully covered pixels only, i.e. neglect fractional coverages f_i. This wins more space for those thin subpixel-size rectangles. For non-numeric attributes a_i^j, e.g. author names or IDs, we do the color mapping first in Eqn. 1, followed by weighted averaging.

Looking now carefully in area A, we see a very thin vertical line, i.e. a rapid allocation-deallocation pair. Hence, there was no allocator bug. Secondly, area B (Fig. 2 b) looks now, correctly, much more densely populated than its counterpart in Fig. 2 a. Finally, areas such as A (Fig. 3 a) which seemed to show high change (red) due to undersampling appear now in cooler colors (Fig. 3 b), correctly showing a moderate change amount.

Yet, *isolated* thin segments like the one in Fig. 2 a (area A) are still hard to see. These are isolated, short-lived memory blocks in the memory log (thin vertical rectangles), which indicate short-lived temporary variables, or files created at singular moments in the repository log (thin horizontal rectangles). We further emphasize such events with a modified anti-aliasing function

$$C = \frac{F c \left(\frac{\sum_{i=1}^{K} f_i^{\alpha} a_i}{F} \right) + B c_B}{F + B} \qquad (2)$$

where $\alpha > 0$ is a bias factor and $F = \sum_{i=1}^{K} f_i^{\alpha}$ and $B = \left(1 - \sum_{i=1}^{K} f_i\right)^{\alpha}$ are the pixel fractions covered by foreground (element) colors and background color c_B respectively. Low α values emphasize areas containing few *and* thin segments. Comparing Figures 2 b and c or Figures 3 b and c, drawn with $\alpha = 0.03$, we see many isolated events which would have otherwise passed undetected. Conversely, high α values filter out sparsely covered pixels (isolated rapid events), e.g. Fig. 3 d ($\alpha = 3$). This simplifies the view when we are interested only in dense-event areas, e.g. removes the short-lived temporary variables in the memory log view.

5. Hierarchical visualization

Although giving an overview of dynamic log datasets, the visualizations discussed so far contain little structure. We cannot yet address some goals adequately. One such goal is to emphasize same-lifetime memory blocks, typical for array elements, e.g. the green vertical stripes in Fig. 2. Revealing such patterns helps us visually check if a) the allocator serves all memory requests quickly; b) allocates the blocks at consecutive memory addresses, and c) wastes as little memory as possible. Similar questions exist for the repository log data (see Sec. 6.2).

We reveal such patterns using a bottom-up agglomerative clustering. We first discuss the distance metrics used for clustering (Sec. 5.1), followed by a new way to visualize clusters (Sec. 5.2).

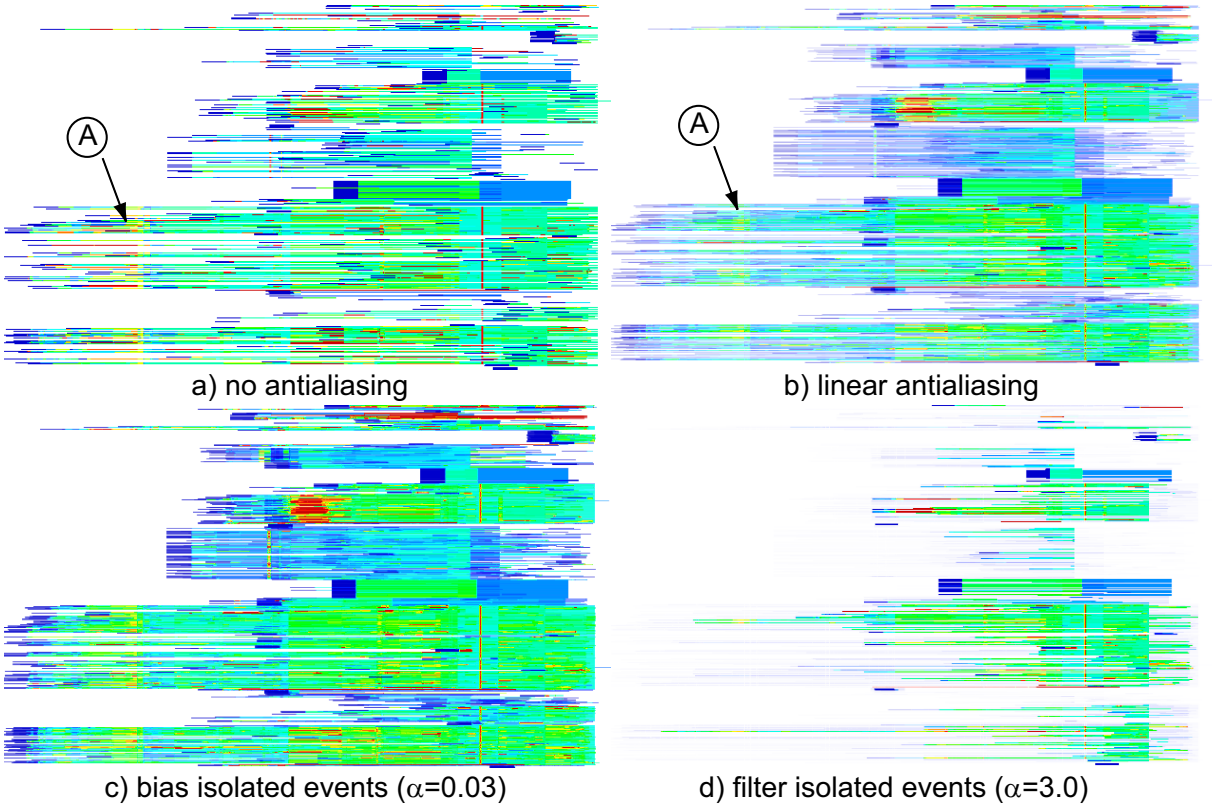

Figure 3: *Importance-based antialiasing for the software evolution log*

5.1. Distance metrics

Our agglomerative clustering uses a distance metric $d : S \rightarrow \mathbb{R}$ to measure the similarity between the log items to cluster. We start with all memory blocks or file versions e_i^j of our log. We repeatedly pick the two elements e_i^j and e_k^l for which $d(e_i^j, e_k^l) = \min$ and merge them into a new cluster until we obtain a single root cluster. Every cluster has an extent E and an area A. For leaves, E is the rectangle given by the start and end attribute values along the x and y axes, i.e. the memory range and lifetime for memory blocks, and file identity and version lifetime for the repository log. The area A is simply the area of E. For non-leaf clusters, E is the rectangular bounding-box of the children's extents and A is the sum of the children's areas. The distance metric choice determines the type of global patterns that will be visible, as follows.

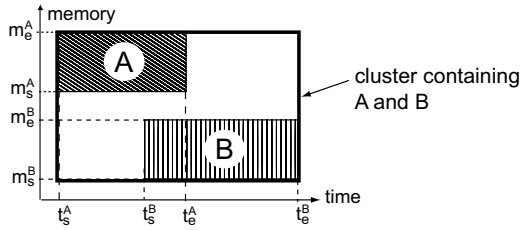

Figure 4: *Distance metric construction*

Consider e.g. two memory blocks A and B with lifetimes t_s^A, t_e^A and t_s^B, t_e^B and memory address ranges m_s^A, m_e^A and m_s^B, m_e^B, respectively (Fig. 4). A good distance metric is

$$d_1(A,B) = |t_s^B - t_s^A| + |t_e^B - t_e^A| \tag{3}$$

This distance clusters elements with similar lifetimes, e.g. blocks allocated and freed at similar moments in the memory log, or files changed at similar moments in the repository log. Sometimes, however, not all log items are equally important. For example, items having short lifetimes $t_e - t_s$, such as rapid allocation-deallocation events corresponding to temporary variables, are less important than items with longer lifetimes. Clustering such high-frequency events first unclutters the visualization. For this we use the distance

$$d_2(A,B) = \frac{\mathcal{A}(A) + \mathcal{A}(B)}{2\mathcal{A}_{max}} d_1(A,B) \tag{4}$$

where $\mathcal{A}(X) = (t_e^X - t_s^X)(m_e^X - m_s^X)$ is the area of cluster X and $\mathcal{A}_{max}$ is the maximal element area over the entire log dataset S. Clearly, Equations 3 and 4 can be used on both leaf and non-leaf cluster.

The distance metrics d_1 and d_2 do not constrain clustering along the y axis, i.e. can yield non-compact clusters consisting of scattered elements. Sometimes, though, we want to cluster only elements which are strict neighbors along the y axis, e.g. contiguous memory blocks for the memory log

or same-directory files for the repository log (see Sec. 4). Restricting clustering to y-neighbors only also simplifies the visualization by favoring compact clusters. We do this by the distance

$$d_3(A,B) = \begin{cases} d_1(A,B), & \text{if } d_y(A,B) = 0 \\ \infty, & \text{if } d_y(A,B) > 0 \end{cases} \quad (5)$$

where $d_y(A,B) = \min(|m_s^A - m_e^B|, |m_s^B - m_e^A|)$. Finally, by using the 'short lifetime first' metric d_2 instead of d_1 in Equation 5, we can favor compact clusters *and* cluster short-lifetime segments first.

Clustering creates a tree containing all log items as leaves. To visualize this tree at a user-chosen level of detail, we compute a tree *section*. For a tree $T = \{C_i\}$, a section Sec is a set of disjoint clusters $Sec = \{C_j\} \in T, C_i \cap C_j = \oslash \forall i \neq j$, such that the union of leaves of C_j equals the initial dataset S. Two sectioning methods are particularly useful. The *same-error* section contains clusters C_j of similar distances $d(x,y)$ between the clusters' children x,y. The *same-size* section contains clusters C_j of similar number of leaves or 'area' (time extent times space extent) The same-error method targets the goal "group elements which are more similar than a given value". The same-size method targets the goal "show the data partitioned in self-similar groups of similar sizes". In our tool, we can specify either sectioning method and interactively control the size or error parameter to view the cluster tree at different levels of detail.

5.2. Interleaved cushions

As explained in Sec. 5.1, some distance metrics yield non-compact clusters consisting of scattered rectangles. How to visualize such clusters? Color coding does not work, as one can distinguish only around 10 different colors and a section has more clusters. We present here *interleaved cushions*, a new method to render non-compact clusters. The basic idea is simple. For each cluster consisting of several rectangles $C = \{r_i\}$, we compute the bounding box B of the rectangles. Next, we construct a parabolic profile $h(x,y)$ spanning the rectangle B, similar to the so-called cushions used to render treemaps [vWvdW99]. Finally, we render all rectangles r_i, the luminance of a point $(x,y) \in r_i$ being given by $h(x,y)$. Hue can encode extra data attributes.

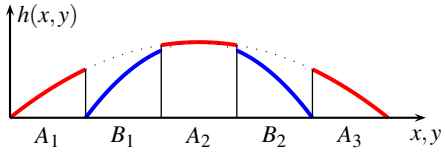

Figure 5: *Two interleaved cushions: $A_1A_2A_3$ and B_1B_2*

Figure 5 sketches the idea in 2D for two clusters containing the non-compact children A_1, A_2, A_3 and B_1, B_2. A cluster can contain non-compact rectangles, but the eye can still find it by following the continuous variation of the parabolic luminance function. Dark ($h(x,y) = 0$) discontinuities are cluster borders. Bright ($h(x,y) > 0$) discontinuities separate rectangles which belong to different clusters. Clusters appear visually as intersecting cushion profiles, hence the name 'interleaved cushions'. If we moreover color each cluster with one color, e.g. picked randomly from a small color set, visually segregating clusters becomes even easier. Figure 6 shows the result. Note the interleaved cushion of sizes w,h in the center which depicts one large cluster. Although this cluster is non-compact, as shown by the thin horizontal bands splitting its cushion, we perceive it as separate following the cushion's smooth luminance variation. Figure 7 shows more examples of interleaved cushions, discussed in detail in Sec. 6.

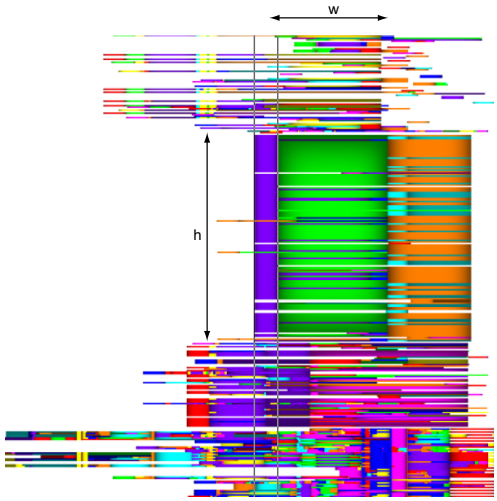

Figure 6: *Interleaved cushions*

Figure 6 uses the *luminance* cushions, where the luminance signal itself is parabolic [LNVT05, VT06a]. *Geometric* cushions [vWvdW99] have a luminance profile of a parabola lit from a certain angle. Although geometric cushions are arguably better for treemaps, their luminance signal is non-symmetric when the light vector is not vertical. We found luminance cushions clearly better for showing non-compact clusters, due to the intuitive symmetry of the luminance signal. Plateau cushions [LNVT05] are best to show individual elements, e.g. Figs. 4 and 8.

6. Applications

We implemented our anti-aliased hierarchical cushions and various navigation, zoom-and-pan, details-on-demand, and brushing functions in an application for visualizing software log data. We discuss our application for two problem domains.

6.1. Dynamic memory allocator analysis

First, we visualize the behavior of a dynamic memory allocator. This allocator, used on a mobile device, serves tens of processes with thousands of `malloc` and `free` calls per

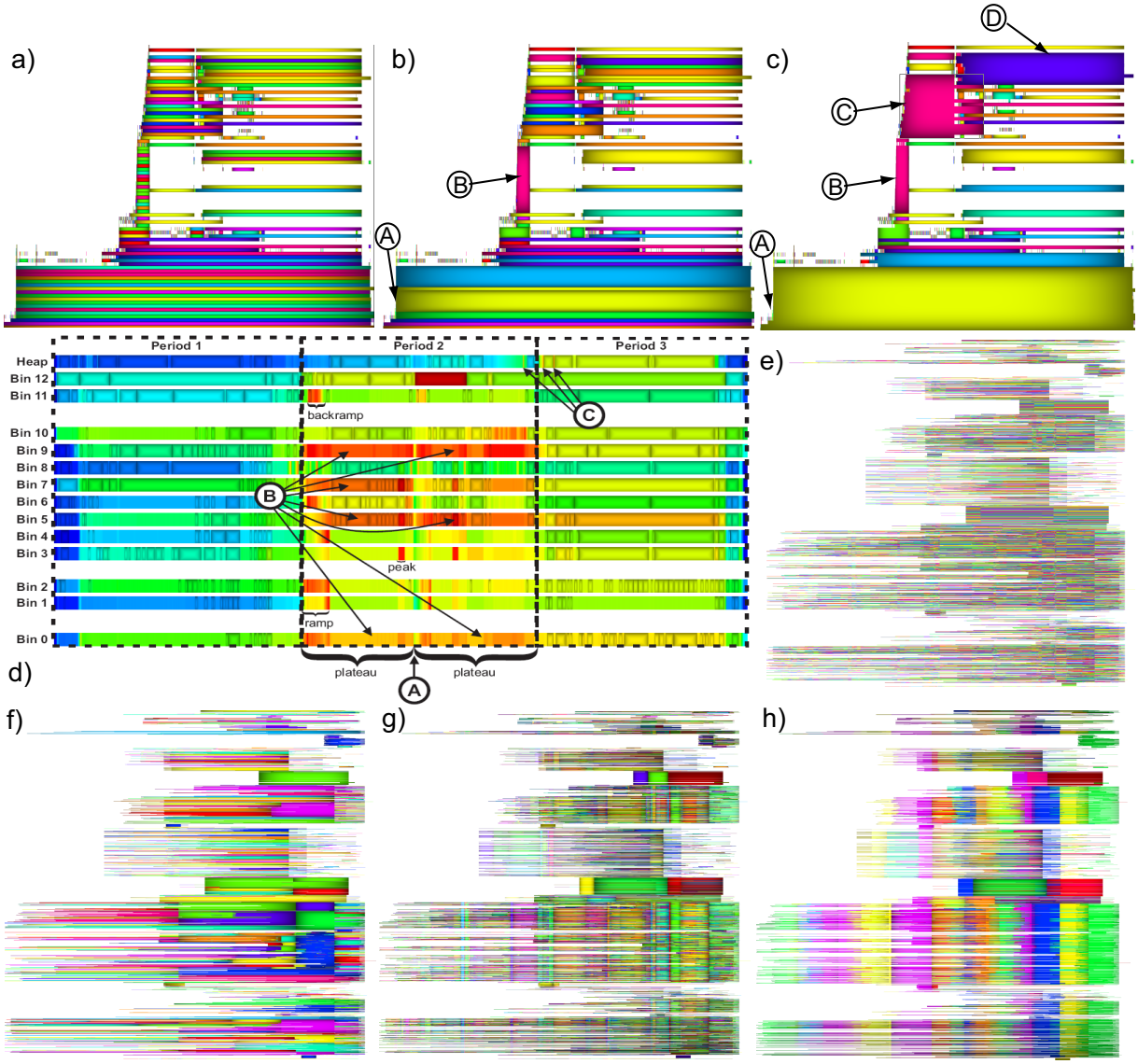

Figure 7: *Hierarchical visualization. Memory log, compact metric (a-c) and occupancy evolution (d). Repository log, compact (e-f) and non-compact (g-h) metrics. Clustered visualizations show same-lifetime memory blocks (c) and software releases (h)*

second. The allocator organizes memory in a *pool*, partitioned into B bins, and an unstructured *heap*. Each bin b_i has a fixed number N_{b_i} of free blocks of equal size $dim_0 < dim_i < dim_B$. A `malloc` request of size $s < dim_B$ is served by allocating a full block in the bin b_i whose block size best fits s. If b_i is full or $s > dim_B$, memory is allocated on the heap. Important quality metrics are waste and fragmentation. *Waste* equals the memory lost because of the fixed block sizes. *Fragmentation* manifests itself by having scattered instead of contiguous free blocks. Typical questions are:

- How does fragmentation depend on time and pool?
- How does waste depend on time and pool?
- Which are the largest quasi-compact regions allocated?
- Are the (de)allocations served in the right order?

Figure 8 shows all $B = 13$ bins and the heap, rendered as explained in Sec. 4, all scaled to the same window size. Color shows per-block waste (blue=none, red=maximal). A red bar right of each view shows the free memory in that pool/heap. The black-framed bars under the views show the occupancy evolution in time (blue=all free, red=all full). We see several interesting facts. Bins 1,9 and 12 have the most per-block waste (warm colors) and bins 4 and 5 the least (cold colors). The heap has zero waste (dark blue), which is indeed correct. Bins 9,11,12 and 13 are the fullest (shortest vertical red bar). All bins begin with little fragmentation (compact blocks at bottom of all bin views), but end up with a higher

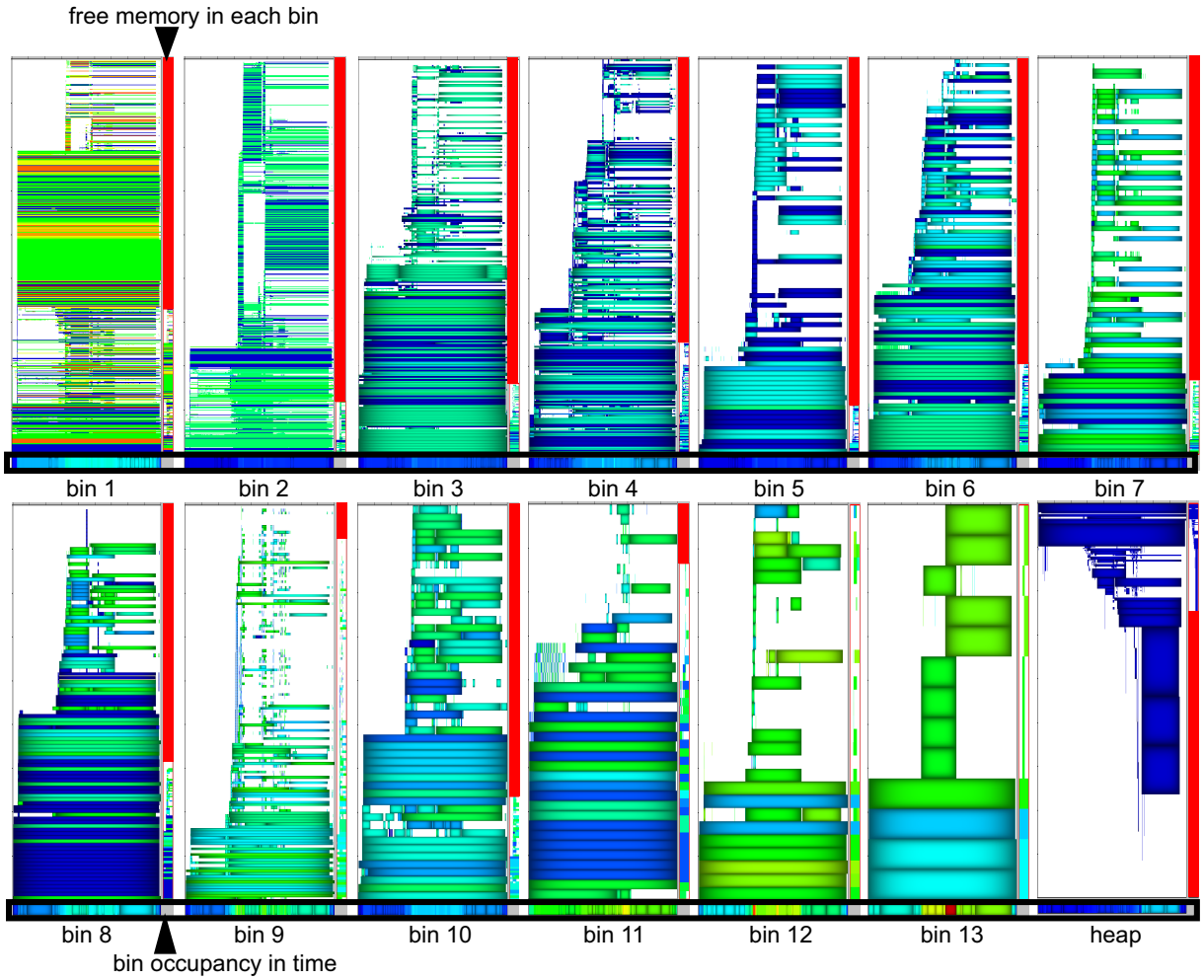

Figure 8: *Bins and heap occupancy visualization for the dynamic memory allocator log*

one (less compact blocks at top of all bin views). Figure 7 d shows a detail view of the occupancy bars. The 'flat shaded', non-cushioned parts of the occupancy bars indicate high-frequency, short-lived, subpixel-size allocations, i.e. a high activity. During the second third of the monitored period, memory occupancy suddenly increases. Yet, an overall occupancy drop (Figure 7 d A) splits the occupancy patterns of bins 0,5,7 and 9 into two near-constant-occupancy 'plateaus' (Figure 7 d) B). In the last third of the monitored period, occupancy decreases. Yet, there are three very short periods where memory occupancy bursts to a maximum in the heap (Figure 7 d C). Using the importance-based antialiasing revealed this dangerous moments which would otherwise have passed undetected. In the heap (Fig. 8 lower-right) blocks get allocated from high to low addresses, conversely than for the bins, which is indeed correct for this allocator. Finally, the heap shows a higher block size variation (cushion height) as compared to all bins. This validates the best-fit allocator policy explained above.

Figure 7 a-c shows a hierarchical clustering with interleaved cushions for the memory log data. Figure 7 a shows the cluster tree leaves, i.e. all allocated blocks, using random colors from a small color set. There is little structure in this image. Figure 7 b shows a same-error section of the cluster tree built with the y-adjacent metric (d_3, Sec. 5.1). We see here some large blocks at the bottom (A) and a vertical strip in the middle (B). Figure 7 c shows a same-error section in the same tree, further simplified. We see here four large, clearly delineated clusters (A-D). These indicate blocks allocated and freed almost at the same time. Details-on-demand by mouse brushing revealed their meaning: A contains global variables, whose lifetime equals that of the whole process; B is a dynamic array of equal-sized elements; C and D hold local function variables.

6.2. Software evolution analysis

In the second application, we analyze the evolution of the VTK code base. Our clustering and interleaved cushions let

us push the analysis further than with tools such as CVSgrab [VT06b]. Figure 7 e shows the unclustered VTK software repository log. Clearly, there is little structure to see. Figure 7 f shows a same-error section of the VTK log clustering using the y-adjacent metric (d_3, Sec. 5.1). We see several large same-color clusters. These are files which were changed together for given periods of time *and* are located in the same directory. For example, cluster A contains the "Python examples" of the VTK code base. Finding such clusters means obtaining an evolution-based system decomposition. However, often files located in different directories may evolve together. To find these, we can use the d_1 or d_2 metrics (Sec. 5.1). Figures 7 g-h show two same-error sections of a clustering using the d_2 metric. The interleaved cushions make several vertical stripes visible. These correspond to aligned (interleaved) cluster borders. The stripes in Fig. 7 h are broader than in Fig. 7 g, as we decrease the level-of-detail. These stripes correspond to stable development periods (when few files change), separated by "releases" (moments when many files change together). We validated this by comparing these moments inferred by our clustering with the actual release moments recorded in the repository as extracted by CVSgrab [VT06b].

This visualization differs in two main respects from CVSgrab's hierarchical visual clustering [VT06b]. Given the artifacts e^i, each having the versions e^i_j, we cluster here the individual *versions* e^j_i, whereas CVSgrab clusters *entire* artifact evolutions e^i. Thus, we can detect which files evolve similarly *and* also during which time period this happens. This is the exact meaning of the cushions in Fig. 7. This is useful, as the CVSgrab users complained in the past that clustering entire evolutions is too restrictive. Indeed, given some files A, B and C, A and B can evolve similarly for a while, after which B evolves similarly with C. Obtaining images such as Fig. 7i, which show system releases as vertical stripes, is not possible with CVSgrab. Stronger, CVSgrab changes the layout to group similar files together, as it can render only rectangular clusters. Our interleaved cushions support clusters with arbitrary borders and even non-compact clusters, thereby keeping the layout unchanged.

7. Acknowledgements

We are grateful to Christian del Rosso, from Nokia Research, for providing us with the case study information and with useful feedback on the results of our visualization.

8. Conclusions

We have presented several new techniques for visualizing dynamic software log data. Our importance-based antialiasing guarantees visibility of rapid (isolated) events rendered as subpixel items in large datasets. We discuss a hierarchical clustering method that reveals structure from "flat" log data, thereby answering questions such as "show array allocations" in memory allocation logs and "show system releases" in code repository logs. We proposed interleaved cushions, a simple but effective way to visualize non-compact clusters on a 2D layout. We illustrated the above on two real-world applications: the monitoring of a dynamic memory allocator and the analysis of a large software repository. We plan to apply our techniques to different domains and also extend them to visualize higher multivariate datasets.

References

[ACS90] ALPERN B., CARTER L., SELKER T.: Visualizing computer memory architectures. In *Proc. IEEE Visualization* (1990), IEEE Press, pp. 107–113.

[Bos01] BOSCH R.: *Using Visualization to Understand the Behavior of Computer Systems*. PhD thesis, Stanford University, 2001.

[BST*00] BOSCH R., STOLTE C., TANG D., GERTH J., ROSENBLUM M., HANRAHAN P.: Rivet: A flexible environment for computer systems visualization. *Computer Graphics 34*, 1 (2000).

[GT89] GRISWOLD R., TOWNSEND R.: The visualization of dynamic memory management in the icon programming language. In *Tech. Report 89-30* (Dec. 1989), Dept. of Comp. Science, Univ. of Arizona.

[JG94] JEFFERY C., GRISWOLD R.: A framework for execution monitoring in icon. *Software - Practice and Experience 24*, 11 (1994), 1025–1049.

[LNVT05] LOMMERSE G., NOSSIN F., VOINEA L., TELEA A.: The visual code navigator: An interactive toolset for source code investigation. In *Proc. InfoVis* (2005), IEEE Press, pp. 4–11.

[Lyn06] LYNUXWORKS: The lynxinsure++ analysis and visualization toolkit, 2006.

[SK93] STASKO J., KRAEMER E.: A methodology for building application-specific visualizations of parallel programs. *J. of Parallel and Distributed Computing 18*, 2 (1993), 258–264.

[Sta92] STASKO J.: Animating algorithms with x-tango. *SIGACT News 23*, 2 (1992), 67–71.

[VT06a] VOINEA L., TELEA A.: Cvsgrab: Mining the history of large software projects. In *Proc. IEEE EuroVis* (2006), pp. 187–194.

[VT06b] VOINEA L., TELEA A.: Multiscale and multivariate visualizations of software evolution. In *Proc. ACM SoftVis* (2006), pp. 47–56.

[vWvdW99] VAN WIJK J., VAN DE WETERING H.: Cushion treemaps: Visualization of hierarchical information. In *Proc. IEEE InfoVis* (1999), IEEE Press, pp. 73–78.

[WKT04] WEIDENDORFER J., KOWARSCHIK M., TRINITIS C.: A tool suite for simulation based analysis of memory access behavior. In *Proc. ICCS* (2004), pp. 440–447.