Author
Daniel Rašo

Submission
Institute for System Software

Thesis Supervisor
Dipl.-Ing. Dr. Markus Weninger, BSc.

September 2022

ONLINE MEMORY CITY VISUALIZATION TOOL

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Bachelor's Thesis

Online Memory City Visualization Tool

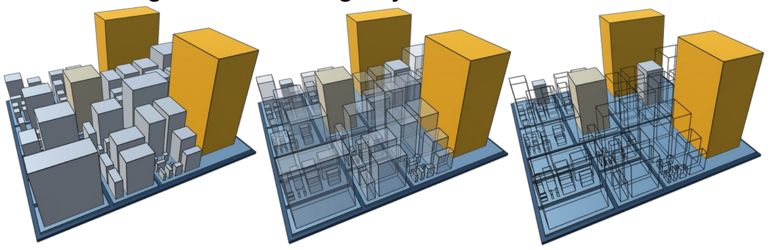
Dipl.-Ing. Dr. Markus Weninger, BSc
Institute for System Software
P +43-732-2468-4361
markus.weninger@jku.at

Student: Daniel Rašo

Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc

Start date: February 2022

Memory Cities, are 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.



Currently, Memory Cities is a standalone application developed in Unity, with a JSON-based input file format for easy data import from external tools. Typically, we use Memory Cites in conjunction with AntTracks, a trace-based memory monitoring tool that is able to export memory data in this format.

The goal of this bachelor thesis is to port our Memory Cities desktop application (built in Unity using C#) to a web-based visualization tool that can be used from within the browser without any manual download or installation needed. To achieve this, Memory Cities should be ported to JavaScript, specifically using the three.js 3D library (https://threejs.org/). All features (time travel, color highlighting, opacity settings, reference visualization, etc.) supported in the desktop application must also be available in the web tool. As a nice-to-have feature, animations could be used to fluently transform from one point in time to another. Since the tool will be extended in the future, readability and a clean code structure is important, as well as clean documentation (i.e., documented method headers).

Modalities:

The progress of the project should be discussed at least every three weeks with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor and the supervisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 30.09.2022.

Abstract

Understanding the memory behavior of intricate applications, together with the task of detecting hard-to-find memory leaks, is tied to the role of memory analysts and experienced developers. In their inspiring paper, Weninger et al. [10] presented *Memory Cities*, a novice-friendly tool that *visualizes an application's heap memory evolution over time using the software city metaphor*. In a memory city, heap objects (that were previously grouped by properties such as package and type) are represented as buildings arranged in districts. The size of a building portrays the object/byte count of its respective heap object group. Constantly evolving buildings reveal suspicious growth in heap memory along with memory leaks.

The visualization tool was developed in Unity. Unfortunately, this requires a Unity installation, or the manual download of Memory Cities, in order to run the visualization. In this work, we overcome these problems by introducing the online version of Memory Cities, a *web application* accessible from any browser without any download or installation needed. Using JavaScript and the 3D library *Three.js*, we supply all the features in the desktop application and extend the feature set to some degree.

Kurzfassung

Verständnis für das Verhalten des Speichers von komplexen Anwendungen sowie die Erkennung von schwer aufzuspürenden Speicherlecks sind an die Rolle von Speicheranalysten und erfahrenen Entwicklern gebunden. In ihrem inspirierenden Paper stellten Weninger et al. [10] *Memory Cities* vor, ein benutzerfreundliches Werkzeug, dass die *zeitliche Entwicklung des Heap-Speichers einer Anwendung mittels der Software-Stadt-Metapher visualisiert*. In einer Speicherstadt werden Heap-Objekte (die zuvor nach Eigenschaften wie Paket und Typ gruppiert wurden) als Gebäude dargestellt, die in Bezirken angeordnet sind. Die Größe eines Gebäudes stellt die Anzahl an Objekten/Bytes der jeweiligen Heap-Objektgruppe dar. Ständig sich entwickelnde Gebäude geben verdächtiges Wachstum des Heap-Speichers sowie Speicherlecks preis.

Das Visualisierungstool wurde in Unity entwickelt. Leider erfordert dies die Installation von Unity oder den manuellen Download von Memory Cities, um die Visualisierung auszuführen. In dieser Arbeit überwinden wir diese Probleme, indem wir die Online-Version von Memory Cities entwickeln, eine *Web-Anwendung*, die von jedem Browser aus verfügbar ist, ohne Download oder Installation. Mithilfe von JavaScript und der 3D-Bibliothek *Three.js* stellen wir alle Funktionen der Desktop-Anwendung bereit und erweitern das Funktionsspektrum in gewissem Maße.

Table of Contents

1 Introduction

The most efficient way of storing local variables and primitives such as integers and booleans is to put them on the local stack memory. Local variables are allocated automatically when a method is called and deallocated automatically when the method exits. In contrast, we use the *heap* memory (also known as dynamic memory) to store larger and more complex (user-defined) objects. Furthermore, the programmer explicitly requests allocating that space, for instance, by using the `new` operator in Java. In some programming languages (notably C and C++), the developer explicitly needs to request an object in heap memory to be deallocated. Dropping all references to a memory location without deallocating it is a significant source of errors in C/C++ and causes memory leaks.

Modern programming languages such as Java or JavaScript (JS) release the developers from that task of freeing memory manually by using automatic garbage collection. The garbage collector (GC) determines which heap objects are no longer being used by examining the *GC roots* (e.g., static fields or thread-local variables) [9] and releases the memory allocated for them. Despite this, memory problems and anomalies such as memory leaks can arise even in garbage-collected languages. For example, a developer may forget to remove objects from their containing data structures. The garbage collector cannot reclaim these objects, which will accumulate over time.

Memory leaks can be identified by (1) finding suspicious objects with large ownership, i.e., those objects that keep alive a lot of other objects and (2) searching for groups of objects that grow suspiciously over time. Memory monitoring tools such as VisualVM[1], Eclipse MAT[2] or the Chrome DevTools heap profiler[3] help users achieve this task of finding potential memory leaks. Unfortunately, many of these state-of-the-art tools lack a visualization method to emphasize the heap's evolution over time (except for time-series charts).

As a result, the authors in [10] presented *Memory Cities*, an approach to visualize the heap's memory evolution using the *software city metaphor*. Such a city comprises *buildings* that represent heap object groups that are arranged in *districts* based on shared heap object properties such as type (e.g. String). As a monitored application progresses in time, its heap object groups will increase and shrink in size. As a consequence, buildings in Memory Cities can also change in dimensions.

Currently, Memory Cities is a standalone application developed in Unity[4], necessitating Unity to be available or installing Memory Cities directly as a desktop application. This bachelor thesis tackles this problem and aims to port the original Memory Cities tool to a web-based visualization tool that can be used within a browser without manual downloading or installation. To achieve this, we use JavaScript together with

[1] http://visualvm.github.io/
[2] https://www.eclipse.org/mat/
[3] https://developer.chrome.com/docs/devtools/memory-problems/
[4] https://unity.com/

Three.js[5], a 3D library for the web. The main requirements of the new tool are as follows:

- all features supported in the desktop application must also be available in the web tool (including *Time Travel*, color highlighting/opacity settings and reference visualization) (Section 3.4) (Section 3.5)
- adding additional features such as *tiling method* (Section 3.3) and animations (Section 5.4)
- clean code structure, as well as clean documentation (e.g., documented method headers), considering that the tool will advance in the future (Section 6)

2 Background & Related Work

The purpose of Memory Cities is to help out developers during the *memory analysis process* and in *detecting suspiciously growing heap object groups*. The tool takes existing techniques from the memory monitoring domain and combines them with the software city metaphor, resulting in a new category of software visualization tools.

Hence, this section serves as an introduction to these domains.

2.1 Memory Analysis

The process of memory analysis usually starts with taking a snapshot (so-called *heap dump*) of the heap memory. A heap dump is a representation of all the objects that were in memory at a certain point in time. Secondly, numerous heap objects are grouped according to some criteria, such as type or call site, resulting in a *memory tree*. Lastly, memory profiling tools provide several views of the heap that show a user which objects account for memory usage and which code parts allocate space. For the most part, tools display a memory tree using the tree table view, similar to the one shown in **Table 1**.

Table 1. A memory tree grouped by types and allocation sites represented in tree table view. Shallow size is the object size itself. Retained size of an object is its shallow size plus the shallow sizes of the objects that are accessible, directly or indirectly, only from this object.

	Object Count	*Shallow Size*	*Retained Size*
▪ Heap (root)	10,000	95 MB	345211 MB
o type String	5,000	…	…
• allocated in substring()	2,000	…	…
• allocated in toString()	3,000	…	…
o type Integer	2,500	…	…
• allocated in valueOf()	2,200	…	…
…			

[5] https://threejs.org/

Most tools also provide a feature that compares two snapshots and highlights the delta in freed memory and reference count between two points in time. They do this by calculating the differences in the number of objects and displaying these discrepancies in a tree table. Although comparing two heap dumps may confirm the presence and cause of a memory leak, it does not reveal *general trends in an application's memory behavior* or heap object groups' growth. As we will see in Section 2.3, Memory Cities addresses this problem, which is an uncommon and novel feature of most state-of-the-art tools.

2.2 Software Cities in General

Building knowledge of large-scale software systems is a tedious task. Advanced visualization techniques such as the software city metaphor can be utilized to cope with this problem. As Wettel and Lanza describe in their paper [12], a software city typically depicts object-oriented software systems and their static artifacts, such as class hierarchies and packages. The authors developed a 3D visualization tool called CodeCity [13], in which buildings represent classes grouped into districts based on their packages. The number of methods in a class determines the height of buildings. The area a building occupies is proportional to the number of attributes inside a class.

Inspired by the widespread use of the software city metaphor, Weninger et al. [10] applied this metaphor to the domain of memory monitoring.

2.3 Original Memory Cities Tool

Memory Cities' goal is to visualize an application's dynamic memory behavior over time to help support the task of memory leak detection. It maps a heap state to a 3D environment. In particular, the tool takes a memory tree (i.e., grouped sets of heap objects) as input and displays it as a 3D city visualization. Heap object groups are visualized as buildings arranged in districts. All buildings correspond to leaf nodes in the memory tree, where a building's area and height are proportional to the number of objects/bytes represented by the respective tree node. Therefore, districts are parent tree nodes of buildings or simply inner nodes of the memory tree.

Memory cities visualize an application's heap memory evolution over time to detect trends in the growth of conspicuous heap object groups. Multiple memory trees serve as an input for the tool to accomplish this task. Every single memory tree represents the memory at one point in time. Users can then update the city over time, creating the immersive feeling of an evolving city, which makes it possible to search for strongly growing buildings. Hence, memory leaks may be discovered.

Fig. 1 shows an exemplary memory city, where the type and the allocation site are the grouping criteria (so-called *classifiers*) for heap objects, i.e., here, buildings are a set of heap objects of the same type allocated in the same method.

Having a well-defined JSON interface means that Memory Cities are independent of the memory monitoring tools that construct memory trees. Regardless of that, the authors of the original Memory Cities tool predominantly work with data imported from

AntTracks[6]. AntTracks has the distinct benefit of allowing users to choose multiple classifiers simultaneously, based on which the heap's objects are grouped (e.g., type, allocation site, call site, address, package) [11].

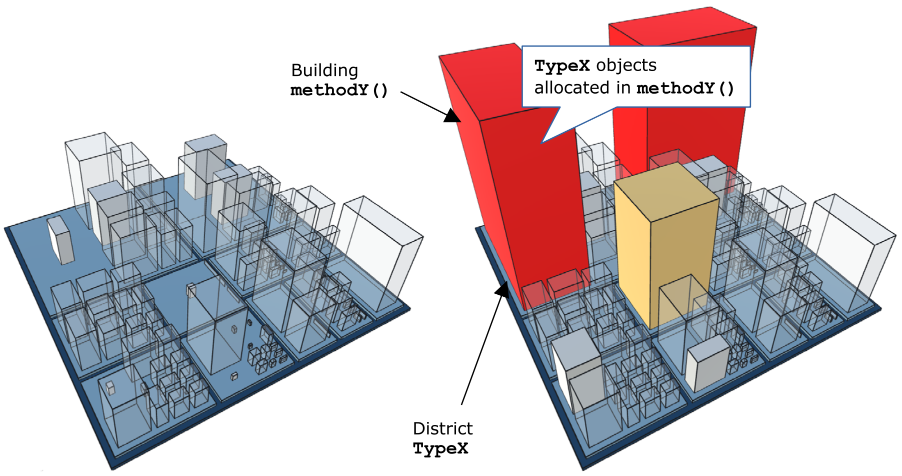


Fig. 1. *"An application's heap visualized with memory cities shortly after startup (left) and 2 minutes / 300 garbage collections later (right)"* [10]. The tool further utilizes various visual attributes for more manageable memory leak detection. It highlights strongly growing buildings using color, while making less suspicious buildings (semi-)transparent.

Last but not least, the original Memory Cities tool is implemented as a standalone desktop application developed in Unity using C#. This fact covers only one of the differences compared to the new application, as we will see in Section 3.6. Nonetheless, both versions have significant similarities in their approach and theory, which are covered in the following sections.

3 Online Memory Cities

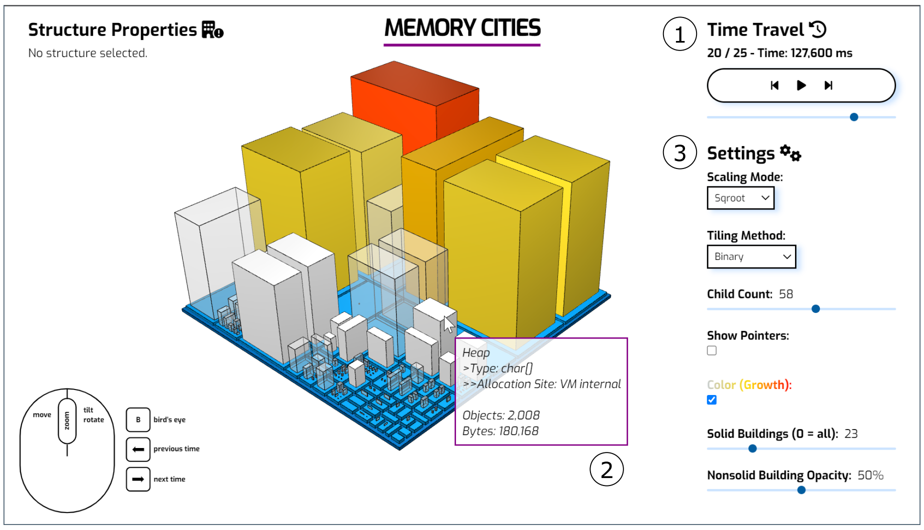
As the name suggests, the new tool is an online version, i.e., a web application, of the original one, with differences found during *data processing* (Section 3.2), in the appearance and used frameworks (Section 4). The general idea remains identical: visualize a memory tree as a 3D city composed of buildings placed in districts. The following sections will discuss the steps for creating such a memory city (presented in **Fig. 2**).

3.1 Approach

The workflow of the newly developed online tool is inspired by the original tool. It also uses nearly the same input data. Yet, certain aspects, such as metadata calculation,

[6] http://mevss.jku.at/AntTracks

differ from the original implementation. This section will present the whole workflow of the new tool, from importing the data (i.e., multiple memory trees), processing it, up to visualizing it in 3D.



No structure selected.

20 / 25 - Time: 127,600 ms

Scaling Mode: Sqroot

Tiling Method: Binary

Child Count: 58

Show Pointers: ☐

Color (Growth): ☑

Solid Buildings (0 = all): 23

Nonsolid Building Opacity: 50%

Heap
>Type: char[]
>>Allocation Site: VM internal

Objects: 2,008
Bytes: 180,168

Fig. 2. Example screenshot of the new Memory Cities tool taken in Google Chrome. (1) The *Time Travel* feature allows users to inspect the growth behavior of heap object groups. (2) Hovering over a structure (i.e., either district or building) displays information about that structure. (3) Adjusting the *settings* changes the appearance of a memory city, which can help reveal memory leaks.

The following list describes all the necessary steps (as shown in **Fig. 3**) toward creating a memory city:

(1) Once *multiple* heap dumps are taken and grouped according to a set of heap object properties, a folder containing all the resulting memory trees (as JSON files) can be selected and imported, respectively.

(2) Based on these memory trees, a JS object is created (called `structureInfo`), which maps the keys (names) of all possible tree nodes to their respective properties. These properties include the *max* value of a tree node, which stores the maximum number of objects/bytes a tree node represents at any point in time (i.e., the largest size a district or building will reach). Additionally, we calculate and store the *absolute growth* of each tree node, which is equal to the node's growth between the first and the last memory tree.

(3) We use *treemapping* [2, 7] to generate the city's general layout. First, we take the object/byte counts stored as max values in `structureInfo` to build a new tree (called *layout tree*) which serves as an input for a treemap algorithm. The layout tree is basically a memory tree where each node only has two properties; name and max object/byte count. By doing this, we

reserve an area for every building that will eventually be displayed in the city. Even if all heap object groups (buildings) reach their largest size at the same time, the generated layout guarantees that every building can fit into the city.

(4) Before we can render any new structures, i.e., districts or buildings, we first must get rid of the previously created city by disposing of all existing 3D objects.

(5) To display a memory city, we first process the inner nodes of our layout tree and render the districts as flat cuboids inside the 3D environment.

(6) To display buildings at a certain point in time, we first calculate each building's base area at that point plus its height. Secondly, the buildings are centered in their layout spots reserved for them. Lastly, the corresponding cuboids get generated on top of the districts.

(7) We adjust the color and opacity attributes of buildings to emphasize strongly growing heap object groups.

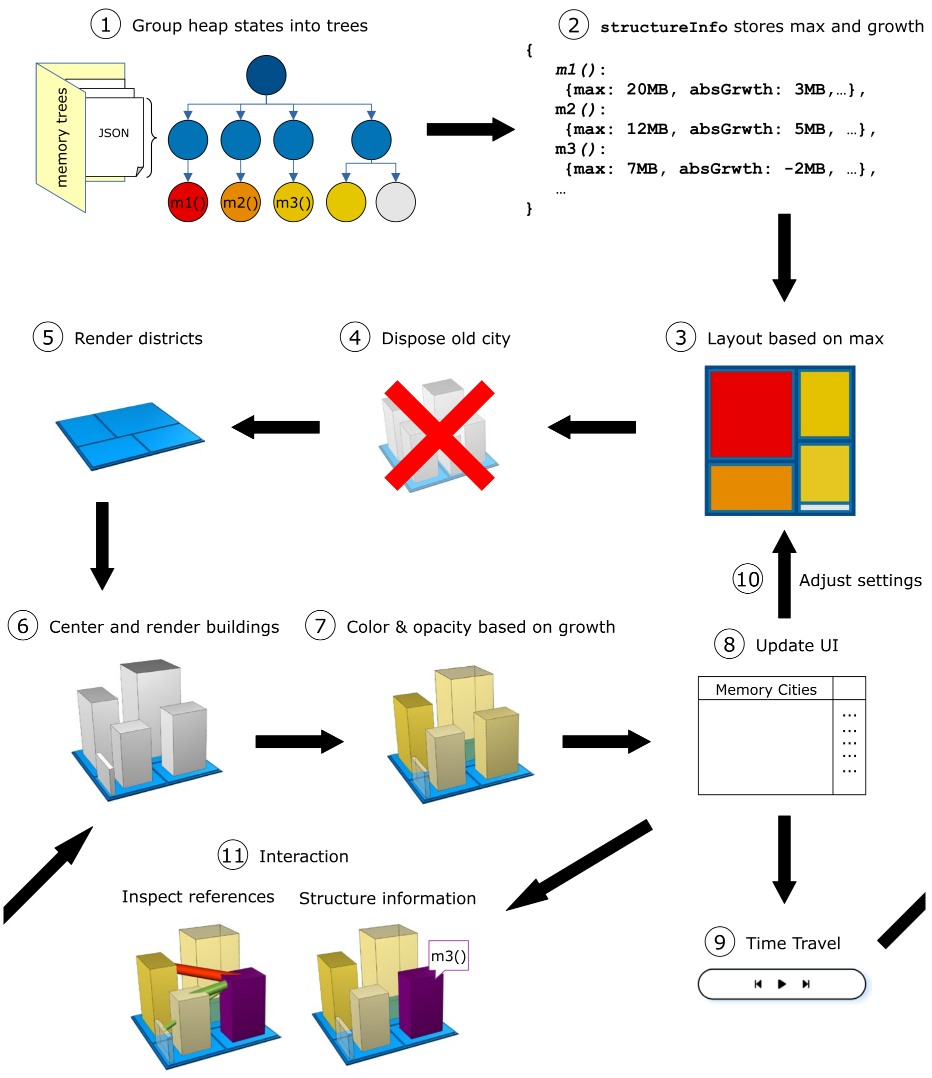
(8) After successfully rendering a memory city, we update the user interface (embedded in HTML) to match the city's current state.

(9) The Time Travel feature allows users to visualize the heap at different timestamps (one timestamp per imported memory tree). Stepping back and forth in time means executing the steps (6) to (8) and updating the visualization. Using the play button on the user interface (UI) triggers an automatic animation of the city's evolution.

(10) Applying specific settings (**Fig. 2**, (3)) (*scaling mode*, *tiling method* and *child count*), requires a new general city layout to be computed. In this case, the steps starting from (3) are performed again.

(11) There are more ways in which a user might interact with memory cities. To see information about a structure, one may hover over it or click it. Another feature is to show references between one and two (or more) buildings. This feature is inspired by the Dominators View[7] available in most memory profiling tools (or used in browsers such as Chrome and Firefox). It helps to identify heap object groups that cause other buildings to grow and those that grow because their objects are kept alive by others through references.

[7] https://firefox-source-docs.mozilla.org/devtools-user/memory/dominators_view/index.html



```
{
  m1():
   {max: 20MB, absGrwth: 3MB,…},
  m2():
   {max: 12MB, absGrwth: 5MB, …},
  m3():
   {max: 7MB, absGrwth: -2MB, …},
  …
}
```

Fig. 3. An overview of the entire memory cities approach. The main difference compared to the original tool lies within preprocessing, specifically, metadata calculation. This comes from the fact that the new tool does not use meta trees such as *max tree* and *growth tree* [10], but rather a JS object for mapping these properties to heap object groups.

In the following sections, we explain the separate steps in more detail. These sections focus on concepts; for implementation details, see Section 5.

3.2 Data

This section will answer the following two questions in more detail: What data do we need to be a fully functional memory city? How is this data processed in order to build a treemap?

Fig. 4 presents the major steps in processing input data (set of memory trees) based on which memory cities are created. The following list explains each one of them in more detail:

(1) In general, a software city is built upon tree data. Each tree node has to hold at least two properties: a unique key for labelling and identification purposes and one value based on which the city gets laid out. In the case of memory cities, each tree node represents a group of heap objects which share common properties (classifiers) like type or call site. Each node or group is quantified by its object count or by counting the number of bytes the respective objects take up in a heap. Furthermore, as the goal is to visualize the heap's evolution over time, memory cities load a set of memory trees instead of only one. Not only does each tree represent the heap state at a garbage collection point, but it also possesses a timestamp to ensure correct ordering [10]. To summarize, the Memory Cities tool expects the following data (format) inside a JSON file:

- a *classifiers* array containing grouping criteria information (e.g. `[{"name": "Type", "id": 0}, {"name": "Allocation Site", "id": 1}]`)
- the *time* at which the snapshot of the heap was taken
- the *root* node of this memory tree, where each tree node contains:
  - a *key*/name to display (e.g. `"methodY()"`)
  - a *fullKey* array containing all the *keys* from the root node to the current node (e.g. `["Heap", "TypeX", "methodY()"]`)
  - a unique *fullKeyAsString* to identify the heap object group (e.g. `"Heap#TypeX#methodY()"`)
  - a *classifierId* to identify the grouping criteria of the heap object group
  - *objects* value
  - *bytes* value
  - a *children* array (non-existent for leaf nodes/buildings)

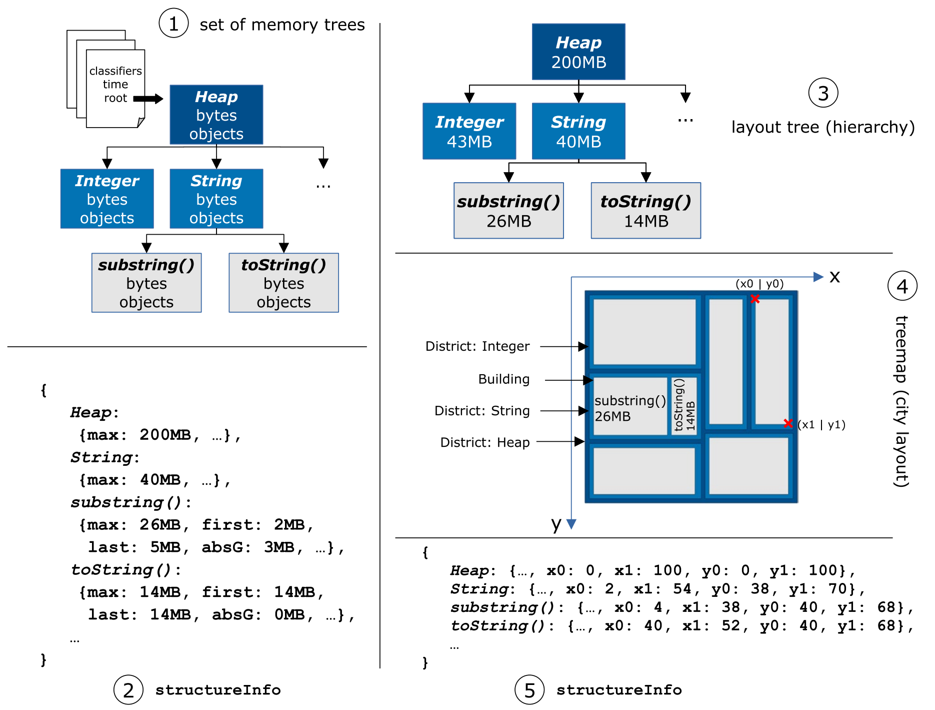
(2) To build a city, we first need a map or a layout which tells us where to position our districts and buildings. Before computing the layout using treemapping, we first have to determine and store the largest size a structure may reach (i.e., out of all memory trees, the most significant object/byte count for each node). In the case of buildings, we are also interested in growth information (i.e., the difference between the first and last object/byte count for each node; *absG*). All this information about a structure

gets stored in a separate JS object called `structureInfo` (among other properties, see Section 5.1).

(3) A treemapping algorithm expects a tree data structure, where each node has a name and value property (the so-called layout tree; or *hierarchy* as stated in Section 5.2). In fact, this tree is a simplified memory tree, where the only two values inside a node are the *fullKeyAsString* of the heap object group and the max object/byte count stored in `structureInfo`.

(4) An algorithm returns the final layout of our city, including the *coordinates* of each rectangle which corresponds to a node.

(5) In the last step of data processing, we save the coordinates (*x0*, *x1*, *y0*, *y1*) of the base area for each building in the `structureInfo` object.



```
{
   Heap:
    {max: 200MB, …},
   String:
    {max: 40MB, …},
   substring():
    {max: 26MB, first: 2MB,
     last: 5MB, absG: 3MB, …},
   toString():
    {max: 14MB, first: 14MB,
     last: 14MB, absG: 0MB, …},
   …
}
```

```
{
   Heap: {…, x0: 0, x1: 100, y0: 0, y1: 100},
   String: {…, x0: 2, x1: 54, y0: 38, y1: 70},
   substring(): {…, x0: 4, x1: 38, y0: 40, y1: 68},
   toString(): {…, x0: 40, x1: 52, y0: 40, y1: 68},
   …
}
```

Fig. 4. An overview of the data processing pipeline. We use treemapping to create the general city layout.

3.3 Layout

This section shows how we can use various treemapping algorithms in order to lay out a memory city. Additionally, we present techniques that help achieve a stable and (if required) less complex layout.

Single Tree. Introduced by Shneiderman and Johnson [2, 7], a treemap recursively partitions space into rectangles based on *hierarchical tree-structured data*. A rectangle represents each node in the (memory) tree, and the rectangles of child nodes are placed within the rectangles for the parent nodes (i.e., so-called nested treemap). One of the tree node's values determines the size of its rectangle. In the case of memory cities, this is either the object count or byte count value. Instead of using the value directly, a user can apply three *scaling modes* (as shown in **Fig. 5**) to control the ratio between the value and the building size.

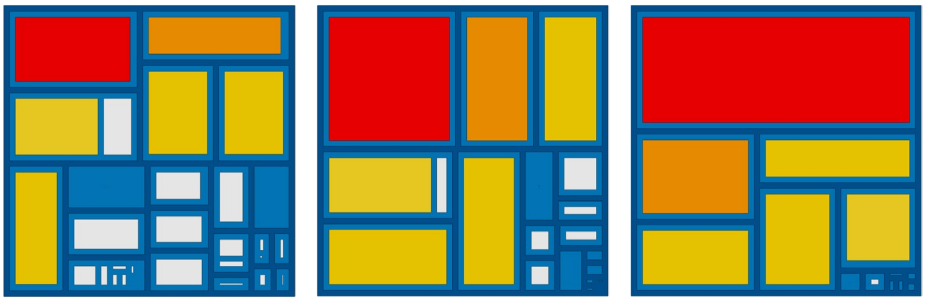


Fig. 5. Different mapping functions applied to each node's object/byte count result in distinct city layouts : `sqrt` (left), linear (middle) and quadratic (right).

To create a treemap, we call a treemap layout function, which expects the parameters: (1) the root node of a tree (2) the size of the rectangle, which represents the root node (3) a recursive *tiling algorithm*. The alignment, ordering and aspect ratio of rectangles and the ability to preserve stability when there are changes in underlying data vary between tiling algorithms [6]. **Fig. 6** presents all five tiling methods available in the new Memory Cities tool.

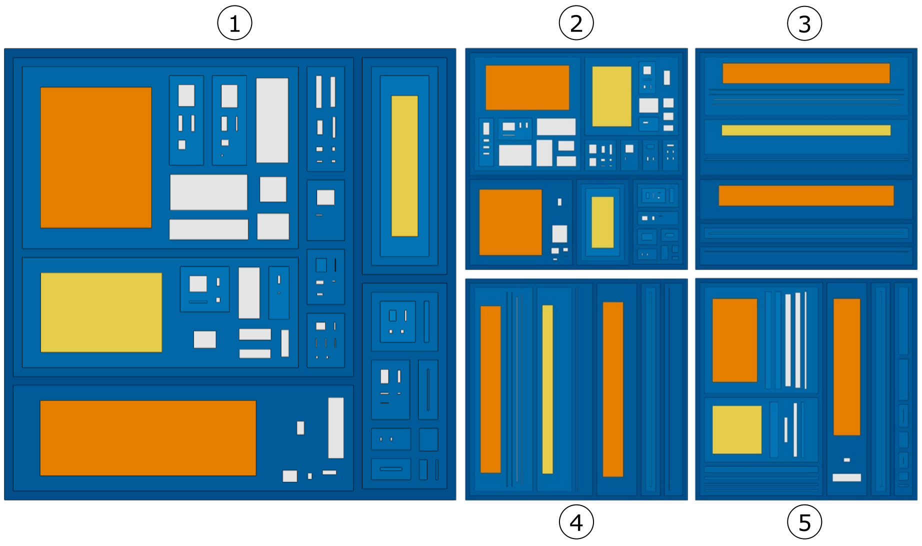


Fig. 6. A memory tree represented with different tiling methods applied during treemapping: (1) squarify, (2) binary, (3) slice, (4) dice, (5) slice & dice.

Ideally, a treemapping algorithm tries to satisfy the criteria of a small aspect ratio (close to one) as it is a metric on how accurate the user can interpret the areas of the rectangles in the treemap [8]. Therefore, the *squarified treemap algorithm* by Bruls et al. [1] seems to be the best choice, as it seeks to produce rectangles as approximate squares (aspect ratio of 1). Nevertheless, Kong et al. [3] argue that squares should be avoided in treemaps. Thus, we use the squarified treemap algorithm with a desired aspect ratio of the *golden ratio* Φ [5] as the default tiling method. This hybrid creates relatively realistic and appealing cities.

Lastly, we use the resulting layout to render 3D objects and build a city visualization with inner nodes representing districts and leaf nodes depicted as buildings.

Evolution Over Time. As already discussed in previous sections, Memory Cities load multiple trees to visualize the heap's evolution over time to inspect the growth of heap object groups. Consequently, buildings in Memory Cities can increase and shrink in size. One might think that it is sufficient to run a treemapping algorithm each time we switch from one tree to another, but this is not the case.

Nodes do not grow at the same rate. In fact, the object/byte count of a heap object group can increase or decrease drastically between two points in time. Besides, it is even possible for nodes to be totally missing during one memory tree because the GC collected all objects of a particular type. As a result, we would have a new arrangement of structures whenever the treemap has changed. Thus, we heavily rely on the stability criteria of tiling methods. It will most likely get tough to track buildings and determine if and which two buildings in different heap states represent the same tree node, making it hard to follow an application's growth.

To overcome the problem of an unstable layout, we apply *static position animation* [4]. That way, each node (i.e., district or building) that might exist at some point gets its reserved space on the general layout. This reserved space is calculated based on each node's *max* object/byte count, in other words, the largest possible area a structure might have during one heap state (i.e., in one of the many memory trees). Therefore, the input of any tiling algorithm is the layout tree presented in Section 3.2, which holds all the max values of all nodes. To visualize a heap state, we calculate the area of a building at this specific point in time and center it in the rectangle that has been reserved for it (**Fig. 7**).

This general layout is computed *once* (a) when the memory city gets initialized (b) every time we switch the scaling mode or tiling method, and (c) during *tree pruning*, which we describe in the following section.

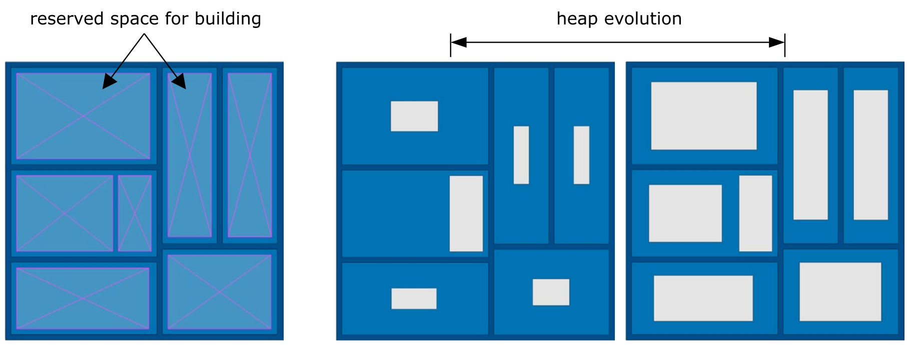


Fig. 7. The general layout (left) acts as guidance on where to position buildings (right) during the heap's evolution without the necessity of constantly computing a new layout when we step back and forth in time.

Tree Pruning. Using the *child count* feature, users can prune the layout tree before it gets passed to a treemapping algorithm. By defining how many child nodes per parent node get visualized, we only reserve space for the N largest heap object groups (districts and buildings) contained in each inner node (districts).

This feature helps to reduce the complexity of significantly broad trees. By dropping small object groups that are not of particular interest to a user (as seen in **Fig. 8**), we emphasize heap object groups that accumulate a considerable amount of objects/bytes over time, which might be the reason behind memory leaks.

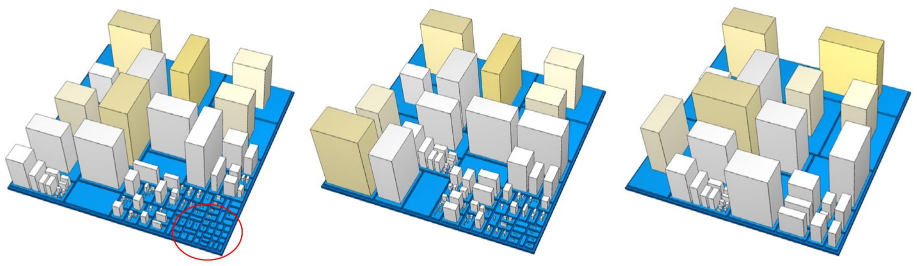


Fig. 8. The same heap state getting restricted in the number of child nodes per parent: 100 (left), 50 (middle) and 20 (right, default setting). For large child count values, some buildings are either too small or not rendered at all (red circle), thus being of no interest.

3.4 Metrics & Visual Mapping

In various other software cities such as CodeCity [13], the visible properties of the city artifacts depict a set of chosen software metrics. This section discusses the visual attributes of memory cities.

Districts. Districts are flat structures with a fixed height that does not encode information. Multiple levels of districts are stacked upon each other, hence visualizing the *underlying memory tree*, specifically the inner nodes. Therefore, the bottommost district represents the entire heap. We use a linear color gradient from dark blue to light blue to ease the task of recognizing a district's level. Lastly, the area of a district is determined by the number of objects or the number of bytes its tree node represents.

Buildings. A leaf node inside a memory tree is visualized as a building. Not only is the building's area based on the object/byte count of the respective node, but also other visual attributes transport information about metrics of interest to the user.

Height. Compared to districts, the height of a building is not fixed, but rather *2 * sqrt(A)* units, if we assume a surface area of A square units. Just like with the golden rectangles from Section 3.3, this results in somewhat realistic proportions of real-world buildings and cities. On the other hand, the downside comes from the fact that both visual attributes are based on the same metric, i.e., either object count or byte count.

As already mentioned by the authors of the original Memory Cities tool [10], trying to separate the object count from the byte count so that one represents the height and the other represents the surface area size did not produce any added value to the visualization. Actually, the contrary was the result, i.e., either extremely narrow buildings that are tall or overly wide buildings that are flat. Thus, the 3D city would suffer visually and be hard to interact with in certain situations (e.g., clicking on narrow buildings).

Even though memory cities avoid having unrealistic buildings, it is common for other software cities, such as CodeCity [13], to contain weird-looking structures.

Color. To build a better perception of the memory evolution, i.e., the growth of heap object groups, memory cities encode the *relative growth* of buildings as color.

The relative growth describes the difference in a node's object/byte count between the first point in time (first memory tree) and the current point in time (current memory tree/currently shown heap state). In comparison, the *absolute growth* of a building is equal to a node's growth between the first and the last memory tree and is stored in `structureInfo`.

We use a linear color gradient ranging from white (negative/no growth) over orange (medium growth) to red (strong growth), which maps a value *alpha* in the range of zero to one to its respective color. To calculate alpha, we divide a building's relative growth by the highest absolute growth of any building in the city, resulting in the formula (with *fullKeyAsString* being the unique identifier of a building):

$$alpha = \frac{current(fullKeyAsString) - first(fullKeyAsString)}{maxAbsoluteGrowth()}$$

Opacity. We are primarily interested in heap object groups that grow substantially over time without dropping in size at the end (i.e., those with tremendous absolute growth), as those are most likely to be involved in a potential memory leak. For this reason, a user can decrease the opacity of less suspicious buildings by utilizing two sliders. The first one allows to select a number of N buildings that should stay solid/fully opaque, specifically, the first N buildings with the highest absolute growth. The second slider sets the reduced level of opaqueness for non-solid buildings (in percent).

Fig. 9 pictures the visual benefits of having the color mode and opacity mode turned on.

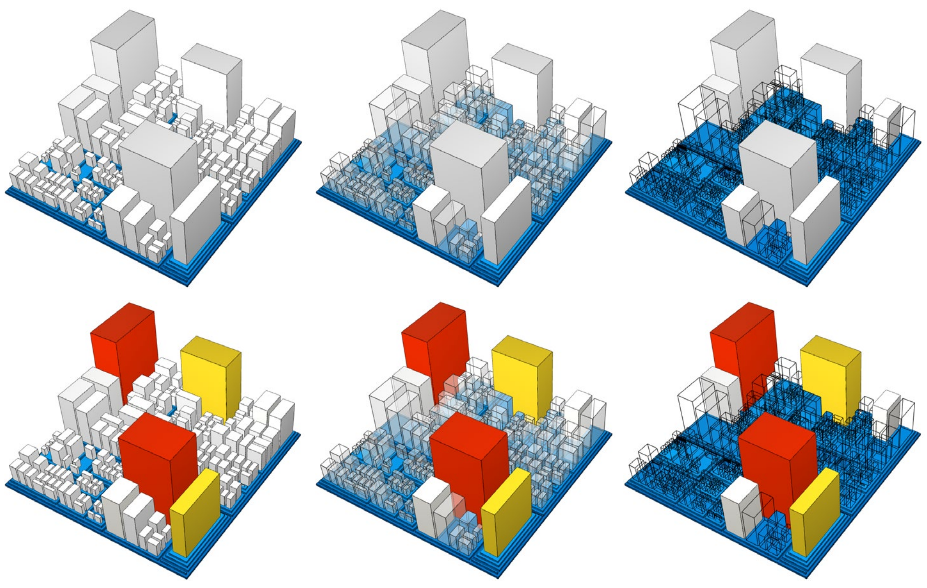
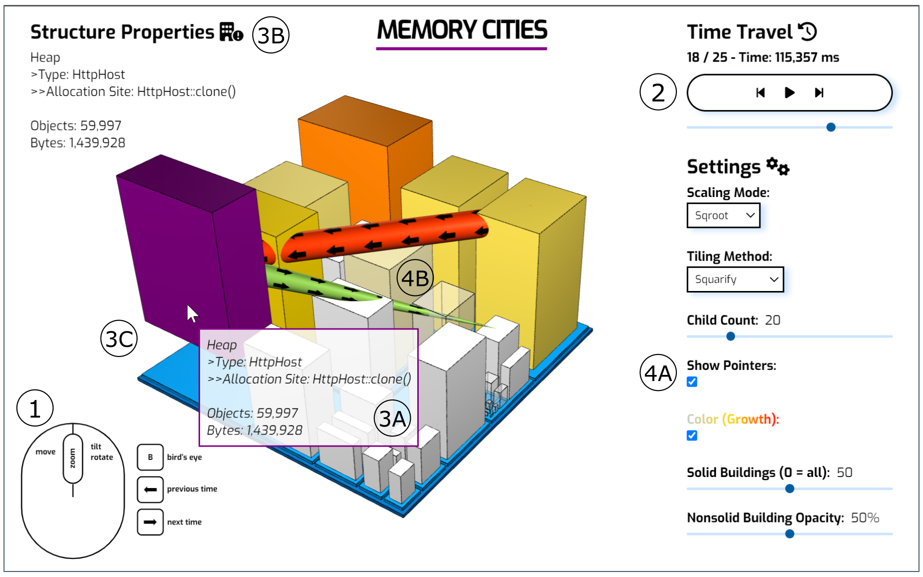


Fig. 9. Six different setting combinations for the same city. The bottommost cities have color mode active. The four rightmost cities have transparency lowered to 50% and 0%, respectively, whereas the seven strongest growing buildings always stay fully opaque.

3.5 Interaction

This section demonstrates all the features and possibilities to interact with memory cities in order to help with the task of finding memory leaks. In **Fig. 10**, one can see numbers assigned to each feature, which are referenced in the following sections.


Structure Properties
Heap
>Type: HttpHost
>>Allocation Site: HttpHost::clone()

Objects: 59,997
Bytes: 1,439,928

MEMORY CITIES

Time Travel
18 / 25 - Time: 115,357 ms

Settings
Scaling Mode: Sqroot
Tiling Method: Squarify
Child Count: 20
Show Pointers: ✓
Color (Growth): ✓
Solid Buildings (0 = all): 50
Nonsolid Building Opacity: 50%

Heap
>Type: HttpHost
>>Allocation Site: HttpHost::clone()

Objects: 59,997
Bytes: 1,439,928

Fig. 10. Building that performs a jump animation right after selection (purple). A structure stays highlighted until the user clicks another one or an empty space.

Navigation. There are several ways of navigating through the city (**Fig. 10** (1)). The camera can (a) be moved using the left mouse button, (b) be tilted by pressing the right mouse button and moving the mouse up or down (though it cannot be tilted below the city), (c) be rotated by pressing the right mouse button and moving the mouse left or right and (d) zoom in and out using the mouse wheel. Furthermore, we set a damping factor (inertia) to give a sense of weight to the controls. Additionally, pressing the B key positions the camera above the city, resulting in a *bird's eye view* (as seen in **Fig. 11**).

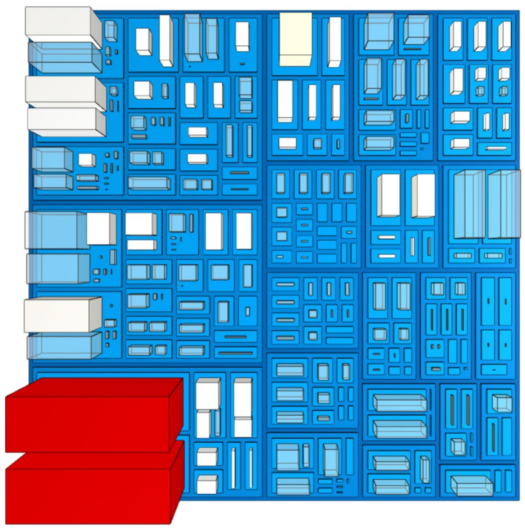


Fig. 11. By pressing the keyboard shortcut B, we can transpose the camera into a bird's eye view.

Time Travel. Time Travel is the central feature of the Memory Cities tool, which is a unique component compared to other memory monitoring tools. Inspired by Time Travel in [14], this feature allows users to step back and forth through the history of an application's heap memory. At the same time, the city updates itself to reflect the currently displayed heap state/memory tree.

Between two timestamps, a node's object/byte count value can increase or decrease, meaning that the size of the respective building also changes. To enhance the visual representation of this change in size, we animate a building's shrinking or expansion process, i.e., animating the *scaling* of the 3D object (instead of instantly rendering the new-sized building). See Section 5.3 for further details. Note that districts are static and do not transform in size once they are created, since, although they are a group of heap objects, their primary purpose is to represent the underlying memory tree of the final/uppermost heap object groups.

Users can switch between the shown memory tree using buttons, a slider (**Fig. 10** (2)), or the keyboard's left and right arrow keys. In addition to this, the play button starts an automatic animation of the heap's evolution, meaning that every heap state is displayed for one second before rendering the next one automatically. It is also possible to pause and restart this procedure at any timestamp.

Structure Properties. Each structure (i.e., district or building) holds three types of information: (a) the path from the root node of the memory tree to this structure's node (e.g., Heap → Type: String → Allocation Site: toString()), (b) the number of objects allocated by this group, (c) the size of all objects in bytes. A user can access this information by hovering over a structure to pop up a tooltip (**Fig. 10** (3A)). Moreover, a click on a structure highlights it, meaning that we display the information in the upper left corner (3B) and change the structure's color to purple to point it further out. A structure stays selected even during Time Travel, making tracking its evolution easier. On top of that, clicking on a structure causes it to do a one-second jump animation (as demonstrated in **Fig. 10** (3C)). Supposing the selected structure is a district, all its child structures perform the same animation (Section 5.4).

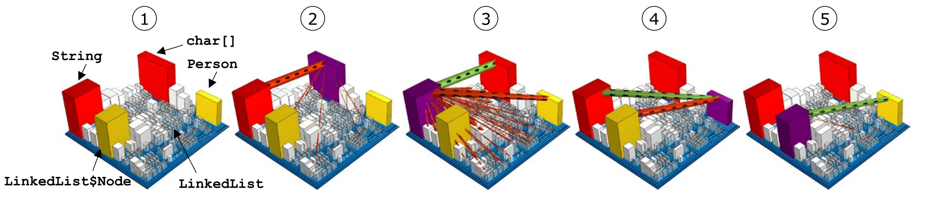
Heap Object References. Ticking the *Show Pointers* checkbox (**Fig. 10** (4A)) draws frustums[8] (4B) between the currently highlighted building and others, revealing *references* among these heap object groups. This novel feature helps differentiate between a memory leak's *symptoms* and the *root cause*. Memory leaks usually happen when some objects (root cause) keep numerous references to other objects (symptoms), even after the latter is no longer needed and only takes up space in memory.

Per memory tree, we also import one *points-to map* and one *pointed-from map* (also as JSON files). For each heap state and structure, respectively, they store how many objects the respective building references in other buildings (points-to) and by how many objects of the other buildings it is referenced by (pointed-from).

The example in **Fig. 12** gives a better understanding of the feature and its usefulness for memory analysts. (1) shows a memory city where heap objects are grouped by package (districts) and type (buildings). Two red buildings can be observed, indicating the highest absolute growth, meaning that the application accumulates lots of `char[]` and `String` objects over time. (2) Clicking on the `char[]` building shows that nearly all its instances are referenced by `String` objects (thick orange frustum). (3) Unsurprisingly, plenty of types contain strings. However, most references come from the `Person` type, which is selected in (4). The person's building only has one outgoing (green) frustum to `String`, along with one (orange) frustum coming from `LinkedList$Node`, signifying that all persons are part of a linked list. In (5), one can notice a thin orange frustum from `LinkedList$Node` to `LinkedList`, hinting that the list head is kept alive by a `LinkedList`.

After inspecting the references, we can conclude that the root cause of the memory leak is a linked list where multiple persons are added without being removed afterwards. The vast amount of `String` and therefore `char[]` objects is only a consequence and symptom of the leak.

[8] A 3-dimensional solid shape formed by cutting a cone or pyramid from the top with a plane parallel to its base.



```
                 "char[]": {             "String": {            "Person": {              "LinkedList$Node": {
                     "String": 22215,        "Person": 10000        "LinkedList$Node":       "LinkedList": 1
pointed-from:        ...                     ...                    10000                }
                 }                       }                      }
                 ---------------------------------------------------------------------------------------------
points-to:       "char[]": {}            "String": {            "Person": {              "LinkedList$Node": {
                                             "char[]": 22215,       "String": 10000,         "Person": 10000,
                                         }                      }                        }
```

Fig. 12. Heap object reference analysis. Incoming references are colored orange and stored in pointed-from maps; outgoing references are colored green and stored in points-to maps. These maps are saved as JSON files in separate folders and imported together with memory trees.

Section 5.5 describes the creation and geometry of a frustum and how it is sized so that a frustum's thickness can visually indicate uneven object amounts.

3.6 Summary of Main Differences to Original Version

The primary difference between the original and new tool is simultaneously the main goal of this bachelor thesis: porting the Memory Cities desktop application (built in Unity using C#) to a web-based visualization tool. The fundamental characteristics of the new tool are the same, as it is a requirement to support existing desktop application features. Nonetheless, the following list sums up more distinctions:

- The new tool is implemented as a web app using HTML, CSS and JS, plus the library Three.js being at the core with regard to rendering 3D structures such as districts and buildings.
- Any modern browser is capable of running online Memory Cities, i.e., no installation is needed when compared to applications developed in Unity.
- Data processing varies heavily between the two versions. The authors of the original tool implemented a tree-like data structure based on a `Node` class. Each `Node` stores its heap object group's properties (e.g., name, level, byte count), including a `List<Node>` attribute that references a node's children. Moreover, even metadata gets stored in trees (max tree and growth tree). On the other hand, the online tool utilizes a single JavaScript object (`structureInfo`) that acts as a container for all properties and metadata regarding a heap object group. `structureInfo` maps a heap object group's name to multiple properties in the form of *key:value* pairs. To access a structure's metadata, one can use the syntax: `structureInfo.districtName.max` or `structureInfo.buildingName.absG`.
- The possibility to adjust the tiling method used during treemapping, i.e., the layout process, is a new feature.

- Animations to fluently transform from one timestamp to another and during structure selection are a nice-to-have extension.
- Lastly, during the development of the online version, a considerable amount of time was put into a modern design and appearance of the user interface with the help of CSS and JS.

4 Three.js

To draw 3D on the web, one might use *WebGL (Web Graphics Library)*, a very complex and low-level system that only draws points, lines and triangles using JavaScript. Making use of WebGL requires a deep understanding of computer graphics, mathematics and geometry. It is a powerful library but with a steep learning curve. Three.js tackles these hurdles and provides built-in features for elements such as scenes, lighting, shadows, controls, textures and 3D math. Thus, it is a 3D library written in JavaScript with WebGL under the hood that allows developers to get 3D content on a webpage more easily.

4.1 Fundamentals

Since Three.js is at the core of the new Memory Cities tool, we introduce Three.js fundamentals throughout Section 4 before we explore the implementation details of the tool in Section 5. We also briefly mention the Three.js components, which are directly used in Memory Cities.

Scene Graph. At the core of each Three.js app, there is a scene graph. The scene graph is a tree-like structure consisting of all the 3D structures (so-called `Mesh`, `Group`, or simply `Object3D` objects) and lights visible on the screen. An example of such a scene graph can be seen in **Fig. 13** inside the blue rectangle (all parts in this figure will be explained in the subsequent sections). Because children are positioned and oriented relative to their parents, this hierarchical parent/child tree-like structure defines where objects are located and how they are oriented. For instance, an arm object may be a child of a human object; affecting the human's location and orientation automatically alters the arm's position in 3D space.

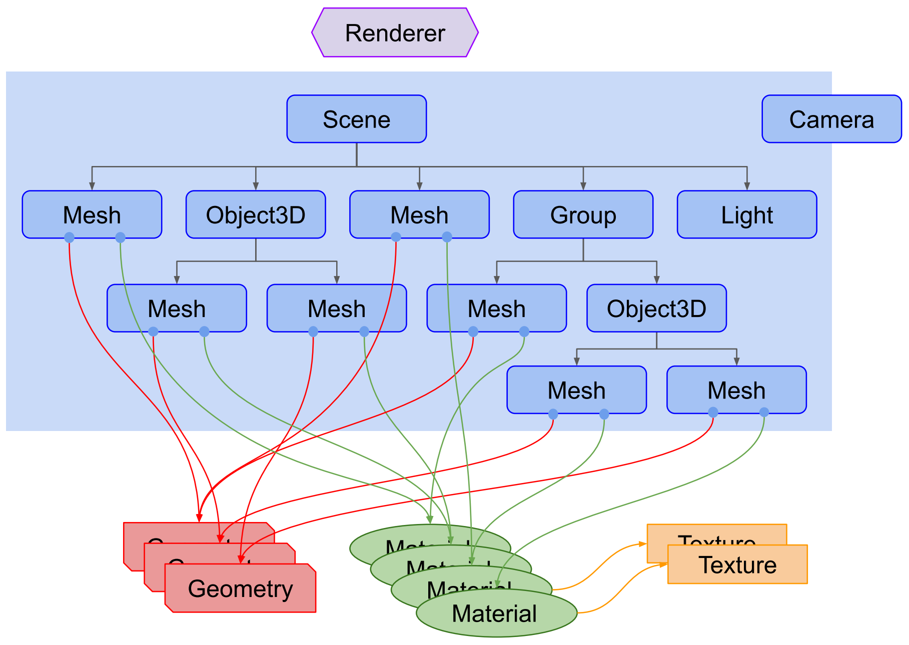
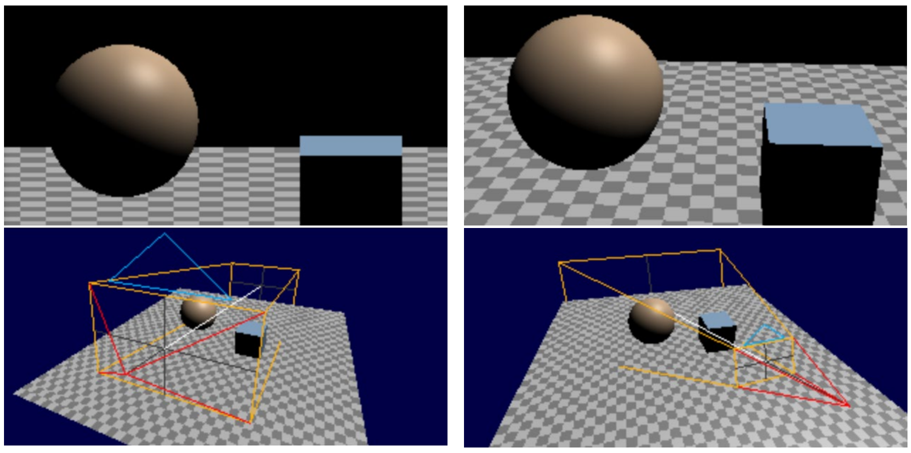


Fig. 13. Exemplary scene graph from the official Three.js documentation[9]. Geometries, materials and textures are not part of the scene graph and can be shared across multiple `Mesh` objects.

The Scene. At the root of each scene graph, there is a `Scene` object. It holds additional information such as background (color or texture) and fog. All instances of `Object3D` (including `Mesh`, `Group`, and `Light` objects, as `Object3D` is the base class for nearly every object) need to be added to the scene using the `add()` method. The `Scene` class also extends the `Object3D` class.

Camera. A `Camera` specifies which portion of the scene is exposed to the screen later. There are numerous cameras in Three.js, the most commonly used ones being the `OrthographicCamera` and the `PerspectiveCamera`. A comparison of the two can be examined in **Fig. 14**.

[9] https://threejs.org/manual/#en/fundamentals



```
new THREE.OrthographicCamera(
        sizes.width / -2,
        sizes.width / 2,
        sizes.height / 2,
        sizes.height / -2,
        CAM_NEAR, CAM_FAR
);
```

```
new THREE.PerspectiveCamera(
        CAM_FOV,
        sizes.width / sizes.height,
        CAM_NEAR, CAM_FAR
);
```

Fig. 14. Orthographic camera (left) and perspective camera (right). Rather than specifying a frustum, an orthographic camera creates a cuboid (bottom left) and thus has no perspective (top left).

For 2D games, one would use an orthographic camera as its missing depth information and perspective.

In contrast, a perspective camera gives the feeling of a 3D space where things in the distance appear smaller than things up close (i.e., it mimics how humans see). It does this by defining a frustum (**Fig. 15**) based on four properties, which are also the parameters for the `PerspectiveCamera` constructor. *Near* and *far* set the distance between the camera and the front face and back face of its frustum, respectively. The field of view (*fov*, which is in degrees) limits how much of the scene is visible based on a full 360 degrees (i.e., it indirectly defines the height of the front and back of the frustum). Lastly, we got the *aspect* ratio, which is based on the user's browser window and calculated by dividing the `window.innerWidth` by the `window.innerHeight`. Anything outside this viewing frustum gets clipped (i.e., not rendered).

Finally, in **Fig. 13**, notice how the camera is half in and half out of the scene graph. The reason for this is that cameras do not have to be in the scene graph in order to operate. However, if they are the child of some object, they will move and orient relative to their parent object (e.g., first-person games).

Because we want to display a 3D city, we utilise a perspective camera.

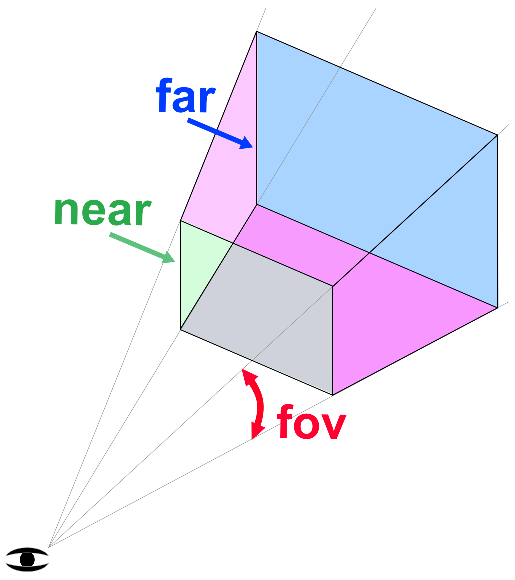


Fig. 15. Because of their viewing frustum, perspective cameras give information about depth.

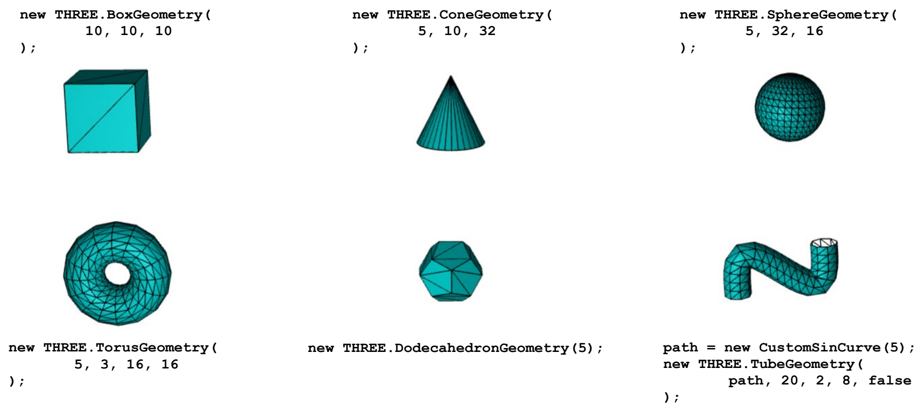
Renderer. The last and arguably the main fundamental object of every Three.js app is a `Renderer`, specifically, the `WebGLRenderer` in the case of Memory Cities (and most other use cases). We pass a `Scene` and a `Camera` to the `render()` method for the purpose of rendering (drawing) the portion of the 3D scene that is inside the frustum of the camera as a 2D image to a canvas. The canvas is an `<canvas>` element inside the app's HTML file and corresponds to the `domElement` property of our renderer. Other valuable properties of renderers include `antialias` (boolean that says whether to perform antialiasing or not) and `info` (an object with statistical information about the graphics board memory and the rendering process, see Section 5.6 **Fig. 22**).

4.2 Geometry + Material = Mesh

In order to create 3D objects and structures, we create a `Mesh`. A `Mesh` symbolizes the combination of three things:

- a geometry (the "skeleton" of our 3D object)
- a material (the "skin" of an object)
- the position, orientation, and scale of that object in the scene relative to its parent object.

Geometry. A geometry defines the vertices and faces of a 3D shape. Three.js provides numerous predefined primitives, including `BoxGeometry`, `ConeGeometry`, `Cyl-inderGeometry`, `SphereGeometry`, `TorusGeometry`, TetrahedronGeometry, OctahedronGeometry, and many more. As shown in **Fig. 16**, each geometry has its own set of parameters needed to construct the shape (e.g., a BoxGeometry needs width, height and depth; a SphereGeometry requires its radius, the number of horizontal segments and the number of vertical segments). For the majority of 3D apps it is more likely to have 3D models made in 3D modelling programs such as Blender.



```
new THREE.BoxGeometry(
        10, 10, 10
);
```

```
new THREE.ConeGeometry(
        5, 10, 32
);
```

```
new THREE.SphereGeometry(
        5, 32, 16
);
```

```
new THREE.TorusGeometry(
        5, 3, 16, 16
);
```

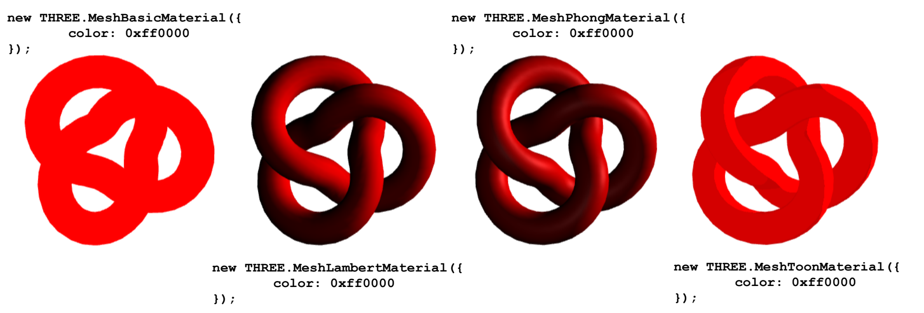
```
new THREE.DodecahedronGeometry(5);
```

```
path = new CustomSinCurve(5);
new THREE.TubeGeometry(
        path, 20, 2, 8, false
);
```

Fig. 16. Subset of available geometries in Three.js. Every surface consists of vertices that form *triangles*.

Material. A Material is responsible for an object's appearance. It tries to simulate light sources in the scene whose light rays hit its object (so-called *shading*). Three.js comes with a wide variety of materials. Depending on the use case, developers must decide which material they apply to their meshes. To achieve more realistic graphics, one might use more sophisticated materials. On the other hand, those require more GPU power and might be unsuitable for mobile phones. Without going into too many details, here is a list of predefined materials (**Fig. 17**):

- MeshBasicMaterial: unaffected by lights
- MeshLambertMaterial: computes lighting only at the vertices
- MeshPhongMaterial: computes lighting at every pixel; supports *specular highlights* (through the shininess parameter)
- MeshToonMaterial: based on MeshPhongMaterial, but drops the smooth gradient effect



```
new THREE.MeshBasicMaterial({
        color: 0xff0000
});
```

```
new THREE.MeshPhongMaterial({
        color: 0xff0000
});
```

```
new THREE.MeshLambertMaterial({
        color: 0xff0000
});
```

```
new THREE.MeshToonMaterial({
        color: 0xff0000
});
```

Fig. 17. Materials define how items appear in the scene. More complex and realistic materials work with extra properties such as shininess and roughness.

Additionally, there is the `MeshStandardMaterial` and the `MeshPhysical-Material`, two *physically based rendering (PBR) materials*, which use much more complex math to come close to real-world looks. They equip properties such as `metalness`, `roughness`, and `clearcoat`.

Texture. Materials not only allow us to add colors to our 3D object but also textures. `Texture` objects generally represent images either loaded from image files, generated from a canvas or rendered from another scene (e.g. MiniMaps in games: you could point a camera onto a scene, take the generated image and use it as a texture). To implement a texture, we first need to create a `TextureLoader` object. Secondly, we call its `load()` method with the URL of an image, which gives us a `Texture` object as a result. Finally, we set the material's `map` property to one or more textures. See Section 5.5 to see where and how textures are used in Memory Cities. It is worth mentioning that it is possible to wait for textures to load, which may be helpful for large amounts of data.

Composition. In the last step of producing a 3D structure, we create a new `Mesh` object and pass geometry and material as parameters to the constructor. Further, one might adapt its `position` and `rotation` properties by each axis (e.g., during animation) to change the mesh's location and orientation within the scene. The rotation is in radians, i.e., to work with degrees, one has to divide the degree value by 180 and then multiply by π. Finally, the complete `Mesh` object needs to be added to the scene. When it is part of a bigger structure, it gets added to a `Group` or `Object3D` object instead, just like in **Fig. 13**.

4.3 Miscellaneous

Light. Without appropriate lighting, a mesh and its material, respectively, would remain dark/black (except for `MeshBasicMaterial`). The next list gives an overview of all the different kinds of `Light` objects available in Three.js (note that every type of light has its own set of parameters, except for `intensity`, which can be set for all the classes):

- `AmbientLight`: does not really accomplish the task of "real lighting" as there is no light source. Basically, it is used to brighten up the scene, making the darks not too dark.
- `HemisphereLight`: works the same as `AmbientLight` and adds a parameter for the sky and ground color - the sky color if the object's surface is pointing up and the ground color if the object's surface is pointing down.
- `DirectionalLight`: will shine in the direction of its `target` (property of type `Object3D`), which needs to be added to the scene. There exists no "point" the light comes from, i.e., it is an infinite plane of light shooting out parallel rays of light (often used to mimic a sun).
- `PointLight`: works like a light bulb. The light source shoots light in all directions from its location. There is also a `distance` and `decay` property to set the range and dim factor. In Memory Cities, a combination of three `PointLight` instances is positioned above the city.
- `SpotLight`: works like a flashlight. The light source shoots light in the direction of its `target` and only shines inside a cone whose size can be adjusted with the `angle` property. The `penumbra` property regulates the light's fade towards the edge of the cone.
- `RectAreaLight`: a rectangular area of light such as a linear fluorescent light; works only with the two PBR materials

To conclude, a scene can include various combinations of `Light` objects. Moreover, they can be not only added to a `Scene` but also meshes and `Group` instances (e.g., two `SpotLights` as headlights of a car object). For a visual representation of the numerous light variations, visit the Three.js manual[10].

Controls. To navigate through Memory Cities, we implement `MapControls`, which allow us to orbit around a target (i.e., the city). We also set the `maxPolarAngle` to 90 degrees, limiting how far one can orbit vertically, meaning that we cannot peek below the memory city.

5 Implementation

This section gives insights into some implementation details of the new Memory Cities tool. It provides complementary information with regard to Section 4 and elaborates on previously mentioned topics, including project structure, treemapping, city rendering, structure animations, structure highlighting, textures, and disposing of 3D objects.

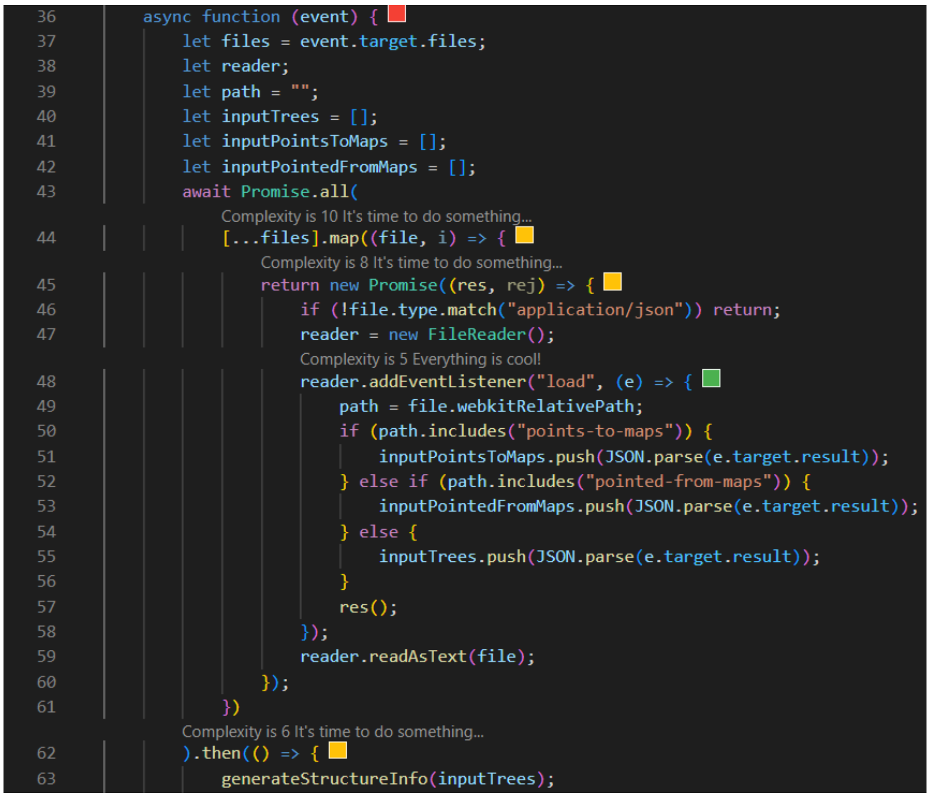
[10] https://threejs.org/manual/#en/lights

5.1 Code Base

Besides *index.html*, which handles the import of needed JavaScript for treemapping, animations and Three.js and which incorporates the `<canvas>` element to which we render memory cities, the code base mainly exists of three files:

- *controls.js*: In this file, we attach event handlers to elements on the right-hand side of the tool, i.e., buttons, sliders, checkboxes and dropdown menus that are part of the Time Travel and the settings category, respectively. Checkboxes and buttons listen to *click* events, sliders to *input* events and dropdown menus to *change* events. The sliders of the Time Travel and the child count feature actually listen to *change* events since we only want to trigger the respective methods *after* dragging the sliders to their desired position in order to avoid excessive render updates of the city (listening to *input* means that for every step during the slide we would generate a new city which is not particularly useful nor making changes between different timestamps/memory trees visually observable). The key presses which trigger bird's eye and Time Travel are also defined here. All this interaction with the UI calls functions defined in the next two files.
- *visualizer.js*: All code regarding Three.js and 3D visualization is covered in visualizer.js. At the beginning of the file, we construct a `Scene` object, to which we `add()` three `PointLight` instances and three `Group` objects, one each for all the districts, buildings and frustums. This approach gives us better control of all the 3D structures (e.g., when trying to update the opacity of all buildings, we get all the meshes referenced in the `children` property (array) of the buildings' `Group` and set the opacity property of a mesh's respective material). Not only do we set up a renderer and camera, but also an event handler for `window`, which listens to *resize* events as a means to adapt the camera's aspect ratio and renderer's pixel ratio. Some other methods inside this file are explained between Section 5.3 and Section 5.6.
- *logic.js*: This file is the entry point of our web app. After selecting the folder that holds all the JSON files (memory trees), we wait for multiple `FileReader` objects to finish their asynchronous task of reading the input trees (as seen in **Fig. 18**). This can be accomplished by wrapping each `FileReader` (one per JSON file) inside a `Promise` and having `await Promise.all(<array of promises regarding file readers>)` resolve when all of the input's promises have been fulfilled. During that process, all input trees are stored in an array. Afterwards, the `structureInfo` object is built upon this array. `structureInfo` maps a node's *fullKeyAsString* to its properties (e.g., classifiers, *object/byte count per timestamp*, max/first/last object/byte count, level, children, coordinates inside the city layout). With all that data, `rebuildEntireCity()` builds a new city from scratch (i.e., destroying a possible old city by executing `resetScene()` (part of visualizer.js), calling the treemap layout function (next section), generating city districts and calling `updateUIAndCity()`). The file's `updateUIAndCity()` method is responsible for generating and updating

buildings during Time Travel, the color & opacity of buildings and the values and structure properties that are part of the UI.



```
async function (event) {
    let files = event.target.files;
    let reader;
    let path = "";
    let inputTrees = [];
    let inputPointsToMaps = [];
    let inputPointedFromMaps = [];
    await Promise.all(
        Complexity is 10 It's time to do something...
        [...files].map((file, i) => {
            Complexity is 8 It's time to do something...
            return new Promise((res, rej) => {
                if (!file.type.match("application/json")) return;
                reader = new FileReader();
                Complexity is 5 Everything is cool!
                reader.addEventListener("load", (e) => {
                    path = file.webkitRelativePath;
                    if (path.includes("points-to-maps")) {
                        inputPointsToMaps.push(JSON.parse(e.target.result));
                    } else if (path.includes("pointed-from-maps")) {
                        inputPointedFromMaps.push(JSON.parse(e.target.result));
                    } else {
                        inputTrees.push(JSON.parse(e.target.result));
                    }
                    res();
                });
                reader.readAsText(file);
            });
        })
    Complexity is 6 It's time to do something...
    ).then(() => {
        generateStructureInfo(inputTrees);
```

Fig. 18. The `Promise` object symbolizes the eventual completion (or failure) of an asynchronous operation (reading memory trees as JSON files) and its resulting value.

5.2 Layout

With the help of *D3.js*[11] (short for *Data-Driven Documents*), we are able to compute the rectangles of a treemap, i.e., the general city layout of a memory city. D3 is a JavaScript library for producing dynamic, interactive data visualizations for the web.

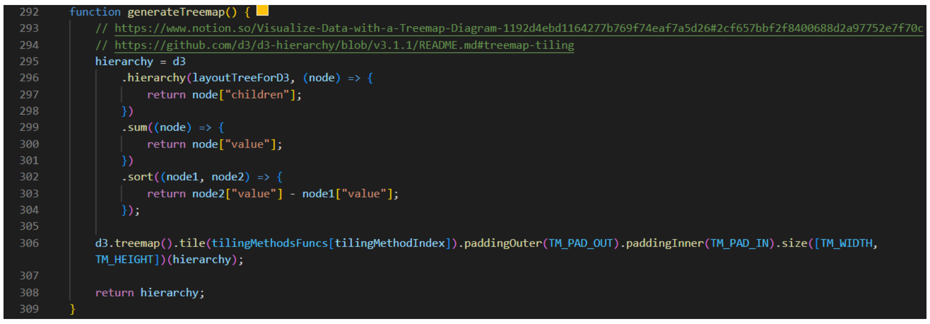
In previous sections, we explained the concept of a layout tree (holding all the max object/byte counts stored in `structureInfo`) being passed to a treemap layout function. This is only partially true, as we will now explain the code snippet from **Fig. 19**.

- Firstly, we create the object `layoutTreeForD3`, which is a simplified memory tree where the only two properties of a node are the `fullKeyAsString` and (a) for inner nodes/districts an array called `children` (b) for leaf

[11] https://d3js.org/

nodes/buildings the max `value` extracted from `structureInfo`. Why not take the max value of a heap object group represented as a district out of `structureInfo`? Because this value does NOT match the object/byte count on which a district's area can be determined (but rather the maximum size of the underlying heap object group throughout all memory trees).

- Secondly, we pass the `layoutTreeForD3` to the `d3.hierarchy()` method and tell it where the child nodes are (`children` array).
- We then call the `sum()` method on this *hierarchy model* which determines the area of the inner node's rectangles (i.e., the space reserved of and for districts) by summing up all their children's `values`, starting from the leaf nodes' max values. This adds the `value` property to the inner nodes of the hierarchy model.
- The `sort()` method orders our data by comparing two nodes' values.
- For the last step, we call the `d3.treemap()` function on this hierarchy and specify the tiling method, padding between rectangles and overall size of the map to generate the dimensions of our rectangles.



```
function generateTreemap() {
    // https://www.notion.so/Visualize-Data-with-a-Treemap-Diagram-1192d4ebd1164277b769f74eaf7a5d26#2cf657bbf2f8400688d2a97752e7f70c
    // https://github.com/d3/d3-hierarchy/blob/v3.1.1/README.md#treemap-tiling
    hierarchy = d3
        .hierarchy(layoutTreeForD3, (node) => {
            return node["children"];
        })
        .sum((node) => {
            return node["value"];
        })
        .sort((node1, node2) => {
            return node2["value"] - node1["value"];
        });

    d3.treemap().tile(tilingMethodsFuncs[tilingMethodIndex]).paddingOuter(TM_PAD_OUT).paddingInner(TM_PAD_IN).size([TM_WIDTH,
    TM_HEIGHT])(hierarchy);

    return hierarchy;
}
```

Fig. 19. D3 supports several treemap tiling methods (Section 3.3).

The dimensions (i.e., the location/base area of our districts and buildings) are stored in the resulting hierarchy object, from which we extract the properties `x0`, `x1`, `y0`, `y1` and store them in `structureInfo`.

5.3 Buildings

`generateCityBuildings()` gets called each time a new memory tree/heap state gets displayed, i.e., during Time Travel. Inside the method, we iterate through all the leaf nodes of the currently shown memory tree. The sequence of events inside this loop is as follows:

1. If the heap object group does not have a respective building, we create the 3D structure. The corresponding scene graph is depicted in **Fig. 20**. Firstly, we make a `BoxGeometry` (with a *uniform edge length of 1*, i.e., a cube) and a `MeshPhongMaterial` for them to merge to a `Mesh`. Because we want seeable edges for each cuboid, we also create an `EdgesGeometry` and pass the

previously created `BoxGeometry` to the constructor. The resulting geometry object basically holds the pairs of vertices which form a cuboid's wireframe and is given to a `LineSegments` instance (in addition to a `LineBasicMaterial`) that draws a series of lines between pairs of vertices. Lastly, the `LineSegments` object gets attached to the building's `Mesh`, which on the contrary, is appended to a `Group`'s children array that holds all the buildings inside memory cities.

2. On creation, a cuboid's `position(.x/.y/.z)` inside the world of memory cities always corresponds to its center of mass, with each edge parallel to one of the axes. That is why we set the building's `position.x` and `position.z` to half of the width and depth of the layout's reserved rectangle, i.e., *x0 + (x1 - x0)* and *y0 + (y1 - y0),* respectively. That is how buildings are centered in the layout spots reserved for them. For clarity, it is worth mentioning that D3.js works in 2D and not 3D like Three.js. Hence, D3's y-axis is congruent with the z-axis inside the Three.js space.
3. The base area's size is based on the layout spot reserved for the building (i.e., max area; *x * y*) and the current object/byte count, resulting in the following formula:

$$newArea = (x1 - x0) * (y1 - y0) * \frac{current(fullKeyAsString)}{\max(fullKeyAsString)}$$

   Mathematical relations lead to the building's width *w*, depth *d* and height *h*.
4. Having a cube/`BoxGeometry` with a length of 1 is beneficial, as we can now set the `scale.x`, `scale.y` and `scale.z` of our `Mesh` to *w*, *h* and *d.* Not only is it *not* possible to reshape a geometry after creation, but we can also reuse a building and scale it each time we switch the displayed memory tree (during Time Travel), with the edges/line segments being rescaled automatically too.

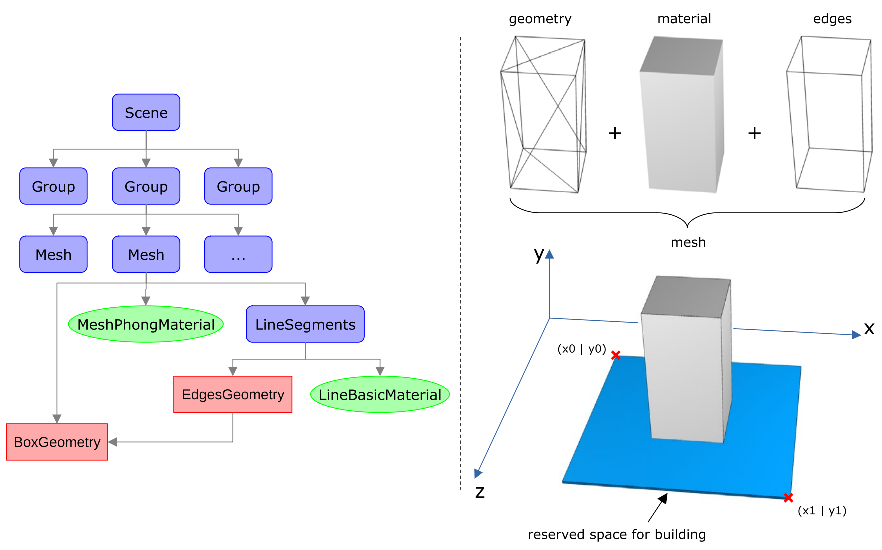


Fig. 20. A building's `Mesh` is assembled with three components (right), resulting in the left scene graph.

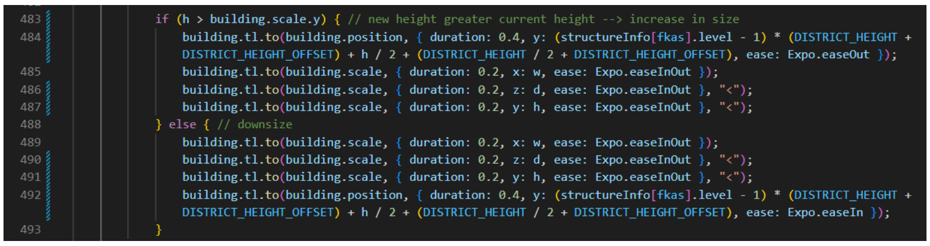
Districts are built the same way, with the only differences being that their height is always one, and they are static (no rescaling).

5.4 Animations

A heap object group varies in object/byte count through several snapshots/memory trees, embodied via shrinking and expanding buildings. This rescaling of buildings is further elevated with the help of the *GreenSock Animation Platform (GSAP)*.
GSAP[12] is a powerful JavaScript library that enables developers to create robust time-line-based animations. During a building's construction, we append a `Timeline`[13] property to its `Mesh` object. A Timeline is a powerful sequencing tool that acts as a container for `Tween`[14] instances. A Tween is that part which performs the animations. It expects a target (the desired object to be animated, e.g., CSS properties, SVG), a duration, and any numeric object properties you want to animate. When the Tween's playhead moves to a new position, it figures out what the property values should be at that point and applies them accordingly.
In the if-statement from **Fig. 21**, one can see four Tweens being added `to()` the Time-line `tl` of the building if it needs to expand (higher object/byte count compared to the previous memory tree). First, in 0.4 seconds, the 3D structure rises to its new `position.y`, which is half its new height (*h / 2*) plus some offset for underneath districts.

[12] https://greensock.com/gsap/
[13] https://greensock.com/docs/v3/GSAP/Timeline
[14] https://greensock.com/docs/v3/GSAP/Tween

Afterwards, it takes the mesh 0.2 seconds to expand along each axis by setting the scale property to the new width *w*, depth *d* and height *h*.
When the object/byte count decreases, the order of Tweens switches, i.e., the building shrinks and then sinks.



```
if (h > building.scale.y) { // new height greater current height --> increase in size
    building.tl.to(building.position, { duration: 0.4, y: (structureInfo[fkas].level - 1) * (DISTRICT_HEIGHT +
    DISTRICT_HEIGHT_OFFSET) + h / 2 + (DISTRICT_HEIGHT / 2 + DISTRICT_HEIGHT_OFFSET), ease: Expo.easeOut });
    building.tl.to(building.scale, { duration: 0.2, x: w, ease: Expo.easeInOut });
    building.tl.to(building.scale, { duration: 0.2, z: d, ease: Expo.easeInOut }, "<");
    building.tl.to(building.scale, { duration: 0.2, y: h, ease: Expo.easeInOut }, "<");
} else { // downsize
    building.tl.to(building.scale, { duration: 0.2, x: w, ease: Expo.easeInOut });
    building.tl.to(building.scale, { duration: 0.2, z: d, ease: Expo.easeInOut }, "<");
    building.tl.to(building.scale, { duration: 0.2, y: h, ease: Expo.easeInOut }, "<");
    building.tl.to(building.position, { duration: 0.4, y: (structureInfo[fkas].level - 1) * (DISTRICT_HEIGHT +
    DISTRICT_HEIGHT_OFFSET) + h / 2 + (DISTRICT_HEIGHT / 2 + DISTRICT_HEIGHT_OFFSET), ease: Expo.easeIn });
}
```

Fig. 21. Easing adapts the animation's timing, changing its behavior and nature, respectively (e.g., `Expo.easeOut`: Tween starts fast and ends gradually/slow). "<" puts the animation at the start of the prior animation, meaning that they trigger at the same instance.

As mentioned in **Structure Properties**, clicking on a structure adds two Tweens, placing the mesh's `position.y` higher, then lower, and therefore letting the structure(s) jump.

5.5 Heap Object References & Frustums

Two types of geometries are used in Memory Cities: `BoxGeometry` for districts and buildings and `CylinderGeometry` for frustums representing heap object references between two buildings (**Heap Object References**). This section covers these frustums in more detail.
The thickness of the `CylinderGeometry` (placed between the roofs of the two buildings) depends on the bottom and top radius. Assume building A objects reference building B objects. The formula for calculating the start radius (from roof A) then looks as follows:

$$rBottom = \frac{buildingA.scale.y}{2} * 0{,}175 * \frac{pointedFromMaps[fullKeyAsStringB][fullKeyAsStringA]}{current(fullKeyAsStringA)}$$

In other words, the bottom radius is based on half the building's height, some downsizing factor and the quotient obtained by dividing the number of objects that reference building B with the overall object count of building A. Differing radii transform the initial cylinder into a frustum.
Part of the frustum's `Mesh` is a green or orange `MeshPhongMaterial` that has an arrow `Texture` (.jpg file loaded with `TextureLoader`) applied to it to indicate the reference direction. The arrow is repeated across the surface horizontally (4x) and vertically (*floor(sqrt(geometryHeight))*). The texture's corresponding `wrap` parameter needs to be set to `THREE.RepeatWrapping` with the aim of achieving this desired tiling effect.

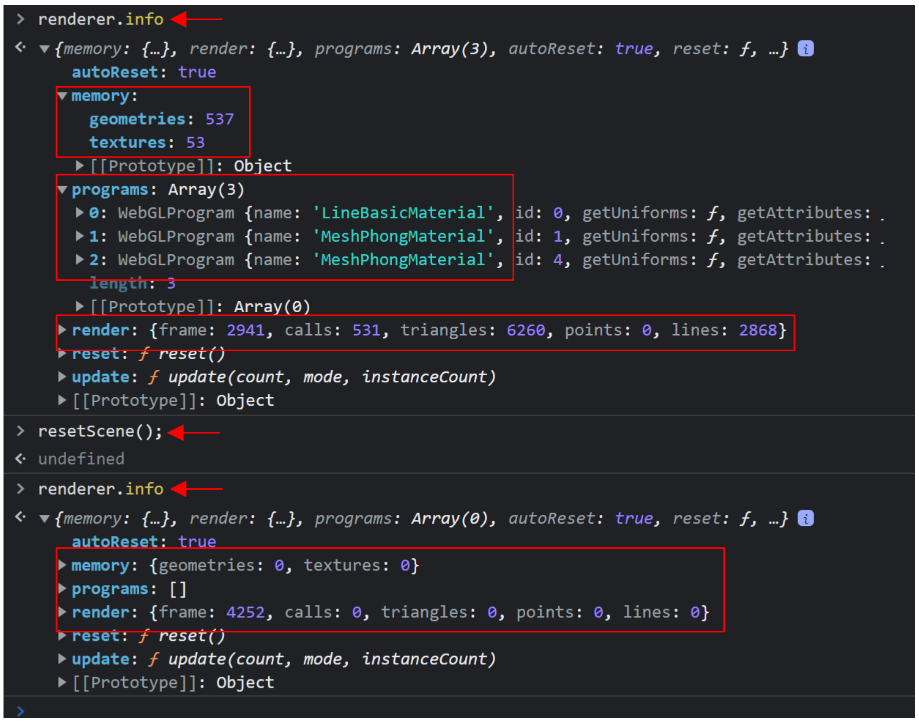
Before adding the mesh to the scene, we involve vector calculation to determine the mesh's `position` and rotation matrix (`quaternion`[15]).

5.6 Disposing

It would be ironic to develop an online tool for the detection of memory leaks that causes memory leaks in the browser itself. That is why we implemented the method `resetScene()`, which is responsible for the disposal of unused Three.js entities, specifically, WebGL-related entities. When creating objects such as geometries, materials and textures, Three.js internally creates objects of the type `WebGLBuffer`, `WebGLProgram` and `WebGLTexture`. These are not automatically released for garbage collection but instead after calling the `dispose()` method.
As part of `resetScene()`, we go through the scene graph and check for every child if it has a geometry, material or texture attached to it. That is indeed the case for all the meshes, hence calling `BufferGeometry.dispose()`, `Material.dispose()` and `Texture.dispose()` (`BufferGeometry` is the base class of `BoxGeometry` and `CylinderGeometry`).
Finally, we call `Timeline.kill()` on a mesh's Timeline to force the completion of any animation and release it for garbage collection.

The switch between scaling modes or tiling methods and the adjustment of the child count (Section 3.3) requires a new general layout and, consequently, the disposal of the currently shown memory city via execution of `resetScene()` before generating a new city. **Fig. 22** shows the effect of calling `resetScene()` during runtime.

[15] https://threejs.org/docs/?q=mesh#api/en/math/Quaternion



```
> renderer.info
<· ▼{memory: {…}, render: {…}, programs: Array(3), autoReset: true, reset: ƒ, …} i
     autoReset: true
   ▼memory:
       geometries: 537
       textures: 53
     ▶[[Prototype]]: Object
   ▼programs: Array(3)
     ▶0: WebGLProgram {name: 'LineBasicMaterial', id: 0, getUniforms: ƒ, getAttributes: …
     ▶1: WebGLProgram {name: 'MeshPhongMaterial', id: 1, getUniforms: ƒ, getAttributes: …
     ▶2: WebGLProgram {name: 'MeshPhongMaterial', id: 4, getUniforms: ƒ, getAttributes: …
       length: 3
     ▶[[Prototype]]: Array(0)
   ▶render: {frame: 2941, calls: 531, triangles: 6260, points: 0, lines: 2868}
   ▶reset: ƒ reset()
   ▶update: ƒ update(count, mode, instanceCount)
   ▶[[Prototype]]: Object
> resetScene();
<· undefined
> renderer.info
<· ▼{memory: {…}, render: {…}, programs: Array(0), autoReset: true, reset: ƒ, …} i
     autoReset: true
   ▶memory: {geometries: 0, textures: 0}
   ▶programs: []
   ▶render: {frame: 4252, calls: 0, triangles: 0, points: 0, lines: 0}
   ▶reset: ƒ reset()
   ▶update: ƒ update(count, mode, instanceCount)
   ▶[[Prototype]]: Object
>
```

Fig. 22. This screenshot is taken out of Google Chrome's console. A renderer's `info` object gives valuable insights into the rendering process and is helpful for debugging. It discloses the number of geometries, textures and more, that are inside a Three.js app's heap memory. After calling `resetScene()`, we successfully delete deprecated city remains.

6 Future Work

This section covers thoughts and ideas on how Memory Cities can be extended to portray a more comprehensive tool.

6.1 Animations

Currently, animations are mostly restricted to buildings and their scaling in size during Time Travel, which gives users better awareness of strongly growing heap object groups. Similarly, animating the sizing of frustums during Time Travel can be beneficial. An innovative feature may be *automatic memory leak detection*. Looking at the example from Section **Heap Object References Fig. 12**, the frustums between buildings could be rendered (and animated) automatically one by one, showing the next largest heap object reference (frustum) until a building is reached that has only outgoing frustums. This way, without user interaction, we can trace the memory leak from its symptoms back to the actual root cause.

Another use case for animations might be during the layout process, specifically when generating districts. For example, switching between scaling modes renders the underlying treemap of the city immediately. Contrastingly, animating the scaling of districts provides a smooth transition between alternative views of the same memory city.

6.2 Metrics & Visual Mapping

Implementing more visual attributes based on provided data and metrics can enrich memory cities' effectiveness in general. At present, a building's height is derived from its base area. Consequently, both visual attributes depict the same metric, either object count or byte count. Establishing a new metric on which a building's height is based as well as splitting both metrics to represent unique optical traits is still for future research to decide. Regarding the second point, a possible solution may be the introduction of several textures and materials applied to the surface of a building.

Printing information onto structures (such as in **Fig. 23**) can also yield instant value to the users. But, if text gets mapped onto buildings, one should pay attention not to build an excessive and unmanageable visualization but rather keep it clear and practical simultaneously.

The novice-friendly nature of Memory Cities is a unique selling point regarding the (usually complicated) task of memory leak analysis. All of the more complex visual mappings should be part of an *expert mode*, as stated in the original paper Section X.B [10].

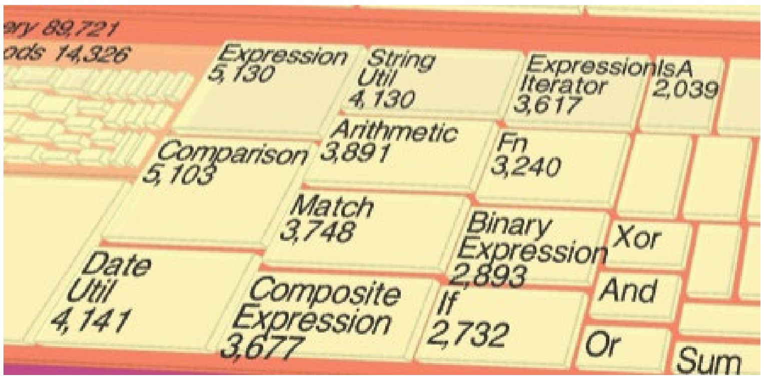


Fig. 23. Text on a 3D treemap (https://observablehq.com/@analyzer2004/3d-treemap).

6.3 Generalization

Memory cities are inspired by the software city metaphor, which also inspired other researchers, resulting in a collection of implementations for diverse application contexts. However, the basic concept is the same: presenting tree-structured data as a 3D city visualization. In the future, a more general and universally usable tool could be implemented, where users can provide parameters which map the input data to various visual attributes of a city. Hence, this new tool cancels any restrictions on the application context and acts as a one-for-all solution when aiming to present any tree-structured data as a city.

7 Conclusion

As part of this thesis, we ported the original Memory Cities [10] to JavaScript. First, we explained the concept of memory trees and how multiple heap dumps are collected and parsed, specifically how their heap objects are grouped based on shared properties such as type and allocation site. Not only does the resulting set of tree-structured input data get visualized as districts and buildings, but also the memory evolution over time can be animated. Using the Time Travel feature, memory leaks are exposed as strongly growing buildings and emphasized with color and opacity settings. Apart from that, displaying heap object references is a powerful way to detect the root cause of incorrectly managed memory allocations.
Moreover, we address the differences between the desktop and the online application (e.g., how input data gets processed) and present a review on Three.js and how it is put to use in Memory Cities.
Online Memory Cities aims to increase the easy accessibility for novice users even more but also assist experienced developers. To benefit the latter to a greater extent, we hope to see future updates for the Memory Cities web app and contributions regarding the ideas presented in the previous Section 6.

References

[1] Bruls, M., Huizing, K., and van Wijk, J. J. uuuu-uuuu. Squarified Treemaps. In *Data Visualization 2000*, W. C. de Leeuw and R. van Liere, Eds. Eurographics. Springer-Verlag, Vienna, 33–42. DOI=10.1007/978-3-7091-6783-0_4.

[2] Johnson, B. and Shneiderman, B. 1991. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Visualization '91. 2nd Annual conference : Papers*. IEEE Comput. Soc. Press, 284–291. DOI=10.1109/VISUAL.1991.175815.

[3] Kong, N., Heer, J., and Agrawala, M. 2010. Perceptual guidelines for creating rectangular treemaps. *IEEE transactions on visualization and computer graphics* 16, 6, 990–998.

[4] Langelier, G., Sahraoui, H., and Poulin, P. 2008. Exploring the evolution of software quality with animated visualization. In *Proceedings*. IEEE Computer Society, Los Alamitos, CA, 13–20. DOI=10.1109/VLHCC.2008.4639052.

[5] Lv, L., Fan, S., Huang, M., Huang, W., and Yang, G. 2017. Golden Rectangle Treemap. *J. Phys.: Conf. Ser.* 787, 12007.

[6] Scheibel, W., Trapp, M., Limberger, D., and Döllner, J. 2020. A Taxonomy of Treemap Visualization Techniques. In *Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*. SCITEPRESS - Science and Technology Publications, 273–280. DOI=10.5220/0009153902730280.

[7] Shneiderman, B. 1992. Tree visualization with tree-maps. *ACM Trans. Graph.* 11, 1, 92–99.

[8] Sondag, M. F. M. 2016. *Stability of treemap algorithms*.

[9] Weninger, M., Gander, E., and Mössenböck, H. 2018. Utilizing object reference graphs and garbage collection roots to detect memory leaks in offline memory monitoring. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes - ManLang '18*. ACM Press, New York, New York, USA, 1–13. DOI=10.1145/3237009.3237023.

[10] Weninger, M., Makor, L., and Mossenbock, H. 2020. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 110–121. DOI=10.1109/VISSOFT51673.2020.00017.

[11] Weninger, M. and Mössenböck, H. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 2018 ACMSPEC International Conference on Performance Engineering*. ACM Conferences. ACM, New York, NY, 115–126. DOI=10.1145/3184407.3184412.

[12] Wettel, R. and Lanza, M. 2007. Visualizing Software Systems as Cities. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007*. IEEE / Institute of Electrical and Electronics Engineers Incorporated, 92–99. DOI=10.1109/VISSOF.2007.4290706.

[13] Wettel, R. and Lanza, M. 2008. CodeCity. In *ICSE'08 companion. Companion material of the thirtieth international conference on software engineering*. ACM Press, New York, New York, USA, 921. DOI=10.1145/1370175.1370188.

[14] Wettel, R. and Lanza, M. 2008. Visual Exploration of Large-Scale System Evolution. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*. IEEE / Institute of Electrical and Electronics Engineers Incorporated, 219–228. DOI=10.1109/WCRE.2008.55.

List of Figures