Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor

Markus Weninger
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
markus.weninger@jku.at

Lukas Makor
CD Laboratory MEVSS
Johannes Kepler University Linz
Linz, Austria
lukas.makor@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
hanspeter.moessenboeck@jku.at

***Abstract*—Tool support is essential to help developers in understanding the memory behavior of complex software systems. Anomalies such as memory leaks can dramatically impact application performance and can even lead to crashes. Unfortunately, most memory analysis tools lack advanced visualizations (especially of the memory evolution over time) that could facilitate developers in analyzing suspicious memory behavior.**

In this paper, we present *Memory Cities*, a technique to visualize an application's *heap memory evolution* over time using the *software city* metaphor. While this metaphor is typically used to visualize static artifacts of a software system such as class hierarchies, we use it to visualize the dynamic memory behavior of an application. In our approach, heap objects can be grouped by multiple properties such as their types or their allocation sites. The resulting object groups are visualized as buildings arranged in districts, where the size of a building corresponds to the number of heap objects or bytes it represents. Continuously updating the city over time creates the immersive feeling of an evolving city. This can be used to detect and analyze memory leaks, i.e., to search for suspicious growth behavior. Memory cities further utilize various visual attributes to ease this task. For example, they highlight strongly growing buildings using color, while making less suspicious buildings semi-transparent.

We implemented memory cities as a standalone application developed in Unity, with a JSON-based interface to ensure easy data import from external tools. We show how memory cites can use data provided by AntTracks, a trace-based memory monitoring tool, and present case studies on different applications to demonstrate the tool's applicability and feasibility.

***Index Terms*—Memory City, Software City, Software Map, Visualization Metaphor, Heap Memory Evolution, Memory Leak Analysis, 3D Visualization, Interactive Analysis System**

I. Introduction

Modern programming languages such as Java use automatic garbage collection to free the developer from the error-prone task of allocating and freeing memory manually. To do so, heap objects that are no longer reachable from static fields or thread-local variables (so-called *GC roots*) are automatically reclaimed by a garbage collector (GC). Nevertheless, memory problems and anomalies such as memory leaks can still occur even in garbage-collected languages. For example, memory leaks happen if objects that are no longer needed remain reachable from GC roots. A common cause for this is that a developer accidentally forgets to remove objects from a (long-living) container data structure [2]–[4]. Such objects cannot be reclaimed by the garbage collector and will therefore accumulate over time.

To inspect a memory leak, users have to search for groups of objects that grow suspiciously over time. To perform such inspections, memory monitoring tools such as VisualVM [5] or Eclipse MAT [6] are often used. Unfortunately, many of these state-of-the-art tools do not use graphical means (except for time-series charts) to visualize the evolution of the heap. Yet, the usefulness of advanced visualization techniques in domains such as software evolution and program comprehension has already been shown in various user studies [7]–[14]. Thus, we think that developers can also profit from software visualizations in the domain of memory monitoring.

Visualizations often rely on metaphors to serve as "*a mapping from the concepts or artefacts required to be displayed in the virtual world to their graphical representation*" [15]. For example, in their inspiring works, Knight and Munro [15], [16] suggest the *software world* metaphor which consists of countries, cities, districts, and even details such as gardens and interior. Since this level of detail may not always be suitable or needed, the *software cities* metaphor emerged. Wettel and Lanza [17], [18] used software cities to visualize the static structure of a software system, where *buildings* represent classes, grouped into *districts* based on their packages. The size of a building is determined by the classes' number of attributes and number of methods. They extended this approach to also visualize the evolution of the code base over time [19]–[21]. Subsequent software city approaches used this metaphor to visualize the *dynamic* behavior of a software system based on recorded traces, such as *SynchroVis* to visualize concurrency [22] and *ExplorViz* to visualize the communication and dependencies between software components [11], [23]–[26].

Inspired by the widespread use of the software city metaphor, we combined existing techniques with new ideas to apply this metaphor to the domain of memory monitoring. In this paper, we present *Memory Cities*, an approach to visualize the heap evolution as an evolving software city. In memory cities, buildings represent heap object groups that are arranged in districts based on shared heap object properties such as type. The size of the buildings can change

The Memory Cities artifact (including binaries, data sets, video, and instructions) is available at [1], a video of the tool can be found at http://ssw.jku.at/General/Staff/Weninger/AntTracks/VISSOFT20/MemoryCities.mp4

DOI 10.1109/VISSOFT51673.2020.00017

over time, representing growing and shrinking heap object groups throughout the lifetime of a monitored application. Our goal is to ease the inspection and comprehension of memory growth over time, a common task in memory leak analysis, by providing an interactive and easy-to-understand visualization. Based on a work-in-progress report [27], this paper discusses the full *visualization pipeline* [28], [29] of memory cities, see Section III. In detail, our scientific contributions encompass:

- a data model based on which memory cities can be generated (Section IV),
- a discussion of the layout algorithm used (Section V),
- a mapping from memory metrics to visual attributes (Section VI),
- interaction features such as time traveling, information retrieval and a novel heap object reference analysis in 3D memory cities (Section VII),
- a fully functional 3D memory city visualization tool,
- various memory city case studies to showcase the approach's feasibility and applicability (Section VIII).

II. Background

Our memory city visualization has a well-defined JSON interface to be independent of a specific data source. Yet, to make this work more tangible, we regularly refer to data imported from the memory monitoring tool AntTracks[1]. Thus, this section presents the basics of AntTracks, a fully functional trace-based memory monitoring tool consisting of the *AntTracks VM* [30]–[32] and the *AntTracks Analyzer* [3], [4], [33]–[38].

A. Trace Recording by the AntTracks VM

The AntTracks VM (a slightly modified Java VM) records events such as object allocations and object movements performed by the GC during garbage collection and writes them into a trace file [30]. Additionally, the VM collects information about garbage collection roots and the references between objects [32], [36]. To reduce the trace size, the VM does not record any redundant data and applies compression [31].

B. Reconstruction in the AntTracks Analyzer

The AntTracks Analyzer incrementally processes the events in a trace file, reconstructing the heap state, i.e., the set of objects that were live in the monitored application, at every garbage collection point [33]. For every heap object, various properties can be reconstructed, including its memory address, type, allocation site, allocating thread, GC roots, the heap objects it references, and the heap objects it is referenced by.

The tool's core mechanism is object classification and multi-level grouping [34], [35]. A classifier groups heap objects according to certain criteria such as type or allocation site. Grouping the heap objects based on multiple classifiers results in a hierarchical *grouping tree*. A common classifier combination is to group all heap objects by their types and then by their allocation sites, as shown in Figure 1. Yellow rectangles represent tree nodes and blue circles represent the objects that were classified into the respective tree branch. For example, the objects 0 to 3 are of type `Object[]`, of which the objects 0, 1 and 3 have been allocated in `Stack:init()` and object 2 has been allocated in `MyService:foo()`.

[1]AntTracks available at: http://mevss.jku.at/AntTracks

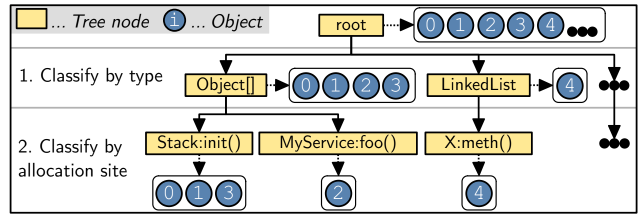

Fig. 1. A *classification tree* that first groups all heap objects by their types and then by their allocation sites.

C. Common Techniques for Growth Visualization

As heap objects can be grouped by their properties, resulting in a grouping tree, it is common to display such data in a tree table, similar to the one shown in Table I.

TABLE I
A TREE TABLE VIEW REPRESENTING A HEAP STATE GROUPED BY TYPES AND ALLOCATION SITES.

	Objects
- Heap	100,000
- Type A	80,000
Allocated in foo()	70,000
Allocated in bar()	10,000
+ Type B	10,500
...	

Some tools also provide features to visualize the differences between two points in time. For this, they typically (1) take a heap snapshot at two points in time, (2) group the heap objects in both snapshots according to the same criteria, (3) calculate the differences of the number of objects for every tree node, and then (4) display these differences in a tree table. Even though object groups that grew between two points in time may hint at a memory problem, comparing two snapshots does not reveal general trends in an application's memory behavior. To detect trends, the heap has to be compared at multiple points in time, a feature that is not supported by most state-of-the-art tools.

III. Approach

Our memory cities approach tackles the problem stated at the end of Section II-C: It aims to provide an intuitive and immersive visualization to inspect an application's *heap evolution over time*. This section discusses the approach in general and presents its most important features and steps (shown in Figure 2) that also serve as an outline for the rest of the paper.

A. Overview

In general, a memory city displays a grouping tree, i.e., grouped sets of heap objects, as a 3D city visualization. Such a

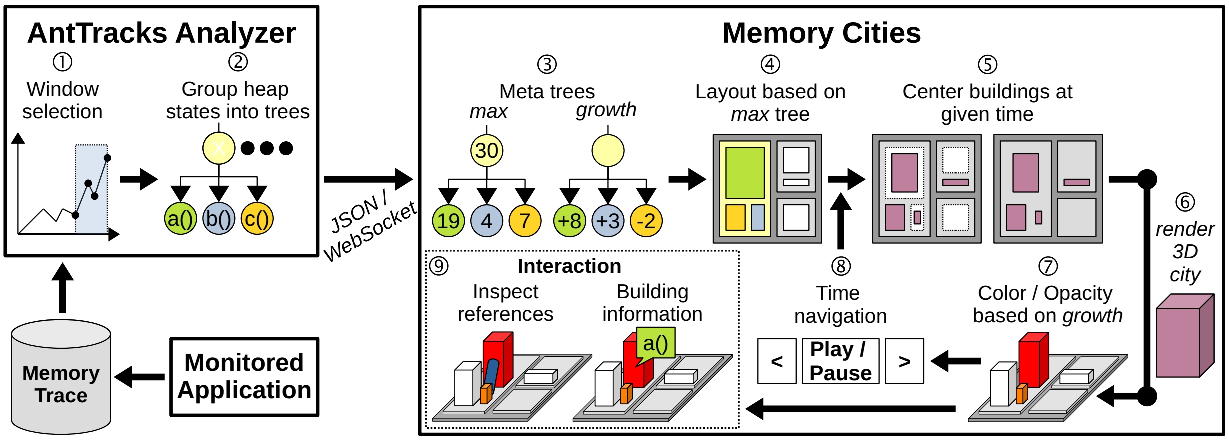

Fig. 2. Overview of our memory cities approach, corresponding to the typical visualization pipeline steps preprocessing, filtering, mapping, and rendering [29].

city consists of two types of structures: *buildings* and *districts*. Buildings represent tree *leaf* nodes, where a building's area and height is determined by the number of objects / bytes represented by the respective tree node. For example, if the heap objects have been grouped by their types and allocation sites, each building represents a set of heap objects of the same type that have been allocated in the same method. These buildings are then grouped into districts based on their parent tree nodes, where districts can again be grouped into other districts. An example of such a city can be seen in Figure 3.

To generate such a layout, a tree map algorithm [39], [40] can be used. Figure 4 shows a tree map example, in which the orange parent node (district) represents 40MB of `Person` objects, with two yellow leaf nodes (buildings) representing 30MB allocated in method $m2$ and 10MB allocated in method $m1$. As the set of classifiers that is used to group the heap objects is user-defined in AntTracks, various memory cities for different analysis purposes can be created. For example, if the user is interested in the most frequent types of objects allocated per thread, one could first group the heap objects by their allocating threads (districts) followed by their types (buildings). Memory cities can not only be inspected at a single point in time, but the user can step back and forth in time. This creates the feeling of an evolving city and enables users to search for strongly growing buildings, i.e., heap object groups that may be part of a memory leak. This task is further supported by the use of color highlighting and opacity settings.

B. Steps

Figure 2 presents the steps that lead from a recorded memory trace to the final memory city. The following list shortly describes each of them; they are explained in more detail throughout the rest of this work.

① Once a memory trace file has been loaded by the AntTracks Analyzer, the user sees the total heap memory utilization over time in a time-series chart. In this chart, the user can then select a suspicious time window, which may also be automatically suggested by AntTracks [38].

② Within the selected time window, the heap is grouped at every garbage collection point according to a user-defined set of heap object properties, resulting in a list of grouping trees.

③ Based on these grouping trees, various *meta trees* are calculated. For example, a *max tree* stores the maximum number of objects and bytes a tree node represents at any point in time (in other words, the largest size a district or building may reach), while a *growth tree* stores the growth of each node between the first and the last grouping tree.

④ To reserve space for every building that will eventually be displayed in the city, we use the object/byte counts stored in the *max tree* as an input for the *squarified tree map* algorithm [41] to generate the city's general layout *once*. By doing so, the generated layout ensures that every building could fit into the city even if all of them reached their largest size at the same point in time.

⑤ To display a memory city at a certain point in time, each building's base area is calculated at that point and the building is then centered in the layout spot reserved for it.

⑥ Once every building has received its location, the building's height is calculated and the corresponding cuboid is placed in the 3D environment.

⑦ To ease the search for growing structures, we use the growth information (stored in the *growth tree*) to highlight certain buildings using color and opacity.

⑧ The user can step back and forth through time to visualize the evolution of the city. When the user navigates between points in time, steps ⑤ to ⑦ are executed for each new point, and the visualization is updated. It is also possible to run this animation automatically to watch the whole city evolution without any user interaction needed.

⑨ Moving through time is not the only interaction possibility in memory cities. Users can also gather more information about a structure (i.e., a building or district) by hovering or clicking it. Another feature is to show references between two buildings. In the case of a memory leak, this feature helps users to differentiate between buildings, i.e., object groups, that cause other buildings to grow and those that grow because their object's are kept alive by others.

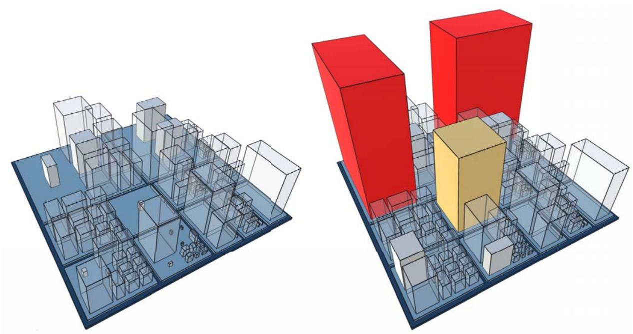

Fig. 3. An application's heap visualized with memory cities shortly after startup (left) and 2 minutes / 300 garbage collections later (right). Districts are colored blue-ish based on their hierarchy level, buildings are colored from gray to red based on their growth. The ten buildings with the strongest growth are shown in solid mode, while the others have reduced opacity.

IV. Data

This section discusses in more detail which data is needed by software cities in general, how this need translates to memory cities, and how we collect and process the needed data using AntTracks.

A. General

In general, a software city is built upon tree data. In its most basic form, each tree node contains a *key* for identification and at least one *value* based on which the city is laid out. Nevertheless, limiting each tree node to a single value also massively limits the number of visual attributes a software city can make use of. For example, a single value can be represented by the size of a building, with no other attributes such as color that could convey further information. If each tree node contained three values, one of them could be used to calculate a building's base area, one could be used to calculate the building's height, and one could be used to determine the building's color, providing much more information for more diverse inspections. Using more visual attributes can make the visualization richer, yet complex mappings should be used for complex tasks or expert systems only since the mappings may become challenging to perceive [29]. Thus, when designing a new software city for a certain task (such as memory cities for the task of heap memory evolution analysis), the designers first have to decide whether they want to develop an expert system or a system that is also usable by novices.

B. Memory Cities

Since many expert memory monitoring tools already exist, our focus is to make memory anomaly inspection easier for novice users [42]. To achieve this, the goal of memory cities is to provide enough details to enable the detection of memory anomalies such as memory leaks, while keeping the visualization simple enough to understand it without prior training or explanations.

Once this decision is made, the next step is to define which data is needed. In general, a memory city is based on a tree in which each node represents a group of heap objects. As already discussed in Section II, a grouping tree can be constructed in AntTracks by applying a user-defined set of classifiers on the heap to group its objects accordingly. There are two ways to aggregate the heap objects in each node: Either by counting the number of objects, or by counting the number of bytes that the respective objects take up in the heap. We decided that for every tree node both metrics should be available for visualization in the memory city. Thus, our memory city tool expects the following data in each tree node:

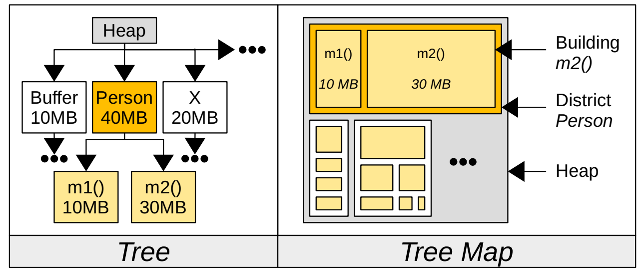

Fig. 4. We use tree mapping to lay out the buildings in memory cities.

- A unique *key* to identify the object group (e.g., "`Heap#Person#m1`").
- A *name* to display (e.g., "`m1()`").
- A *role* that specifies the object group's grouping criteria (e.g., "Allocation Site").
- An *object count* value.
- A *byte count* value.
- A *list of child nodes* which is empty for leaf nodes.

Since the heap grows and shrinks over time while an application is running (as new objects are allocated and others are freed by the GC), a major goal is to visualize this memory evolution. Especially, memory cities should help users to detect object groups that grow suspiciously strong, as this behavior hints at memory leaks. To this end, a memory city may not only load a single tree, but also a list of trees (representing heap states at garbage collection points), where each tree has a timestamp to ensure correct ordering.

Once such a list of trees has been imported, the memory city calculates two *meta trees* that are used to lay out the city and to highlight buildings: The *max tree* stores the maximum number of objects and bytes a tree node represents at any point in time (in other words, the largest size a district or building may reach), while a *growth tree* stores the growth of each node between the first and the last grouping tree.

Our memory city visualization has explicitly been developed to not depend on AntTracks' grouping trees or any internals of AntTracks. To achieve this, we provide two ways of how to import data into our memory cities tool: Either by loading a list of grouping trees in JSON format[2] from disk, or by sending a list of grouping trees in JSON format to the memory cities tool via a WebSocket. Thus, any other memory monitoring tool besides AntTracks could also use our memory cities tool

[2] JSON format example for list of grouping trees: http://ssw.jku.at/General/Staff/Weninger/AntTracks/VISSOFT20/MemoryCities.json

to visualize heap states and heap evolution, as long as the tool can provide a list of trees with the previously mentioned information per tree node.

V. Layout

In the software city metaphor, artifacts are visualized as buildings that are arranged in districts, which can again be contained in other districts. In this section, we present how memory cities are laid out using the *squarified tree map algorithm* [41] and how we apply *static position animation* [43] to achieve a stable layout over multiple points in time.

A. Single Tree

In general, tree maps implicitly visualize a tree's hierarchy via containment, i.e., the tree is visualized as a rectangle that contains other rectangles, which again can contain rectangles and so on. Thus, each rectangle represents a tree node, and the rectangle's area is determined by one of the tree node's values. In case of memory cities, this value is either the node's object count or byte count. Instead of using the value directly (i.e., an increase of objects/bytes by a factor of 2 results in a building with a base area twice as big), a mapping function such as `sqrt` can be applied on the values beforehand.

To generate a tree map, we use a recursive algorithm that is given a tree node and a rectangle, which is then divided to fit the tree node's child nodes [39], [40]. The alignment and rectangle ratio vary between different tree mapping algorithms [44]. We use the *squarified tree map* algorithm by Bruls et al. [41], which tries to shape the area of each tree node as an approximate square. This creates more realistic cities than using elongated shapes. The resulting layout is then used to generate the 3D city visualization by displaying leaf nodes as buildings and inner nodes as flat districts, which will be explained in more detail in Section VI.

B. Evolution Over Time

The visualization of the heap's evolution over time, i.e., visualizing multiple trees one after another to inspect their growth, needs special handling, as it is *not* enough to perform a simple tree map layout whenever switching from one tree to another. One of the reasons for this is that if tree nodes were added or removed between two points in time, the respective rectangles in the tree map layout also have to be added or removed. This may happen if all objects of a certain type were collected by the GC, which leads to the disappearance of the respective tree node. Such a change in the tree structure would result in a change of the overall arrangement of districts and/or buildings. An unstable layout may cause users to lose track of a certain building. It becomes hard to figure out if and which two buildings in different heap states represent the same tree node, a core requirement for visualizations that want to visually express a system's growth.

We apply *static position animation* [43] to overcome the problem of an unstable layout. This technique creates a general city plan in which all buildings and districts remain at the same position at every point in time. To create this general city plan, we use the *max tree* presented in Section IV-B as an input to the tree map algorithm. Every node in this meta tree stores the maximum number of objects / bytes represented by the respective node at any time. This layout is calculated *once* when the memory city is initialized and contains a rectangle for every district and every building that will eventually be shown. More specifically, it reserves space for every district and building based on its *largest possible area*. Then, to visualize the heap at a certain point in time, buildings are centered in the space that has been reserved for them.

C. Tree Pruning

During the layout phase, it is also possible to prune the tree to reduce the complexity of the resulting memory city. For example, the tree map algorithm could be restricted to only take into account the N largest child nodes per parent, thus only reserving space for those buildings that represent larger groups of heap objects. When the city is shown for a certain point in time and no reserved space is found for a given tree node, this means that the object group is not relevant enough for the visualization and no building is shown for that node.

This feature is particularly useful for very wide trees. For example, grouping the live objects of a real-world application by type (e.g., `String`, `HashMap`, etc.) may result in hundreds or thousands of tree nodes, many of which may only represent a few objects [36]. Since one of the main goals of memory cities is to support the visualization of memory leaks, i.e., object groups that accumulate a large number of objects over time, small object groups are not of interest to the user and can be dropped. By default, memory cities have tree pruning enabled, using a user-defined number of child nodes to be shown.

VI. Metrics and Visual Mapping

As discussed in the previous section, the area of a building in a memory city depends either on the number of objects or the number of bytes its tree node represents. Yet, memory cities also use a number of other visual attributes to convey information to the user. This section discusses these attributes.

A. Districts

Similar to other software cities, districts in memory cities are flat structures, i.e., their height is fixed and does not encode information. Their purpose is to visualize the hierarchy of the underlying grouping tree. Thus, the bottom-most district always represents the whole heap, which may be divided into (multiple levels of) districts, one for each inner node in the underlying grouping tree. We use a linear *color gradient* from dark blue to light blue to encode a district's level.

B. Buildings

As shown in Figure 5, in addition to the area metric (which is either based on the object count or the byte count a building represents) we further utilize the visual attributes height, color and opacity for each building. Each of these attributes will be discussed in the following.

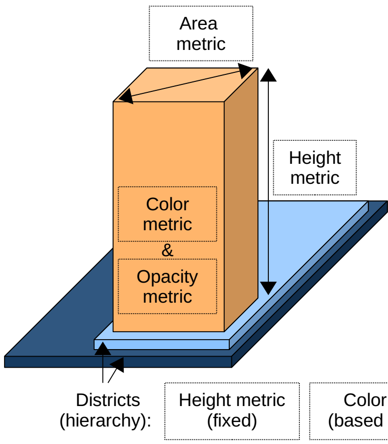

Fig. 5. Various visual attributes are used to express metrics.

1) Height: One of our goals was to achieve building sizes that represent more-or-less realistic measures of real-world buildings. Thus, for a building with an area of A square units, we use $2 * \sqrt{A}$ units as its height. This results in buildings that, if visualized with a perfectly squarified foundation, have a height twice the size of the building's side length. Mapping units to meters, for example, a building with an area of 100 square meters (squarified side length of 10 meters) would have a height of 20 meters, while a building with an area of 400 square meters would have a height of 40 meters. Calculating the height based on the area means that both visual attributes represent the same metric, either object count or byte count. Mixing these metrics, i.e., using one metric for the area and the other one for the height, is still up to future research, since doing so did not yield satisfying results so far. For example, having a node that represents few very large arrays could result in (a) extremely narrow buildings that are quite tall (if the object count was used for the area and the byte count was used for the height) or (b) extremely wide buildings that are quite flat (if the byte count was used for the area and the object count was used for the height). Such unrealistic buildings would distort a realistic city feeling and would also be hard to interact with in certain situations (e.g., narrow tall buildings are hard to see and click). A possible solution in future work could be to use *categorical data* for the height, e.g., mapping the byte count to a few fixed heights such as tiny, small, medium, large and huge.

2) Color: Memory cities try to support users in understanding memory evolution (especially memory growth) over time. To make this task easier, memory cities encode the hitherto growth of a building as color. To this end, we utilize the *linear color gradient* shown in Figure 6.

Fig. 6. The color gradient used for buildings, ranging from gray (shrinking / no growth) over orange (medium growth) to red (strong growth).

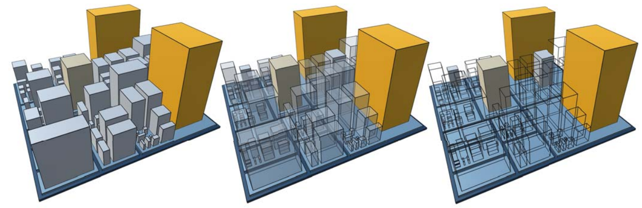

Fig. 7. Three different city representations. Left: Every building fully opaque. Middle: Five strongest growing buildings fully opaque, rest 40% opaque. Right: Five strongest growing buildings fully opaque, rest fully transparent.

The gradient maps a value in the range $[0, 1]$ to its respective color. Given a certain tree node with the identifier key, the access functions $first(key)$ and $cur(key)$ to query the node's value (either objects or bytes) at the first point in time and at the current point in time, respectively, and the function $max()$ that returns the largest growth of any building stored in the *growth tree*. The color can then be calculated using $gradient((cur(key) - first(key))/max())$. Negative values are mapped to gray and represent buildings that shrank.

3) Opacity: To further increase the user's focus on strongly growing object groups, the opacity of less important buildings can be decreased. The *growth tree* contains information about the growth between the first and the last point in time, i.e., the overall growth. Since object groups, i.e., buildings, that grew the strongest over the selected time window are those which are most likely involved in a potential memory leak, it seems reasonable to highlight those buildings and damp the others. Thus, memory cities allow the user to turn on the *building opacity* mode and select a number of N buildings that should stay opaque. As shown in Figure 7, the N buildings with the strongest growth (queried from the growth tree) stay fully opaque, while all other buildings are drawn at a user-defined reduced level of opaqueness (by default 40%). It is worth mentioning that the metric on which this visual attribute is based, namely the *overall* growth over the whole time window, differs from the metric used to define the building's color, namely the *relative* growth since the start of the time window up to the current point in time. It is thus possible for a building to appear red and transparent at some point in time, i.e., strong growth up to that point but no strong overall growth, if the building shrinks again afterwards. Consequently, at the last point in time, those buildings that are shown opaque also have the most intense red color.

VII. Interaction

Users can navigate the camera through a memory city, they can step back and forth in time, they can click and hover structures to inspect them in detail, and they can display the number of references between buildings, i.e., heap objects. All of these features are explained in more detail in the following.

A. Navigation

The camera can be tilted, rotated and zoomed using the mouse wheel. By dragging the mouse or using the keyboard,

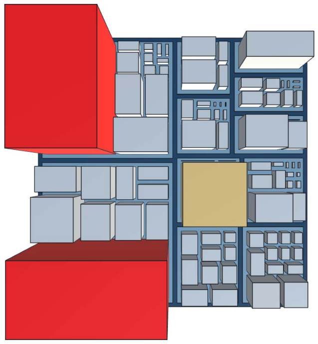

Fig. 8. The keyboard shortcut `B` positions the camera into a bird's eye view.

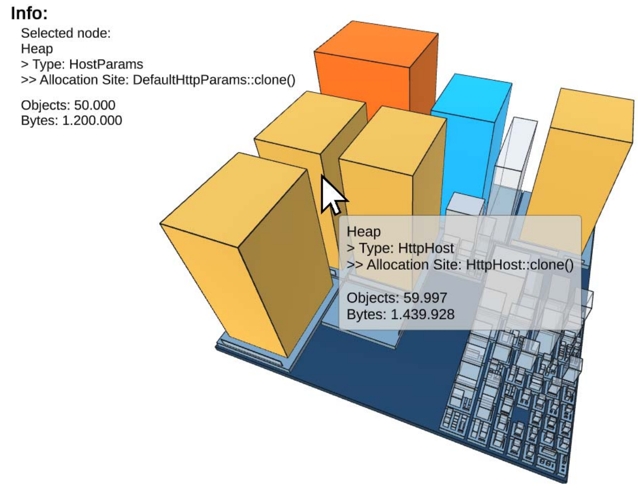

Fig. 9. Information about a structure is shown when hovering it (gray tool tip) or when selecting it with a click (selected building is highlighted in blue).

the user can move the camera. Memory cities also provide keyboard shortcuts for typical tasks. For example, pressing the `B` key moves the camera into a bird's eye view (see Figure 8), which can be useful to inspect the district structure.

B. Evolution Visualization: Time Travel

To visualize the memory evolution over time, we apply time traveling. Wettel and Lanza [19] define time traveling in the context of software cities as stepping back and forth through the history of a system while the city updates itself to reflect the current state. In our case, the history of the system is the sequence of grouping trees. The time stepping can be performed manually using buttons or a slider as well as using the arrow keys on the keyboard. Additionally, the evolution can also be animated automatically. During this animation, every heap state is shown for a user-defined period of time (0.5 seconds by default) before automatically switching to the next one. Users can pause and restart the animation at any point in time.

C. Structure Information

Hovering over a building or district displays information about its respective heap object group. This information includes the path from the tree root, e.g., *Heap* $\rightarrow$ *Type: Person* $\rightarrow$ *Allocation Site: foo()*, the number of objects and the number of bytes, as shown in Figure 9.

Besides showing a structure's information on hover, users can also click on a structure to highlight it, which is also shown in Figure 9. The structure stays selected when moving through time to make it easier to track its evolution.

D. Heap Object References

A novel feature of memory cities is the visualization of heap object references in a 3D environment. This feature is especially useful to reveal the root cause of a memory leak, since objects may accumulate over time even if they are not directly kept alive by a GC root, but rather indirectly by other objects, which would be the actual root cause of the problem. To fix such a leak, we want to find out the root cause by inspecting the references between the heap objects.

For example, imagine a memory leak caused by a `LinkedList<Person>` where persons are only added but never removed. Further imagine that every person has a first name and a last name, each stored as a `String` field. Every addition to this list will result in six heap objects to be created: One `LinkedList$Node` that references the `Person` which in turn references two `String` objects which again reference a `char[]` each. (1) in Figure 10 shows how such an application's memory city could look like if we group all heap objects by package (districts) and type (buildings). Since the application allocates more `String` and `char[]` objects than `Person` and `LinkedList$Node` objects, these two buildings are colored more intensively, even though they are only a *symptom* of the memory leak and not the *root cause*. To find the real root cause of the memory leak, we can inspect the references between the buildings. (2) indicates that nearly all `char[]` instances are referenced by `String` objects (indicated by a thick purple frustum between the buildings). (3) shows the state of the memory city after selecting the `String` building. We can see that a lot of different types reference strings, but the most references come from `Person`, which is selected in (4). All persons are referenced by `LinkedList$Node` objects. (5) contains a very thin purple frustum which tells us that one of the nodes (i.e., the list head) is kept alive by the `LinkedList`.

To create the reference visualization, we utilize two maps, i.e., a *points-to map* and a *pointed-from map*, as shown in Table II. These maps (that, similarly to the grouping trees, can be imported as JSON files or via WebSockets) contain an entry for every building. For each building, they store how many objects the respective building references in other buildings, or by how many objects of other buildings it is referenced by, respectively. Based on these numbers and the building size itself, the frustums between the buildings can be sized, i.e., the more are objects involved, the bigger the visualized frustum. Since we know for every reference between two buildings how many objects are referencing and how many are referenced,

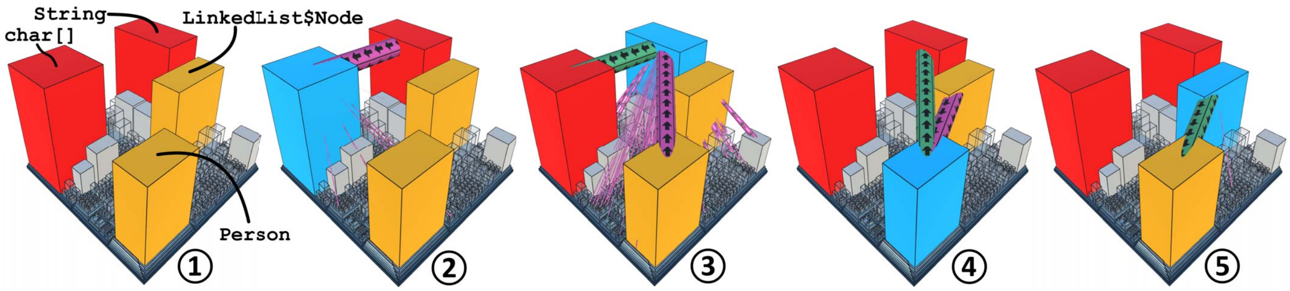

Fig. 10. Heap object reference analysis. The currently selected building in each step is highlighted in blue. Incoming references, i.e., references that keep objects alive in the selected building are colored purple. Outgoing references, i.e., references to objects that are kept alive by objects in the selected building are colored green. The more references there are between the objects of two buildings, the bigger the respective frustum.

TABLE II
A POINTED-FROM MAP AND A POINTS-TO MAP ARE USED AS DATA SOURCE TO CREATE THE REFERENCES BETWEEN BUILDINGS

Pointed-From Map		
String	Person	10,000
	Buffer	300
	...	
Person	LinkedList$Node	10,000
...		

Points-To Map		
LinkedList	LinkedList$Node	1
LinkedList$Node	Person	10,000
Person	String	20,000
String	char[]	20,000
...		

we can even scale the start and the end radius of the frustum differently. For example, if 1% of the objects in building A reference 80% of objects in building B, the radius of the frustum attached to A will be much smaller than the radius at B, indicating a *few-to-many* reference. This information can especially be useful to detect (few) arrays that reference a lot of other objects. Vice versa, this technique can also indicate a *many-to-few* reference behavior, i.e., many objects share few other objects. Currently, a reference between two buildings is shown as a straight color-textured frustum, which might cut through other buildings in its way. Future research includes the evaluation of different reference placement techniques, for example pipe routing [45] or hierarchical edge bundling [46].

VIII. Case Studies

To explain how memory cities can be used and to argue their usefulness and applicability, we present two case studies in which we use them to investigate memory leaks. To this end, we searched for real-world applications that contain memory leaks. In the following, we present the analysis of a memory leak in the *Commons HttpClient* library, as well as the analysis of a memory leak in the *Dynatrace easyTravel* application.

A. Commons HttpClient

Finding applications or libraries that contain memory leaks requires lots of effort, since their source code and the needed build tools have to be publicly available. To find the memory leaking library discussed in this section, we browsed Apache's issue tracker[3] for the keyword *leak*. This way, we found an old issue regarding a memory leak in the *Commons HttpClient* library, a library that can be used to send HTTP requests. As we did not know the library beforehand, it seemed like a good example to check if memory cities are helpful to detect proliferating objects even in an unknown application. We downloaded the affected version 3.0.1[4] and built a small driver application[5] which creates HTTP connections in multiple batches. In each batch, $10,000$ connections are created and deleted shortly thereafter. One would expect to see spikes in the memory usage, as it should go up when connections are created and should go down after their deletion.

Contrary to this assumption, AntTracks reported a continuous memory growth in the application. Thus, we decided to inspect the heap evolution using memory cities. To do so, we selected the *type* and *allocation site* classifiers to be used at multiple GC points to generate grouping trees, which were then imported into the memory cities tool. The left half of Figure 11 shows the evolution of the resulting city over time. As we can clearly see in the third picture, six buildings grew strongly. Inspecting their type names and allocation sites, i.e., the methods in which the object have been allocated, already revealed interesting insights. In addition to that, the right-hand side of Figure 11 shows the reference patterns we observed. This made it clear that the leak has to do with `HostConnectionPool` objects that are kept alive (purple frustum) by `HashMap$Node` (i.e., the nodes of a `HashMap`).

Knowing this reference pattern and the name of the method in which the accumulating `HostConnectionPool` objects are allocated provides us enough information to investigate the problem on the source code level. In the allocating method, we find that the `HostConnectionPool` objects are added to a map upon the creation of a new HTTP creation. However, they are not removed from that map when the connection is deleted, resulting in a memory leak.

[3]Apache's issue tracker for HttpClient: https://issues.apache.org/jira/projects/HTTPCLIENT/issues
[4]Commons HttpClient in version 3.0.1: https://mvnrepository.com/artifact/commons-httpclient/commons-httpclient/3.0.1
[5]Driver application: https://github.com/NeonMika/httpclient-leak-driver

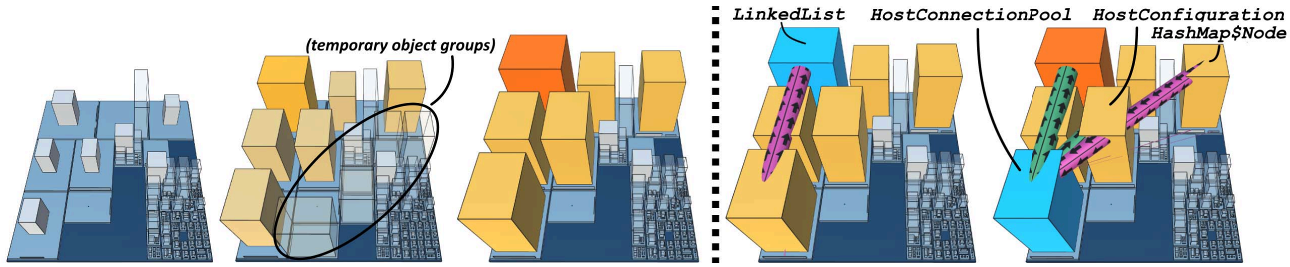

Fig. 11. In the Commons HttpClient application, `HostConnectionPools` (that reference `HostConfigurations` and `LinkedLists`) are kept alive because they are added to a `HashMap` but never removed.

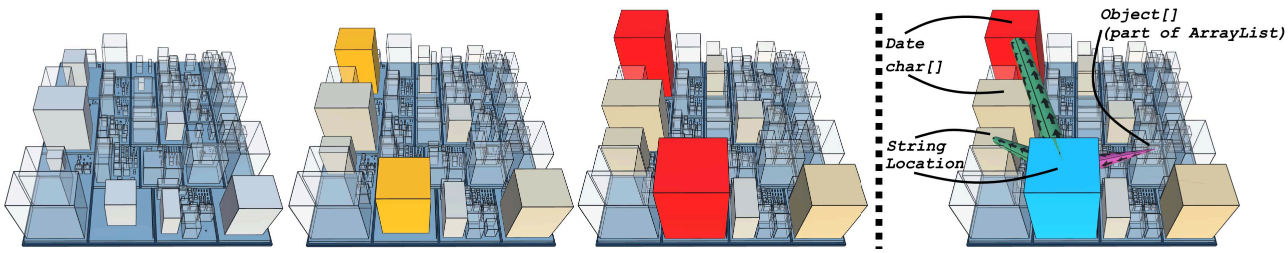

Fig. 12. In easyTravel, `Location` objects accumulate over time, together with many `Date` objects and a few `String` objects that they reference.

B. *easyTravel*

The second investigated application is *Dynatrace easy-Travel*. Dynatrace focuses on application performance monitoring (APM) and distributes easyTravel as their state-of-the-art demo application. It is a multi-tier travel agency application, using a Java backend. A built-in load generator can simulate accesses to the service. When easyTravel is started, different problem patterns can be enabled and disabled, one of which is a hidden memory leak somewhere in the backend.

To inspect the heap evolution over time, we grouped all heap objects by *type* and *closest domain call site*, i.e., the method within easyTravel that led to the allocation even if the allocation itself was hidden inside a third-party framework. Figure 12 depicts the resulting memory city as it evolves over time. The two buildings that are clearly visible as strongest contributors to the heap growth represent `Location` and `Date` objects, each allocated by a certain method. To inspect if this parallel growth is coincidental or caused by either of the two, we inspected their references, as shown on the right-hand side of Figure 12. This makes it clear that the `Locations` reference the `Date` objects, as well as some `Strings`.

Using this information, we inspected the problem on the source code level. We found that the method in which all `Location` object are allocated is only called by the method `findLocations` in class `JourneyService`. There, we found a map that should have served as a cache for location searches. Once a search has been executed, a `QueryKey` instance is created and stored in the map, together with a list of the `Location` objects (the backbone of these lists can also be seen connected to the `Location` building via a purple frustum in the last picture of Figure 12). Subsequent searches for the same key should have found the respective entry in the map. However, `QueryKey` neither implements `hashCode` nor `equals`. Thus, every request (even for an already existing key) resulted in a cache miss, which led to this typical memory leak.

IX. Related Work

In this section, we discuss the use of visualization metaphors in general, as well as the application of the software city metaphor in various domains.

A. *Using Visualization Metaphors*

The use of metaphors in information visualization is widespread and has a long history. In general, metaphors such as *more is bigger* (e.g., bigger visual artifacts represent more of the underlying objects) or *similarity is closeness* (e.g., similar objects are positioned more closely to each other) often unconsciously shape the way we think and act [47]. In the following, we present a few examples of visualization that explicitly state the use of metaphors. For example, Waguespack [48] used geometrical figures as a metaphor for coding constructs to teach programming concepts. Boyle and Gray [49] used 3D structures to visualize database query results, using attributes such as size and position to convey information. More immersive and advanced usages of metaphors include colored virtual reality tunnels for program analysis and comprehension of concurrent programs [50], [51], or interactive map-like interfaces to visualize academic research fields and their similarity to each other [52].

B. *Software Cities and Related Metaphors*

As explained in Section I, Knight and Munro [15], [16] promoted the use of metaphors for software visualizations, especially their metaphor of a *software world*. As an alternative to software worlds, 3D city visualizations emerged. While early 3D city visualization contained a lot of details and sophisticated layouts [53], most modern software cities are based on tree maps that have been extended to three dimensions [54]. New stable tree map algorithms [55], [56] may improve the process of laying out software cities in the future.

Software cities and similar metaphors have been applied in a variety of domains [29], [57]. For example, Langelier et al. [43], [58] as well as Bohnet and Döllner [59] used software cities to visualize quality metrics of software systems. Wettel and Lanza [8], [17]–[21] used software cities to visually explore the evolution of large-scale software system over time. Steinbrückner and Lewerentz [60], [61] adopted and extended this idea by visualizing the development history of software systems using elevated city maps. Software cities have been applied in the domains of concurrency visualization [22], software component communication and dependency visualization [11], [23]–[26], software performance visualization [62], [63], business process visualization [64], and test case analysis [65], [66]. Software cities have also been used in virtual reality [13], [14], [63], [67] and have been integrated into computer games such as Minecraft [68]. To the best of our knowledge, we are the first to employ the software city visualization metaphor in the domain of memory monitoring.

X. Current Limitations and Future Work

In this section, we discuss current limitations of our work and how we will address them in the future.

A. User Study

We believe that memory cities are a useful metaphor to inspect memory growth, especially for novice users that could otherwise be easily overwhelmed if the visualized data was presented in raw format or tables. We presented case studies to demonstrate the usefulness of memory cities and to showcase how they can be used to inspect real-world applications. Nevertheless, a more thorough evaluation is still missing. We thus plan to conduct a user study in the future to compare the performance of participants who use memory cities with the performance of those who use other tools.

B. Expert Mode

Currently, a primary focus of memory cities is to make the task of *memory leak analysis* more novice-friendly. For this, we rely on a small set of visual attributes, namely area, height, position, color, and opacity. In their taxonomy of software maps (the term software city is not uniquely defined in the software cartography domain), Limberger et al. [29] presented a large set of visual attributes that can be used to map data to the software city metaphor. However, they also mention that *a complex mapping [...] should be used for complex tasks or expert systems only*. Thus, we plan to further expand the feature set of memory cities in the future, including the use of more complex visual mappings such as a more advanced growth visualization using different object shapes and juxtaposition. These "expert mode features" should not be enabled by default but could be switched on by experienced memory analysts. Memory cites can also be expanded to support other typical memory analysis tasks such as *memory churn analysis* [69], [70] or *memory bloat analysis* [71]–[75].

XI. Conclusion

In this paper, we presented our *memory cities* approach to visualize memory monitoring data using the software city metaphor. We discussed how a heap state, more specifically its heap objects, can be grouped into a tree, and how such a tree can be visualized as districts and buildings. Our approach is not only able to display a single heap state, but can also visualize the memory evolution over time by using static animation positioning and time traveling. Our approach can animate the memory evolution of an application as a city that evolves over time, where growing buildings hint at a proliferation of objects that could be the result of a possible memory leak. Such growing buildings are further highlighted using color and opacity.

We implemented our approach as a standalone 3D visualization tool using Unity and presented case studies on different applications to show its feasibility and usefulness. Memory cities have especially been designed with a focus on easy accessibility even for novice users. We hope that they can assist experienced users as well as users with a limited background in memory analysis to visually inspect their applications for memory anomalies and problems. We also think that memory cities and their immersive visualizations could even be used for other tasks besides typical memory analysis. For example, they could be used in software engineering education to teach students about the risks of careless use of memory in a less theoretical but more tangible way.

XII. Acknowledgement

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

References

[1] M. Weninger, L. Makor, and H. Mössenböck, "Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor - Artifact (Binaries, Data Sets, Video, Instructions)," 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3991785

[2] G. H. Xu and A. Rountev, "Precise Memory Leak Detection for Java Software Using Container Profiling," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 17:1–17:28, 2013. [Online]. Available: https://doi.org/10.1145/2491509.2491511

[3] M. Weninger, E. Gander, and H. Mössenböck, "Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring," in *Proc. of the 9th Symp. on Software Performance (SSP)*, 2018, pp. 64–66. [Online]. Available: http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/WeningerGanderMoessenboeck18.pdf

[4] ——, "Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection," in *Proc. of the 2019 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2019, pp. 273–284. [Online]. Available: https://doi.org/10.1145/3297663.3310297

[5] Oracle. (2020) VisualVM. [Online]. Available: https://visualvm.github.io/

[6] Eclipse Foundation. (2020) Eclipse Memory Analyzer (MAT). [Online]. Available: https://unity.com/

[7] B. Cornelissen, A. Zaidman, A. van Deursen, and B. V. Rompaey, "Trace Visualization for Program Comprehension: A Controlled Experiment," in *Proc. of the 17th IEEE Int'l. Conf. on Program Comprehension (ICPC)*, 2009, pp. 100–109. [Online]. Available: https://doi.org/10.1109/ICPC.2009.5090033

[8] R. Wettel, M. Lanza, and R. Robbes, "Software Systems as Cities: A Controlled Experiment," in *Proc. of the 33rd Int'l. Conf. on Software Engineering (ICSE)*, 2011, pp. 551–560. [Online]. Available: https://doi.org/10.1145/1985793.1985868

[9] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller, "Comparing Trace Visualizations for Program Comprehension Through Controlled Experiments," in *Proc. of the 23rd IEEE Int'l. Conf. on Program Comprehension (ICPC)*, 2015, pp. 266–276. [Online]. Available: https://doi.org/10.1109/ICPC.2015.37

[10] F. Fittkau, A. Krause, and W. Hasselbring, "Hierarchical Software Landscape Visualization for System Comprehension: A Controlled Experiment," in *Proc. of the 3rd IEEE Working Conf. on Software Visualization (VISSOFT)*, 2015, pp. 36–45. [Online]. Available: https://doi.org/10.1109/VISSOFT.2015.7332413

[11] ——, "Software Landscape and Application Visualization for System Comprehension with ExplorViz," *Inf. Softw. Technol.*, vol. 87, pp. 259–277, 2017. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.07.004

[12] A. F. Blanco, J. P. S. Alcocer, and A. Bergel, "Effective Visualization of Object Allocation Sites," in *Proc. of the IEEE Working Conference on Software Visualization (VISSOFT)*, 2018, pp. 43–53. [Online]. Available: https://doi.org/10.1109/VISSOFT.2018.00013

[13] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza, "On The Use of Virtual Reality in Software Visualization: The Case of the City Metaphor," *Inf. Softw. Technol.*, vol. 114, pp. 92–106, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2019.06.007

[14] ——, "The City Metaphor in Software Visualization: Feelings, Emotions, and Thinking," *Multim. Tools Appl.*, vol. 78, no. 23, pp. 33 113–33 149, 2019. [Online]. Available: https://doi.org/10.1007/s11042-019-07748-1

[15] C. Knight and M. Munro, "Virtual but Visible Software," in *Proc. of the Int'l. Conf. on Information Visualisation, (IV)*, 2000, pp. 198–205. [Online]. Available: https://doi.org/10.1109/IV.2000.859756

[16] ——, "Comprehension with[in] Virtual Environment Visualisations," in *Proc. of the 7th Int'l. Workshop on Program Comprehension (IWPC)*, 1999, pp. 4–11. [Online]. Available: https://doi.org/10.1109/WPC.1999.777733

[17] R. Wettel and M. Lanza, "Visualizing Software Systems as Cities," in *Proc. of the 4th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2007, pp. 92–99. [Online]. Available: https://doi.org/10.1109/VISSOF.2007.4290706

[18] ——, "Program Comprehension Through Software Habitability," in *Proc. of the 15th Int'l. Conf. on Program Comprehension (ICPC)*, 2007, pp. 231–240. [Online]. Available: https://doi.org/10.1109/ICPC.2007.30

[19] ——, "Visual Exploration of Large-Scale System Evolution," in *Proc. of the 15th Working Conf. on Reverse Engineering (WCRE)*, 2008, pp. 219–228. [Online]. Available: https://doi.org/10.1109/WCRE.2008.55

[20] ——, "CodeCity: 3D Visualization of Large-Scale Software," in *Comp. Proc. of the 30th Int'l. Conf. on Software Engineering (ICSE Comp.)*, 2008, pp. 921–922. [Online]. Available: https://doi.org/10.1145/1370175.1370188

[21] R. Wettel, "Visual exploration of large-scale evolving software," in *Comp. of the 31st Int'l. Conf. on Software Engineering (ICSE Comp.)*, 2009, pp. 391–394. [Online]. Available: https://doi.org/10.1109/ICSE-COMPANION.2009.5071029

[22] J. Waller, C. Wulf, F. Fittkau, P. Dohring, and W. Hasselbring, "Synchrovis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency," in *Proc. of the 1st IEEE Working Conf. on Software Visualization (VISSOFT)*, 2013, pp. 1–4. [Online]. Available: https://doi.org/10.1109/VISSOFT.2013.6650520

[23] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, "Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach," in *Proc. of the 1st IEEE Working Conf. on Software Visualization (VISSOFT)*, 2013, pp. 1–4. [Online]. Available: https://doi.org/10.1109/VISSOFT.2013.6650536

[24] F. Fittkau, A. van Hoorn, and W. Hasselbring, "Towards a Dependability Control Center for Large Software Landscapes," in *Proc. of the 10th European Dependable Computing Conf.*, 2014, pp. 58–61. [Online]. Available: https://doi.org/10.1109/EDCC.2014.12

[25] F. Fittkau, P. Stelzer, and W. Hasselbring, "Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance," in *Proc. of the European Conf. on Software Architecture (ECSA)*, 2014, pp. 28:1–28:4. [Online]. Available: https://doi.org/10.1145/2642803.2642831

[26] F. Fittkau, S. Roth, and W. Hasselbring, "ExplorViz: Visual Runtime Behavior Analysis of Enterprise Application Landscapes," in *Proc. of the European Conf. on Information Systems (ECIS)*, 2015. [Online]. Available: http://aisel.aisnet.org/ecis2015_cr/46

[27] M. Weninger, L. Makor, and H. Mössenböck, "Memory Leak Visualization using Evolving Software Cities," in *Proc. of the 10th Symp. on Software Performance (SSP)*, 2019, pp. 44–46. [Online]. Available: http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf

[28] S. dos Santos and K. Brodlie, "Gaining Understanding of Multivariate and Multidimensional Data Through Visualization," *Computers & Graphics*, vol. 28, no. 3, pp. 311 – 325, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849304000251

[29] D. Limberger, W. Scheibel, J. Döllner, and M. Trapp, "Advanced Visual Metaphors and Techniques for Software Maps," in *Proc. of the 12th Int'l. Symp. on Visual Information Communication and Interaction (VINCI)*, 2019, pp. 11:1–11:8. [Online]. Available: https://doi.org/10.1145/3356422.3356444

[30] P. Lengauer, V. Bitto, and H. Mössenböck, "Accurate and Efficient Object Tracing for Java Applications," in *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2015, pp. 51–62. [Online]. Available: https://doi.org/10.1145/2668930.2688037

[31] ——, "Efficient and Viable Handling of Large Object Traces," in *Proc. of the 7th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2016, pp. 249–260. [Online]. Available: https://doi.org/10.1145/2851553.2851555

[32] P. Lengauer, V. Bitto, S. Fitzek, M. Weninger, and H. Mössenböck, "Efficient Memory Traces with Full Pointer Information," in *Proc. of the 13th Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*, 2016, pp. 4:1–4:11. [Online]. Available: https://doi.org/10.1145/2972206.2972220

[33] V. Bitto, P. Lengauer, and H. Mössenböck, "Efficient Rebuilding of Large Java Heaps from Event Traces," in *Proc. of the Int'l. Conf. on Principles and Practices of Programming on The Java Platform (PPPJ)*, 2015, pp. 76–89. [Online]. Available: https://doi.org/10.1145/2807426.2807433

[34] M. Weninger, P. Lengauer, and H. Mössenböck, "User-centered Offline Analysis of Memory Monitoring Data," in *Proc. of the 8th ACM/SPEC on Int'l. Conf. on Performance Engineering (ICPE)*, 2017, pp. 357–360. [Online]. Available: https://doi.org/10.1145/3030207.3030236

[35] M. Weninger and H. Mössenböck, "User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring," in *Proc. of the 2018 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2018, pp. 115–126. [Online]. Available: https://doi.org/10.1145/3184407.3184412

[36] M. Weninger, E. Gander, and H. Mössenböck, "Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring," in *Proc. of the 15th Int'l. Conf. on Managed Languages & Runtimes (ManLang)*, 2018, pp. 14:1–14:13. [Online]. Available: https://doi.org/10.1145/3237009.3237023

[37] M. Weninger, L. Makor, E. Gander, and H. Mössenböck, "AntTracks TrendViz: Configurable Heap Memory Visualization Over Time," in *Comp. of the 2019 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2019, pp. 29–32. [Online]. Available: https://doi.org/10.1145/3302541.3313100

[38] M. Weninger, E. Gander, and H. Mössenböck, "Detection of Suspicious Time Windows In Memory Monitoring," in *Proc. of the 16th ACM SIGPLAN Int'l. Conf. on Managed Programming Languages and Runtimes (MPLR)*, 2019, pp. 95–104. [Online]. Available: https://doi.org/10.1145/3357390.3361025

[39] B. Johnson and B. Shneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures," in *Proc. of the IEEE Conf. on Visualization*, 1991, pp. 284–291. [Online]. Available: https://doi.org/10.1109/VISUAL.1991.175815

[40] B. Shneiderman, "Tree Visualization with Tree-Maps: 2-D Space-Filling Approach," *ACM Trans. Graph.*, vol. 11, no. 1, pp. 92–99, 1992. [Online]. Available: https://doi.org/10.1145/102377.115768

[41] M. Bruls, K. Huizing, and J. J. van Wijk, "Squarified Treemaps," in *Proc. of the Joint Eurographics and IEEE TCVG Symp.*

on Visualization (VisSym), 2000, pp. 33–42. [Online]. Available: https://doi.org/10.1007/978-3-7091-6783-0_4
[42] M. Weninger, P. Grünbacher, E. Gander, and A. Schörgenhumer, "Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study," *Proc. ACM Hum.-Comput. Interact.*, vol. 4, no. EICS, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3394977
[43] G. Langelier, H. A. Sahraoui, and P. Poulin, "Exploring the Evolution of Software Quality with Animated Visualization," in *Proc. of the IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*, 2008, pp. 13–20. [Online]. Available: https://doi.org/10.1109/VLHCC.2008.4639052
[44] W. Scheibel, M. Trapp, D. Limberger, and J. Döllner, "A Taxonomy of Treemap Visualization Techniques," in *Proc. of the 15th Int'l. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, 2020, pp. 273–280. [Online]. Available: https://doi.org/10.5220/0009153902730280
[45] G. Belov, W. Du, M. G. de la Banda, D. Harabor, S. Koenig, and X. Wei, "From Multi-Agent Pathfinding to 3D Pipe Routing," in *Proc. of the Int'l. Symp. on Combinatorial Search (SOCS)*, 2020, pp. 11–19. [Online]. Available: https://aaai.org/ocs/index.php/SOCS/SOCS20/paper/view/18513
[46] P. Caserta, O. Zendra, and D. Bodenes, "3D Hierarchical Edge bundles to Visualize Relations in a Software City Metaphor," in *Proc. of the 6th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2011, pp. 1–8. [Online]. Available: https://doi.org/10.1109/VISSOF.2011.6069451
[47] G. Lakoff, *Master Metaphor List*. University of California, 1994.
[48] L. J. W. Jr., "Visual Metaphors for Teaching Programming Concepts," in *Proc. of the SIGCSE Techn. Symp. on Computer Science Education*, 1989, pp. 141–145. [Online]. Available: https://doi.org/10.1145/65293.71203
[49] J. Boyle and P. M. D. Gray, "The Design of 3D Metaphors for Database Visualisation," in *Proc. of the 3rd IFIP 2.6 Working Conf. on Visual Database Systems*, vol. 34, 1995, pp. 185–202. [Online]. Available: https://doi.org/10.1007/978-0-387-34905-3_12
[50] B. Reitinger, D. Kranzlmüller, and J. Volkert, "The MOST Immersive Approach for Parallel and Distributed Program Analysis," in *Proc. of the Int'l. Conf. on Information Visualisation (IV)*. IEEE Computer Society, 2001, pp. 517–522. [Online]. Available: https://doi.org/10.1109/IV.2001.942105
[51] B. Reitinger, D. Kranzlmüller, and A. Ferko, "Program Visualization Through Visual Metaphors," in *Proc. of the Int'l. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2003. [Online]. Available: http://wscg.zcu.cz/wscg2003/Papers_2003/J79.pdf
[52] A. Hiniker, S. R. Hong, Y. Kim, N. Chen, J. D. West, and C. R. Aragon, "Toward the Operationalization of Visual Metaphor," *J. Assoc. Inf. Sci. Technol.*, vol. 68, no. 10, pp. 2338–2349, 2017. [Online]. Available: https://doi.org/10.1002/asi.23857
[53] T. Panas, R. Berrigan, and J. C. Grundy, "A 3D Metaphor for Software Production Visualization," in *Proc. of the Seventh Int'l. Conf. on Information Visualization (IV)*, 2003, pp. 314–319. [Online]. Available: https://doi.org/10.1109/IV.2003.1217996
[54] T. Bladh, D. A. Carr, and J. Scholl, "Extending Tree-Maps to Three Dimensions: A Comparative Study," in *Proc. of the 6th Asia Pacific Conf. on Computer Human Interaction (APCHI)*, 2004, pp. 50–59. [Online]. Available: https://doi.org/10.1007/978-3-540-27795-8_6
[55] W. Scheibel, C. Weyand, and J. Döllner, "EvoCells - A Treemap Layout Algorithm for Evolving Tree Data," in *Proc. of the 13th Int'l. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, 2018, pp. 273–280. [Online]. Available: https://doi.org/10.5220/0006617102730280
[56] M. Sondag, B. Speckmann, and K. Verbeek, "Stable Treemaps via Local Moves," *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 729–738, 2018. [Online]. Available: https://doi.org/10.1109/TVCG.2017.2745140
[57] P. Caserta and O. Zendra, "Visualization of the Static Aspects of Software: A Survey," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 7, pp. 913–933, 2011. [Online]. Available: https://doi.org/10.1109/TVCG.2010.110
[58] G. Langelier, H. A. Sahraoui, and P. Poulin, "Visualization-based Analysis of Quality for Large-scale Software Systems," in *Proc. of the 20th IEEE/ACM Int'l. Conf. on Automated Software Engineering (ASE)*, 2005, pp. 214–223. [Online]. Available: https://doi.org/10.1145/1101908.1101941
[59] J. Bohnet and J. Döllner, "Monitoring Code Quality and Development Activity by Software Maps," in *Proc. of the 2nd Workshop on Managing Technical Debt (MTD)*, 2011, pp. 9–16. [Online]. Available: https://doi.org/10.1145/1985362.1985365
[60] F. Steinbrückner and C. Lewerentz, "Representing Development History in Software Cities," in *Proc. of the ACM Symp. on Software Visualization (SOFTVIS)*, 2010, pp. 193–202. [Online]. Available: https://doi.org/10.1145/1879211.1879239
[61] ——, "Understanding software evolution with software cities," *Information Visualization*, vol. 12, no. 2, pp. 200–216, 2013. [Online]. Available: https://doi.org/10.1177/1473871612438785
[62] K. Ogami, R. G. Kula, H. Hata, T. Ishio, and K. Matsumoto, "Using High-Rising Cities to Visualize Performance in Real-Time," in *Proc. of the IEEE Working Conference on Software Visualization (VISSOFT)*, 2017, pp. 33–42. [Online]. Available: https://doi.org/10.1109/VISSOFT.2017.25
[63] L. Merino, M. Hess, A. Bergel, O. Nierstrasz, and D. Weiskopf, "PerfVis: Pervasive Visualization in Immersive Augmented Reality for Performance Awareness," in *Comp. of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2019, pp. 13–16. [Online]. Available: https://doi.org/10.1145/3302541.3313104
[64] S. Saito, "ProcessCity - Visualizing Business Processes as City Metaphor," in *Proc. of the CAiSE Forum on Information Systems Engineering in Responsible Information Systems*, 2019, pp. 207–214. [Online]. Available: https://doi.org/10.1007/978-3-030-21297-1_18
[65] A. Sosnówka, "Test City Metaphor as Support for Visual Testcase Analysis Within Integration Test Domain," in *Proc. of the Federated Conf. on Computer Science and Information Systems*, 2013, pp. 1353–1358. [Online]. Available: http://ieeexplore.ieee.org/document/6644194/
[66] ——, "Test City Metaphor for Low Level Tests Restructuration in Test Database," in *Proc. of the 8th Int'l. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2013, pp. 141–150. [Online]. Available: https://doi.org/10.1007/978-3-642-54092-9_10
[67] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring Software Cities in Virtual Reality," in *Proc. of the 3rd IEEE Working Conf. on Software Visualization (VISSOFT)*, 2015, pp. 130–134. [Online]. Available: https://doi.org/10.1109/VISSOFT.2015.7332423
[68] G. Balogh and Á. Beszédes, "CodeMetrpolis - A Minecraft based Collaboration Tool for Developers," in *Proc. of the IEEE Working Conf. on Software Visualization (VISSOFT)*, 2013, pp. 1–4. [Online]. Available: https://doi.org/10.1109/VISSOFT.2013.6650528
[69] C. U. Smith and L. G. Williams, "Software Performance Antipatterns," in *Proc. of the Int'l. Workshop on Software and Performance (WOSP)*, 2000, pp. 127–136. [Online]. Available: https://doi.org/10.1145/350391.350420
[70] M. Peiris and J. H. Hill, "Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern," in *Proc. of the 7th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*. ACM, 2016, pp. 237–248. [Online]. Available: https://doi.org/10.1145/2851553.2851563
[71] N. Mitchell and G. Sevitsky, "The Causes of Bloat, the Limits of Health," in *Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007, pp. 245–260. [Online]. Available: https://doi.org/10.1145/1297027.1297046
[72] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-scale Object-oriented Applications," in *Proc. of the Workshop on Future of Software Engineering Research (FoSER)*, 2010, pp. 421–426. [Online]. Available: https://doi.org/10.1145/1882362.1882448
[73] G. H. Xu and A. Rountev, "Detecting Inefficiently-used Containers to Avoid Bloat," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2010, pp. 160–173. [Online]. Available: https://doi.org/10.1145/1806596.1806616
[74] N. Mitchell, E. Schonberg, and G. Sevitsky, "Four Trends Leading to Java Runtime Bloat," *IEEE Software*, vol. 27, no. 1, pp. 56–63, 2010. [Online]. Available: https://doi.org/10.1109/MS.2010.7
[75] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, "Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, pp. 23:1–23:50, 2014. [Online]. Available: https://doi.org/10.1145/2560047