JOHANNES KEPLER
UNIVERSITY LINZ

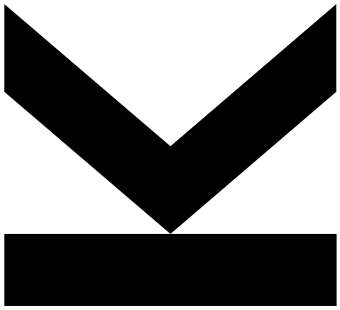
Author
Kaan Baylan

Submission
Institute for System Software

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

May 2023

Synchronized Timeline View for Software City Visualization



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

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis

Synchronized Timeline View for Memory Cities

Student: Kaan Baylan
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Start date: March 2022

Dipl.-Ing. Dr.
Markus Weninger, BSc
Institute for System Software

P +43-732-2468-4361
F +43-732-2468-4345
markus.weninger@jku.at

The software city visualization metaphor is used in a wide range of visualization applications such visualizing the development history of a software system, its memory consumption, or the communication between its various components. The core idea of this metaphor is to depict given software entities (for example classes) as individually sized buildings (for example based on line count and method count) and arranging these buildings hierarchically in districts (for example based on their packages).

While some approaches only visualize the state of the software system at one point in time, a number of city visualizations enable users to step through time (*time-traveling*) to inspect the system's development over time. Yet, changes from one point in time to the next are often minimal, making it hard to detect general (growth) trends in the city without performing a larger number of steps and carefully inspecting the resulting visual changes.

In the past, we have developed *Memory Cities*, a technique to *visualize an application's heap memory evolution over time* using the software city metaphor. In this tool, 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. Time-travelling through the different states of the heap, i.e., watching the city evolve, can be used to detect and analyze memory leaks by searching for suspicious growth behavior.

The goal of this work is to introduce a *synchronized timeline view* for memory cities. Such a timeline outlines the evolution of the system over time by depicting the city in multiple thumbnails that are placed in juxtaposition, i.e., side by side. These thumbnails show the city at the current point in time, as well as a number of times in the future and the past. They are synchronized with the main view, i.e., if the city is rotated or zoomed, all thumbnails are rotated and zoomed at the same time as well.
This approach should provide easier insights into trends over time, as the user can inspect the city at the current point in time from different angles and at the same time can see their respective past and future forms without the need to step through time.

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. 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 31.09.2022.

Abstract

Many different visualization applications employ the software city visualization metaphor. This concept's central notion is to represent certain software entities (such as classes) as individually sized buildings that are arranged hierarchically into districts, for example by looking at their packages. The sizes of these buildings are based on factors such as line count or the method count.

Several city visualizations let users move across time (time-travel) to examine the systems progress through time. However, changes from one time period to the next are oftentimes very small, thereby making it difficult to identify general trends (growth) in the city without taking multiple steps and closely examining the resulting visual changes.

To overcome these problems a synchronized timeline view has been implemented. A timeline such as this illustrates how the system has changed over time by displaying various thumbnails of the city side by side in juxtaposition. These thumbnails portray the city in the present, as well as various points in the past and future. They move in sync with the main view; for example, if the city is rotated or zoomed, then all thumbnails do the same simultaneously.

This solution should simplify understanding trends across time since it allows the user to examine the city from many perspectives whilst also simultaneously viewing its previous and future iterations without having to travel through time.

Kurzfassung

Viele verschiedene Visualisierungsprogramme verwenden das Konzept der Software Cities. Die Idee dieses Konzeptes ist es, eine bestimmte Software-Entität (beispielsweise Klassen) als Gebäude darzustellen. Diese Gebäude, welche hierarchisch in Distrikte eingeordnet werden, haben eine individuelle Größe. Das Einordnen in die Distrikte geschieht beispielsweise auf Basis des Pakets. Die Größe der einzelnen Gebäude berechnet sich aus Faktoren wie beispielsweise der Anzahl der Zeilen oder der Anzahl der Methoden.

Mehrere Stadt-Visualisierungen lassen den Benutzer durch die Zeit springen, um sich den Fortschritt des Systems, der mit der Zeit passiert, anzusehen. Jedoch sind Änderungen von einem Zeitpunkt zum nächsten öfters sehr klein, weswegen es schwierig ist, generelle Trends (Wachstum) in der Stadt zu erkennen, ohne mehrere Schritte durchführen zu müssen und die visuellen Änderungen genauer zu betrachten.

Um diese Probleme zu lösen, wurde eine synchronisierte Timeline-Ansicht programmiert. Diese Timeline illustriert wie sich das System über die Zeit verändert, indem es verschiedene Thumbnails der Stadt nebeneinander darstellt. Die Thumbnails schildern die Stadt sowohl in der Gegenwart als auch in der Vergangenheit und der Zukunft. Diese Thumbnails bewegen sich synchron zu der Hauptansicht: Wenn die Stadt beispielsweise rotiert oder vergrößert wird, machen dies auch die Thumbnails gleichzeitig.

Diese Lösung soll das Verstehens von Trends über Zeit vereinfachen, da es dem Benutzer erlaubt die Stadt von mehreren Perspektiven zu untersuchen, während gleichzeitig auch die vorherigen und zukünftigen Iterationen angezeigt werden, ohne durch die Zeit reisen zu müssen.

Table of Content

Contents

Abstract i

Kurzfassung i

1 Introduction 1

2 Background 3
2.1 Timeline . . . 3
2.2 Memory Cities . . . 3
2.2.1 Data . . . 3
2.2.2 Metrics and Visual Mapping . . . 4
2.3 Unity . . . 5
2.3.1 Basic Unity Game / Application Structure . . . 5
2.3.2 Scenes . . . 6
2.3.3 Camera . . . 6
2.3.4 Layering . . . 6
2.3.5 Scripting . . . 7
2.3.6 Canvas . . . 7

3 Approach 9
3.1 Overview of the Memory City Application . . . 9
3.2 Architecture . . . 10
3.3 Time Traveling . . . 12
3.4 Methods of generating multiple cities . . . 12
3.4.1 "Offset"-Method . . . 13
3.4.2 "Different Scenes"-Method . . . 14
3.4.3 "Same Scene Using Layers"-Method . . . 15

4 Implementation 17
4.1 Timeline Canvas . . . 17
4.1.1 Timeline Slot . . . 17
4.1.2 Timeline Jump Value . . . 18
4.1.3 Synchronized Cameras . . . 18
4.2 Generating and Layering of the Cities . . . 19
4.2.1 Loading Data for the Cities . . . 19
4.2.2 Creating the Cities . . . 21
4.2.3 Adjusting the cameras to the layers . . . 23
4.2.4 Pointers . . . 23
4.2.5 Growth . . . 26
4.3 Time Travel . . . 26
4.3.1 CurDisplayTreeIndex . . . 26
4.3.2 Reordering the Timeline . . . 28

5 Usage **31**
5.1 General Timeline 31
5.2 Changing the Timeline Jump Value 32
5.3 Timeline with Pointers 32
5.4 Timeline with Growth 33
5.5 Time travel through the Timeline 34

6 Conclusions and Future Work **35**

Literature **37**

1 Introduction

Garbage Collectors (GC) are utilized in modern programming languages to automatically release memory from the heap that is no longer accessible through static fields or local variables. Although the GC is doing a decent job of releasing this unused memory, memory issues and anomalies are still a possibility due to programming mistake. This frequently happens because a developer forgets to delete objects from a data structure that contains long-living objects. Those items can't be recovered by the garbage collector, thus they will just build up over time. [7] In order to locate these problems and anomalies, users must look for groups of objects that increase suspiciously over time in the heap memory.

Markus Weninger et al. [7] developed *Memory Cities* to simplify this task by using the Software City metaphor to display the evolution of the heap memory. In this tool, heap object groups are visualized as buildings and are organised into districts based on shared heap object properties, such as their type. The size of the buildings can alter over time, representing the increase and decrease of the allocated heap memory from an object group throughout the lifetime of a tracked program.

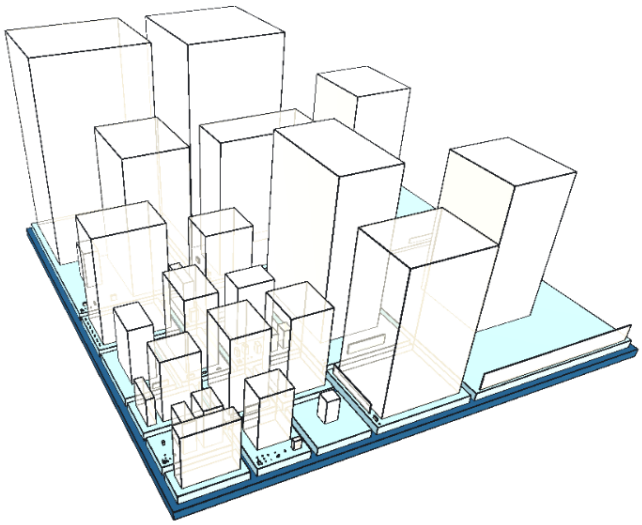


Figure 1: An example of an visualization of the heap memory using the software city metaphor produced by the *Memory Cities* Application

The primary contribution of this thesis is the implementation of a *synchronized timeline view* for the already-existing *Memory Cities* Application. This timeline should be composed of synchronized thumbnails with the city shown at different times. Synchronized in this context means that the thumbnails should mimic the main *city's* movement, ensuring that when the city rotates, zooms, or moves, the thumbnails should simultaneously do the same.

2 Background

The required knowledge about the *Memory City* metaphor, such as its metrics and visual cues, is provided in this section alongside with a description of a timeline. Additionally, because the *Memory City* application was programmed using the Unity framework, fundamental knowledge of the Unity environment is covered and a few Unity terminologies are defined.

2.1 Timeline

A **timeline** is an essential tool to illustrate the chronology of events. This could be accomplished by drawing a line and arranging events on the line chronologically, putting the earliest event on the left and the latest event on the right. To have an easier understanding you can see a timeline in Figure 2 where the past four years are chronologically ordered.

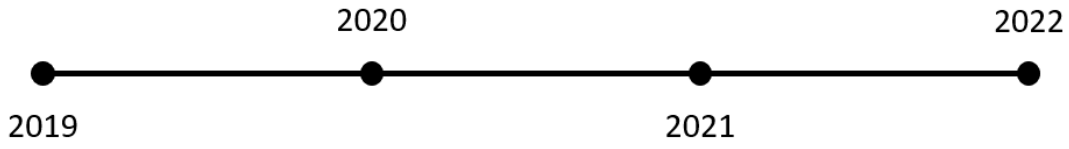


Figure 2: A timeline displaying the past four years.

This application's timeline is comprised of a number of thumbnails that are presented side by side and portray the city in the present, the past, and the future.

2.2 Memory Cities

The term *Memory City* alludes to a metaphor describing software cities used to illustrate the development of heap memory. The designed cities ought to offer enough information to enable users to pinpoint memory leaks while still maintaining a straightforward visualization. [7, 6]

2.2.1 Data

In the *Memory City* application [7], data of heap memory states are handled as trees, where each node characterises a group of heap objects. The subdivision of the heap memory is done with the help of the program *AntTracks*. The heap objects in each node can be counted in either of the following two ways:

- Counting the number of objects in the group
- Counting the number of bytes, that the respective objects take up in the heap

Each tree node in the *Memory City* application stores both of these metrics. Additionally, every tree node stores the following information:

- A unique **key** to identify of the object group
- A human-readable **name**

- A **specification role** which determines the grouping criteria (e.g., “Allocation Site”)
- And a **list of child nodes**. (this list is empty if the node is a leaf node of the tree)

The *Memory City* application loads a list of trees in order to track changes in the heap memory since we want to be able to see how the heap memory has shifted over time. These trees, which also carry a timestamp to guarantee proper ordering, reflect the different states of the heap after garbage collections. After loading a list of trees, the application calculates the following meta trees:

1. **The max tree:**
   The maximum amount of objects and bytes that a tree node holds at any time are stored in this tree.
2. **The growth tree:**
   The growth of each node throughout the evolution of the heap is stored in this tree. The difference between the first and last points in time is the growth value for the respective node.

2.2.2 Metrics and Visual Mapping

A building’s area in a city is determined by the amount of objects or bytes that its tree node holds. To provide users with more information, the *Memory Cities* incorporate additional visual cues.

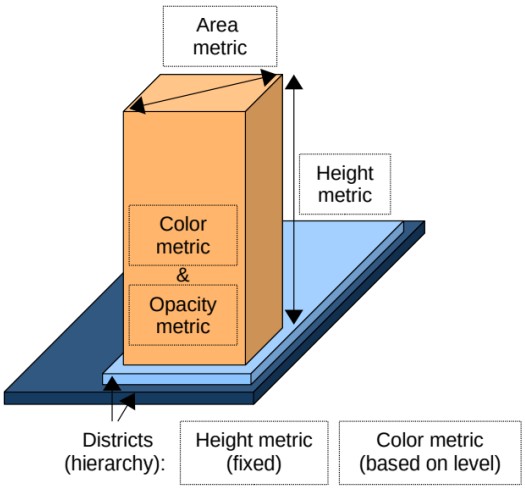


Figure 3: The visual attributes of a building in the *Memory City* application. (taken from [7])

- **Districts**:
  In cities, districts are flat structures, meaning their height is fixed and uninform. These districts serve to illustrate the underlying tree's hierarchy. A district can again be broken into more districts, one for each inner node in the underlying tree. The root district, or bottom most, always represents the complete heap.

- **Buildings**:
  The visual characteristics height, color, and opacity for each building are used additionally to the area metric, which is calculated based on the amount of objects or bytes a building represents:

  - **Height**:
    Similar to the area of a building, the height is derived from the amount of objects or bytes that the building represents.
  - **Color**:
    A building's color is a representation of that building's growth. This is carried out to provide a stronger visual indicator of any unusual growth. If the building is gray, there has been no growth or it is receding. If it is orange, there has been moderate expansion. And if it is red, there has been an significant growth.
  - **Opacity**:
    The non-interesting buildings, i.e. the ones where no growth is taking place, are becoming transparent to provide the viewer a clear indication of where to look, whilst the buildings with a moderate or substantial growth are opaque.

2.3 Unity

The cross-platform gaming engine *Unity* was used in the programming of the *Memory City* application. In the following sections that follows, we talk about *Unity* in general and define a few key terms.

2.3.1 Basic Unity Game / Application Structure

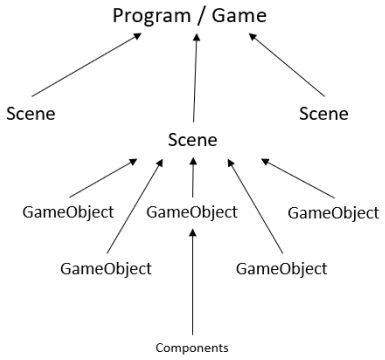


Figure 4: The basic structure of an *Unity* Program / Game. inspired by [5]

Scenes are the foundation of every *Unity* application or game. There is always at least one camera object in a scene. All of the information in their viewport is rendered or "captured" by these camera objects. The user can see everything that is visible in the cameras viewpoint.

GameObjects are the actual objects that make up scenes. Anything that is used in the scene can be considered as a *GameObject*. A button, an image, a sound, etc. are examples of *GameObjects*. These *GameObjects* include a number of components that specify how they interact with other *GameObjects* and how they behave in the scene. [5]

2.3.2 Scenes

Scenes are where one works with *GameObjects*. They are assets that may fully or partially contain a game or an application. A more complicated game might utilize a scene for each loading screen and level, each with its own *GameObjects* such as backgrounds, characters, obstacles, decorations and UI, whereas a simpler game or application might only require one scene. [3]

2.3.3 Camera

GameObjects are shown in three dimensions within scenes. *Unity* must take a view and "flatten" it to be able to display the content, since the viewers screen is only two-dimensional. Therefore, cameras are utilised to show the user a scene's "world". In every scene there can be more than one camera, but at least one.
Cameras are just such as other *GameObjects*: They can be moved, rotated and so on. A cameras output can also be used as a texture, which is then called a *Render Texture*. A texture in *Unity* applied to a *GameObject's* to change its surface. [1]

2.3.4 Layering

Layers may be utilised in the UI and may be used in scripts to change how *GameObjects* interact with one another. They also can be used to optimize the workflow of a project. The common uses for layers are the following:

- **layer-based rendering**:
  Each object is rendered only in the layer it is in.
- **layer-based collision**:
  Objects only have collision detection when the layer they are on, has the collision detection activated for the other objects layer.
- **Camera using the culling-mask with layers**:
  Rendering just the items in a selected layer or levels..
- **Ray cast with layers**:
  Which GameObjects a ray cast can intersect with can be specified using layers. [4]

For this thesis, the most important use case is the culling-mask of the camera: *GameObjects* within a certain layer are then only visible for a camera which has the culling mask activated for that layer. Thus, for those cameras that do not have that layer activated in the culling mask, the respective objects are completely invisible.

2.3.5 Scripting

In *Unity*, writing code blocks that are attached to *GameObjects* is essential for the development of an application. In *Unity*, scripting is done using either C# or UnityScript, which is *Unity's* implementation of *JavaScript*. In the case of the *Memory Citiy* Application, C# was used. [3]

```
public class NewScript: MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Listing 1: An newly created C# script for Unity.

When a new C# script for *Unity* is created, it appears as shown in Listing 1. There are two methods by default:

- **Start-Method**:
  This method is only called once on the first frame that the *GameObject* to which this script is attached to is active in the scene.
- **Update-Method**:
  This method is called on every frame after the start-method has been called. Normally, a Unity application runs at 60 frames per second. This would mean that the update-method is called 60 times in a second.

2.3.6 Canvas

The Canvas is the area where all UI elements ((e.g. buttons or text boxes) ought to be placed. Because to be displayed in *Unity*, every UI element needs to be a child of such a canvas. Multiple UI elements can be placed on a single canvas. A canvas in *Unity* is merely a rectangular object that the programmer can resize and rotate as he wishes to. [2]

3 Approach

This section covers the synchronized timeline's creation. This includes some approaches on how multiple cities for different points in time can be generated and how footage from cameras can be gathered and shown in the user interface as thumbnails.

3.1 Overview of the Memory City Application

In the following section the general overview of the *Memory City* application is discussed.

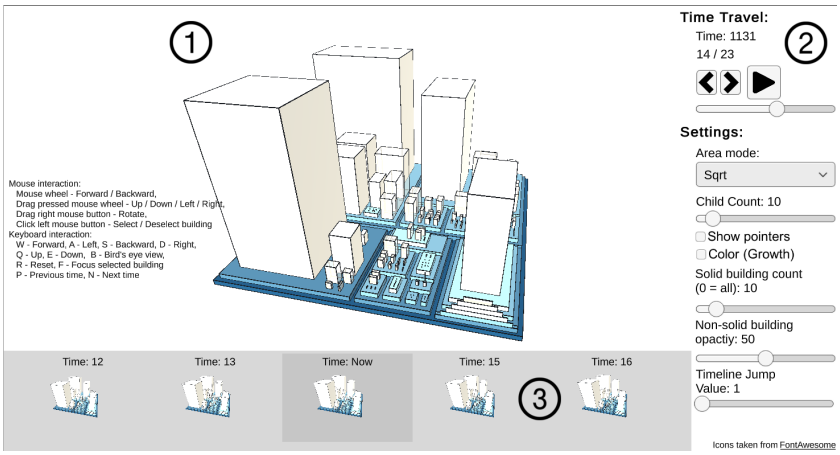


Figure 5: This screenshot of the *Memory City* application shows the application at use and has divided the UI into three sections

The UI of the *Memory City* application can be divided into three sections as we can see in Figure 5:

1. **Main city**:
   This is the city which the user can move, rotate and zoom as he wishes.

2. **Settings and Time Stepping**:
   These are the options that the user can change as he pleases, along with a time travel option which lets the user decide in which point of time he desires to see the city.

3. **NEW: Timeline**:
   This is new timeline which has been contributed through this thesis. The user can view many cities at various times here, as well as jump to the desired time by clicking on one of the thumbnails.

3.2 Architecture

In this section the general architecture is explained and the steps how the new timeline is integrated into the application is discussed. The architecture can be seen in Figure 6

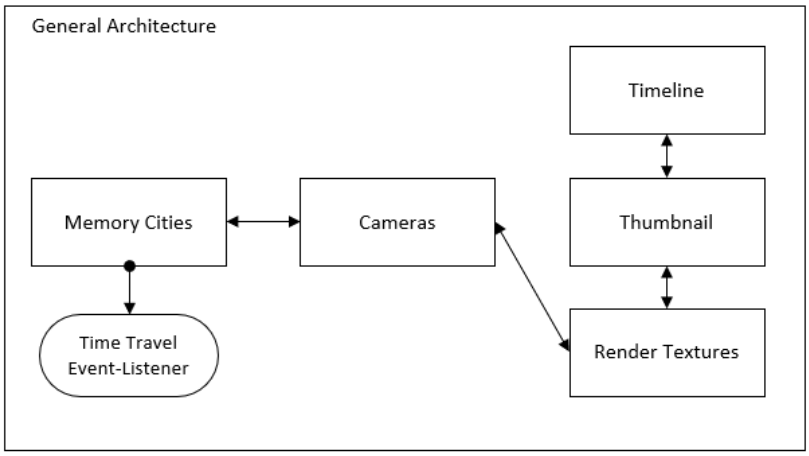


Figure 6: The general architecture and steps involved in integrating a timeline into the *Memory City* application.

- **Memory Cities**:
  Since the application should display multiple cities at different points in time in the timeline, there are multiple cities needed. Thus, for every thumbnail in the timeline a city is generated.
  The generation of the multiple cities can be achieved through multiple ways:

  1. generating the cities in different scenes,
  2. generating the cities far apart from each other in the same scene or
  3. generating the cities inside each other and employing layers in the cameras to limit each cameras point of view to only one city

  The creation of the cities is discussed in more detail in the Section 3.4.

  Inside of the various cities there is a event listener which gets called when a time travel happens:

  - **Time-Travel Event Listener**:
    It should be possible to travel through time in the main view. When time travel occurs, the various cities in the thumbnails should update themselves accordingly. This is accomplished with an event listener, which updates the cities simultaneously as the main city gets changed. The approach on how time traveling works is discussed in Section 3.3

This event listener is also responsible for moving the current *city's* thumbnail to the right spot in the timeline: At the left-most when it is the first state, in the middle if there are past and future states of the city or at the right-most position of the timeline if the state we are currently in is the final one. This can be seen in Figure 7.

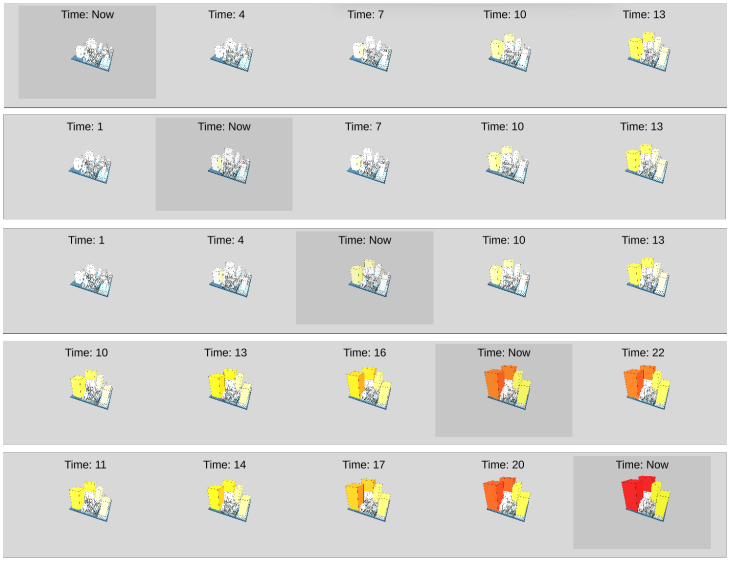


Figure 7: The positions in which the "current" city can be.

- **Cameras**:
  Cameras are needed in order to record the various cities in the city application. Each camera is assigned to a certain city and thus there are as many cameras as there are cities. With the exception of the main camera, every other camera has a script attached to it which replicates the rotation and movement of the main camera. This guarantees that all cameras always view the different cities from the same angle.

- **Render Texture**:
  The application also has a *Render Texture* for each camera pointing at a city. The views from the various cameras are displayed in the timeline by using the generated *Render Textures* of the cameras.

- **Thumbnail**:
  The thumbnails of the cities include the text describing in which point of time the city is in this thumbnail and the *Render Texture* of the camera pointed at this city. These thumbnails serve also as buttons to jump to the given timestamp. This feature is discussed in more detail in Section 3.3.

- **Timeline**:
  The complete thumbnails are displayed in the timeline adjoining to each other and in the proper chronological order. The *Memory City* application displays the concluded timeline view at the bottom of the screen.

3.3 Time Traveling

A large emphasis should be placed on using the created timeline for the *Memory City* application to travel through time. The user should be able to view the city at various points in time using the timeline, and when he wishes to travel to a certain timestamp shown in the timeline. This can be done by simply clicking on the respective thumbnail. After clicking the thumbnail, the city represented in the thumbnail, should become the main city and the other cities in the thumbnails should update accordingly.

Additionally, there is a slider to adjust the *timeline jump* value between the cities in the thumbnails. This is useful when the user wants to see bigger steps between the displayed cities. The range of the slider is a whole numbers between one and a calculated value, which assures that there are always at least five thumbnails possible to display. Thus, the timeline is always able to display the same amount of thumbnails. At the start of the application the sliders value is set to one.

Figure 8: Two example timelines one with a *timeline jump* value of one (top) and one with a *timeline jump* value of four (bottom).

3.4 Methods of generating multiple cities

Underlying each thumbnail in the application, there is a different city, as we already outlined in Section 3.2. In the following Methods the main city is always generated by using the same steps: It gets generated in the programs main scene, where the focus of the main camera is set to this city. There are several approaches to generate the additional cameras that are going to be featured in the timeline view, which are discussed in the following sections.

3.4.1 "Offset"-Method

The first approach is the "Offset"-Method, which involves creating cities in the same scene but with a spatial offset to place the cities far away from one another.

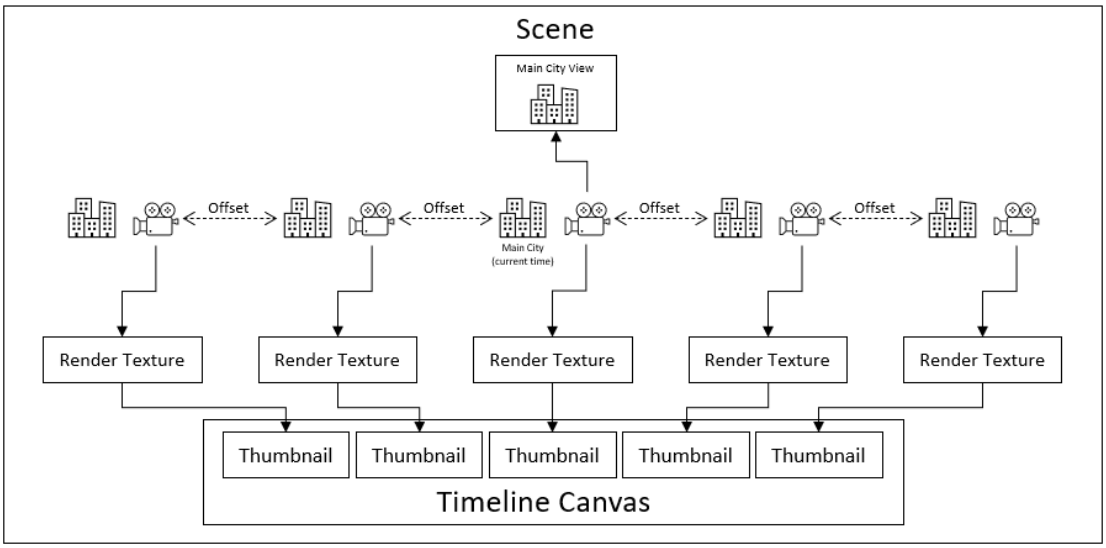


Figure 9: The generation of cities while only using one scene and placing the cities far away from each other.

This approach only requires one scene, with the cities dispersed throughout it. We have three strategies to spread them out:

1. using the X or the Y coordinate or a combination of both to spread them on the same plain,
2. using the Z coordinate to spread them vertically up or down, or
3. Using a combination of X, Y and Z to spread them out.

No matter how the offset of the coordinates of the cities is done, their result is the same: Multiple cities spread across a scene.

The drawback of this approach is that the user may be able to see two cities with either the main camera or one of the other cameras displayed in the timeline if our offset is too little. To remedy this, we could adjust the render distance (in *Unity*: the far clipping plane) so that the user cannot see that far, or we could also tweak the offset value to allow a city to spawn further away.

3.4.2 "Different Scenes"-Method

The second technique discussed uses multiple scenes to display our various cities.

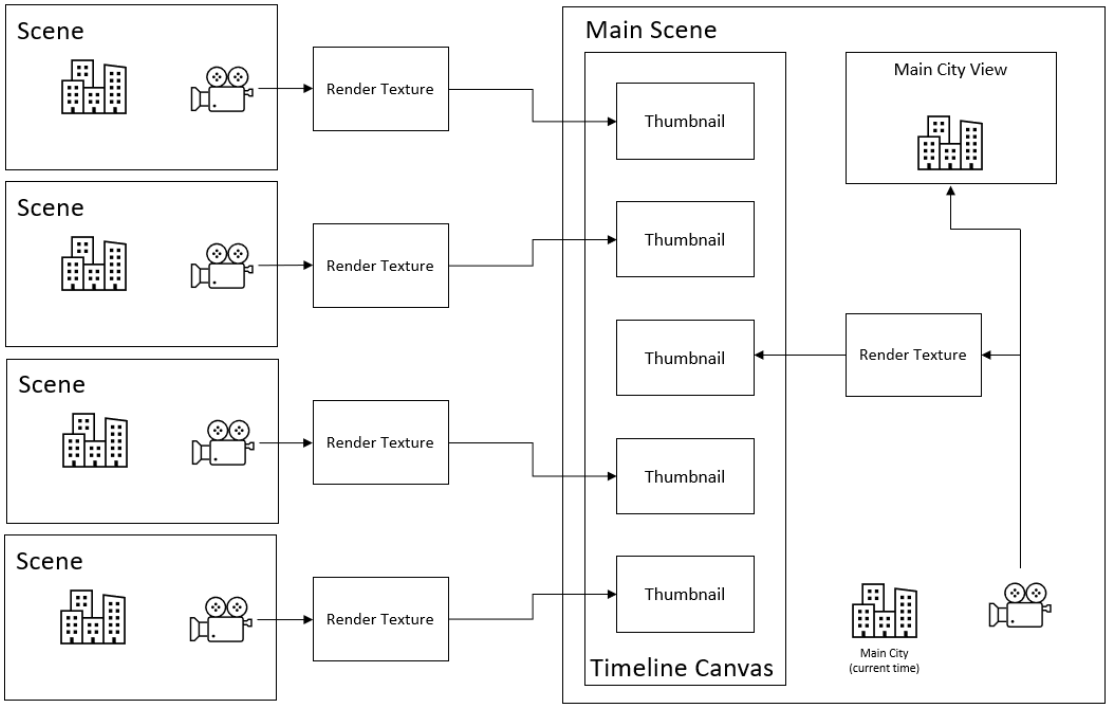


Figure 10: The generation of cities in multiple scenes and displaying them in the timeline view.

Using this technique multiple scenes are produced. More precisely, it requires one scene per city in our timeline. Each scene has one of the various cities and a camera that records the city inside of it. Each camera has a corresponding *Render Texture* that is utilized by the associated thumbnail to access and present the content of it.

On the one hand, in this method there is no possibility that the user ever sees two cities on the same camera. On the other hand, because numerous scenes would be running simultaneously, the performance would be impacted negatively.

3.4.3 "Same Scene Using Layers"-Method

In our last method all cities are constructed at the same location, yet they ware placed in different layers. Due to the addition of these layers the culling mask of the cameras can be employed. As introduced in Section 2.3.3, a camera can be narrowed down to be able to see only objects from a specific layer. By utilizing this, every camera is able to only see the city it shares the layer with.

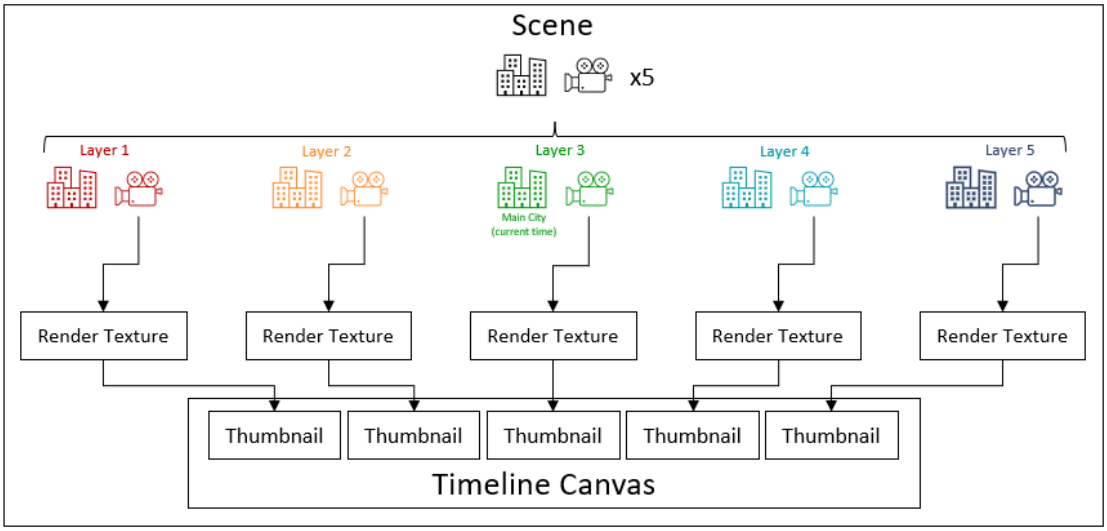


Figure 11: The generation of cities at the same spot utilizing layers.

This approach only requires one scene and the user is never able to see two cities at once. Since every city is only visible for the camera that has its culling mask set to the city's layer.

This is in our opinion the best solution for our use case because it merely demands the use of a single scene and the user can never see two cities at once.

4 Implementation

We discussed the timeline's structure and how we establish the various functionalities in Section 3. We now get deeper into the practical side of programming and constructing the timeline view in this section. We start by looking at how the timeline canvas is created and how it is structured in Unity. Further, we also examine how the synchronization of cameras is done and how the generation and display of the cities is accomplished. Furthermore, we look into the growth functionality and the layering of these as well.
Last but not least, we have a look at the time traveling functionality.

4.1 Timeline Canvas

The primary contribution of this thesis is the timeline canvas. It is located at the bottom of the screen and contains the various thumbnails. This element's width scales with the size of the application window and has a fixed height.

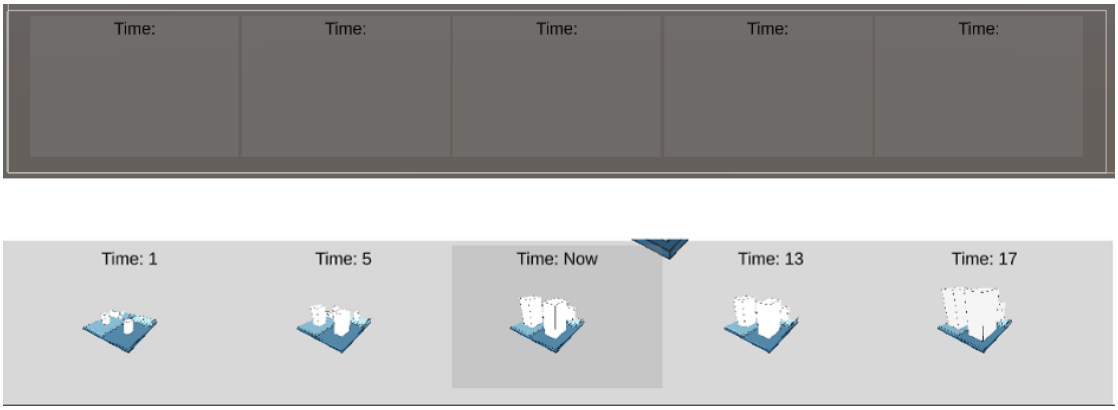


Figure 12: On the top is the raw timeline canvas where we can see the structure and how the timeline will be placed together and on the bottom is an example of a "real" timeline.

The five timeline slots plus the background make up the timeline canvas as we can see in Figure 12. To distinguish the timeline from the rest of the software, it is tinted light gray.

4.1.1 Timeline Slot

Each timeline slot consists of a background, the city itself and a text that specifies the timestamp at which the thumbnail is located. The background of the slots are used to indicate if this thumbnail's city is the main one. If the time slot contains a past or future iteration of the city the background is deactivated and thus the background is not displayed. This can be seen in the real timeline example in the Figure 12.

The thumbnail of the timeline slot is an image, which contains the *Render Texture* of its respective camera. By applying *Unity*'s button component on this image, we can infer from the user's click that they intend to travel to the chosen timestamp. As a result, this city becomes the main one and the timeline view itself updates accordingly. Section 4.3 explains this feature in more detail.

4.1.2 Timeline Jump Value

A slider that reflects the *timeline jump* value is introduced to allow users to glimpse further into the future or the past and travel there more quickly. The minimum value of this slider is set to one and the maximum value is determined by dividing the amount of points (the number of trees) by four and rounding the result. As a direct consequence, we can be certain that the timeline is always able to display five thumbnails. In the Figure 13 the *timeline jump* value slider can be seen.

Figure 13: The *timeline jump* value slider indicates how much the difference from one city to another should be.

4.1.3 Synchronized Cameras

In the timeline canvas all cameras should mimic the movement of the main camera. In order to implement this, every timeline camera attached with the *SyncCamera* script. This is a simple script which assures that all cameras look at the city the same way.

```
public class SyncCamera : MonoBehaviour
{
    Camera Cam;

    void Start()
    {
        Cam = GameObject.Find("Main Camera").GetComponent<Camera>();
    }

    void Update()
    {
        transform.position = Cam.transform.position;
        transform.rotation = Cam.transform.rotation;
    }
}
```

Listing 2: The *SyncCamera* script is used to apply the movement of the main camera to the camera to which this script is attached to.

As we can see in Listing 2, by using the *Start* method we search for the main camera when the first frame is loaded and save the found camera into our *Cam* variable. Afterwards, by using the `Update` method we set two properties every time a frame is rendered:

- **position**: Where is the camera located in the scene
- **rotation**: In which direction does the camera look

The cameras which have this script attached to them are therefore in the same exact place and look at the same direction as the main camera. Because this method is called every frame, which is rendered, the main cameras movement is replicated immediately and therefore no delay can be noticed.

4.2 Generating and Layering of the Cities

In order to be able to show the *cities* in the timeline canvas, we first need to generate them and give all of them a specific layer. In the following we go slightly through the creation of a city and how the layering is accomplished. Afterwards, we also take a look at how the pointers and the coloring of the buildings (based on the building's growth) is done with the addition of layering.

4.2.1 Loading Data for the Cities

Before the creation of the cities begins, we first need to load data. This is stored in a JSON file consisting of a list of trees. The program's heap states are represented by these trees, which should be displayed as the cities. The loaded information is kept in a class called `Data`.

The `Data` class declares two classes inside of it:

- **OverallData**: A class containing a list of trees, which represent the entire evolution of the heap memory.
- **CurrentData**: A class that represents a single tree at a single point in time that should be displayed as a city.

The *Data* class uses the two classes above and creates, besides other, following variables:

- `OverallData` **Overall**: The overall list of trees.
- `CurrentData` **Current**: The tree which is displayed as the main city.
- `CurrentData4` **Timeline**: The trees that are displayed in the timeline are stored in this array.
- `Int` **TimelinePos**: A whole integer between 0 and 4 that represents the location in the timeline, where the current primary city should be presented.

In this class there is also a property called `CurDisplayTreeIndex`, which, when set, updates `Current`, calculates the `TimelinePos` and also sets the trees of the `Timeline-Array`. This property is discussed further in Section 4.3.1.

```
public Data()
{
    Overall = new OverallData(this);
    Current = new CurrentData(this, 10);

    Timeline = new CurrentData[4];
    Timeline[0] = new CurrentData(this, 11);
    Timeline[1] = new CurrentData(this, 12);
    Timeline[2] = new CurrentData(this, 13);
    Timeline[3] = new CurrentData(this, 14);
}
```

Listing 3: The constructor of the *Data* class where the main *city's* (Current) data and the data of the timeline cities (Timeline) are getting initialized

In Listing 3 we can see the constructor of the `Data` class. Here the different fields are getting initialized with the corresponding layer they are associated with. The main city has the layer ten. Then the first timeline city gets the layer eleven and so on.

In *Unity* we can have up to 32 layers, five layers are always built in:

- Layer 0: Is the **Default** layer.
- Layer 1: is the **TransparentFX** layer.
- Layer 2: is the layer for **ignoring ray casting**.
- Layer 4: is the **Water** layer.
- Layer 5: is the **UI** layer.

We have selected the layers 10 to 14 for the simple reason that it is easy to remember.

4.2.2 Creating the Cities

Now that we have the data for the cities, we can start building them. The `DisplayCurrentTree` method is called (see Listing 4) once the cities data has been loaded.

```
Data.Current.CenterBuildingsInReservedSpace(Settings, VisualizationHelper); // 1

foreach (CurrentData data in Data.Timeline)
{
    data.CenterBuildingsInReservedSpace(Settings, VisualizationHelper);
    // 1
}

Data.Current.CenterBuildingsInReservedSpace(Settings, VisualizationHelper); // 2

foreach (CurrentData data in Data.Timeline)
{
    data.CenterBuildingsInReservedSpace(Settings, VisualizationHelper);
    // 2
}
```

Listing 4: A part of the `DisplayCurrentTree` mehtod that shows where the different cities first calculate thes positions and sizes of their buildings (1) and then call the `DisplayNodeRecursive` method to display the buildings and districts. (2)

This method, as we can see in Listing 4, calls the `DisplayNodeRecursive` method . The `DisplayNodeRecursive` method has following parameters:

- `Node` **cur**: The tree node that should be displayed.
- `Int` **level**: The level that this node is positioned on. This is crucial so that the districts can acquire their height and colour, as well as for the buildings, which must be built on top of districts rather than inside or beneath them.
- `GameObject` **parent**: The parent within which the node's district or building should be placed in.
- `CurrentData` **currentData**: The tree of the city that is currently being built.
- `Int` **layer**: The layer that should be used in the generated *GameObjects* (i.e., districts and buildings).

The `DisplayNodeRecursive` method runs through every node recursively and by using the `CreateMesh` method it creates a district or a building, depending on whether the current node is a leaf node. The setting of the layer is also done in the `CreateMesh` method for the cuboid. In Listing 5 the we can see how the layer is set in `CreateMesh` and how CreateMesh is called from `DisplayNodeRecursive`.

```
private void DisplayNodeRecursive(Node cur, int level, GameObject parent,
    CurrentData currentData, int layer)
{
    ...

    GameObject curGO = CreateMesh(curRect, height, color, level, parent,
    currentData, layer);
    currentData.KeyToGO[cur.FullKeyAsString] = curGO;
    curGO.name = cur.FullKeyAsString;

    if (cur.IsNonLeaf())
    {
        HashSet<string> reachableChildren =
            Data.Overall.MergedTopNChildrenPerNode[cur.FullKeyAsString];
        foreach (Node child in cur.Children)
        {
            if (reachableChildren.Contains(child.FullKeyAsString))
            {
                DisplayNodeRecursive(child, level + 1, curGO, currentData, layer);
            }
        }
    }
}

private GameObject CreateMesh(Rectangle rect, float height, Color color, int level
    , GameObject parent, CurrentData currentData, int layer)
{
    ...

    DrawOutlines(cube, rect.Width, height, rect.Height, layer);
    // Here the Outlines will be drawn, where we again need to set the layer to
    let the outlines be displayed inside of the thumbnails.

    cube.layer = layer; // With this the layer of the cube is getting set.

    return cube;
}
```

Listing 5: Calling the CreateMesh method and setting the layer of the created cube

After the `DisplayNodeRecursive` method is done a city has been created and is now ready to be displayed.

4.2.3 Adjusting the cameras to the layers

We need to modify our cameras so that they only display one city. The culling mask of the cameras is used for this. The final part in our applications initialization is shown in Listing 6.

The five cameras called `Timeline1` to `Timeline5` are getting their layers set. The culling mask of a camera is defined by 32-bits, every layer has its own bit. When the bit is false (0), then this layer is not rendered from the camera. If the bit is true (1), the layer is rendered from the camera.

The « operation is called a left shift. For example `1 « 10` shifts the 1 ten position to the left, creating the number 1024 being 10000000000 in binary. This then assures us that the current camera only shows the tenth layer.

```
Timeline1.cullingMask = 1 << 10;
Timeline2.cullingMask = 1 << 11;
Timeline3.cullingMask = 1 << 12;
Timeline4.cullingMask = 1 << 13;
Timeline5.cullingMask = 1 << 14;
```

Listing 6: Setting the culling mask of the cameras, ensures that only relevant buildings are recorded.

4.2.4 Pointers

Every building inside the city can have pointers pointing away from them (colored green) or pointers pointing at them (colored pink). These pointers are references between objects (the buildings) they are shown via a connection between two buildings as you can see in Figure 14

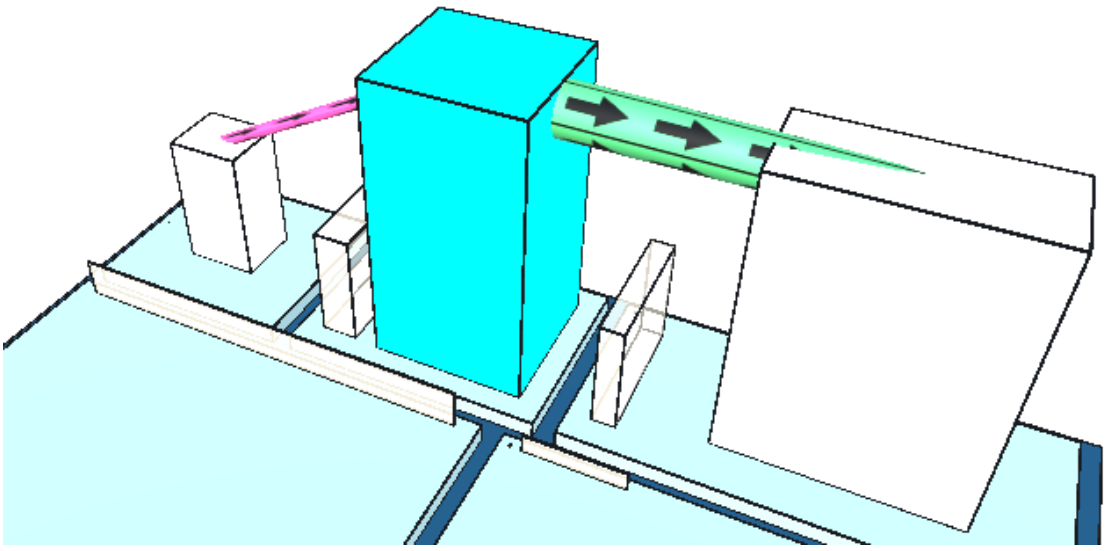


Figure 14: A building with two pointers.

As well as the cities buildings every city's pointers need to be created and set to the right layer to be able to display them inside of the *Render Texture* of the camera. For the creation of the pointers, the user must select a building from the main city. When the user selects a building, the `OnGOSelectedHandler` method is called with the key of the clicked building. This method is responsible for setting the selection to the selected building of every city. In Listing 7 we can see the two `SelectionHandlers` for selecting and deselecting a *GameObject*.

```
private void OnGOSelectedHandler(string fullKeyString)
{
    foreach (CurrentData data in Data.Timeline)
    {
        data.SelectedFullKeyString = fullKeyString;
    }
    Data.Current.SelectedFullKeyString = fullKeyString;
}

private void OnGODeselectedHandler()
{
    foreach (CurrentData data in Data.Timeline)
    {
        data.SelectedFullKeyString = null;
    }
    Data.Current.SelectedFullKeyString = null;
}
```

Listing 7: The selection handler when a building is clicked and the deselection handler when a building is deselected.

In Data, the `SelectedFullKeyString` property (see Listing 8) is a String, which helps us identify if and which building is currently selected. When this property's setter is called, it destroys all former pointers and at the end of the property the `SelectionChanged` method is called.

```
public string SelectedFullKeyString
{
    get { return _selectedFullKeyString; }
    set
    {
        var oldVal = _selectedFullKeyString;
        _selectedFullKeyString = value;
        // Delete pointer objects (connections between buildings)
        ...

        // Send event (updates color and creates new points objects, i.e.,
    building connections)
        SelectionChanged(oldVal, this);
    }
}
```

Listing 8: The `SelectedFullKeyString` property, which calls the `SelectionChanged` method for each city.

In the `SelectionChanged` method the first step is to set the color of the previously selected *GameObject* to its original color, if there is one. After that the selected building is highlighted by coloring it into a specific color. Furthermore, if the setting is enabled, every city gets its pointers generated with the use of the `DisplayPointers` method. This method creates *GameObjects* and adds them to the given layer, so that they show up in the *Render Texture* of the corresponding camera. This can be seen in Listing 9. At last, in this method the texts of the UI is updated, so that the selected buildings information is being displayed on the screen.

```
private void Current_SelectionChanged(string oldSelectedKey, CurrentData data) {
    // Reset color of old selection
    if (oldSelectedKey != null) {
        GameObject previouslySelectedGameObject = data.KeyToGO[oldSelectedKey];
        ChangedColor originalColorStore = previouslySelectedGameObject.
    GetComponent<ChangedColor>();

        if (originalColorStore != null) {
            previouslySelectedGameObject.GetComponent<MeshRenderer>().material.
    color =
                originalColorStore.OriginalColor;
            Destroy(originalColorStore);
        }
    }

    if (data.HasSelection) {
        // Update color of selected object
        MeshRenderer mr = data.SelectedGO.GetComponent<MeshRenderer>();
        data.SelectedGO.AddComponent<ChangedColor>().OriginalColor = mr.material.
    color;
        mr.material.color = VisualizationHelper.SelectedBuildingColor;

        if (Settings.ShowPointer) { DisplayPointers(data, data.Layer); }
    }
    UpdateTexts();
}
```

Listing 9: This listing presents the `SelectionChanged` method where the color of the selected building gets changed and the pointers are displayed when activated.

4.2.5 Growth

To see the color of the buildings in the thumbnails is quite helpful to spot suspiciously growing memory. If the setting for "Show Color (Growth)" in the settings area is set to true, then the color of the building is changed based on its growth. The coloring of the city is done inside of the `DisplayCurrentNode` method, which is called whenever a city needs to be repainted, i.e., whenever a `CurrentData`-object calls the `DisplayCurrentNode` method when it should get repainted. As we can see in Listing 10, the color is being returned by the `GetBuildingColor` method. This method takes the current node, the `OverallData` and the data of the city and returns the color that the building should have.

```
if (Settings.ShowColor)
{
    color = VisualizationHelper.GetBuildingColor(cur, Data.Overall, currentData);
}
```

Listing 10: This code block is a part of the `DisplayCurrentNode` method, where the color of the current node is returned by the `GetBuildingColor` method.

4.3 Time Travel

In this section the time traveling functionality is discussed. To initiate a time jump, the property `CurDisplayTreeIndex` in the `Data` class needs to be changed to the desired time.

4.3.1 CurDisplayTreeIndex

This property represents the index of the current tree that should be displayed as the main city. When this property is set, the first step inside its setter is to decide which time slot in the timeline this city should be placed in. Further, the `Timeline`-Array containing the `Data`-objects of the timeline gets updated to the new cities.

In Listing 11 we can see the selection of which thumbnail this city should be displayed in. First of all the `Timeline Jump Value` from the settings menu and the position of the current city are acquired. To decide where the current city will be placed there are three steps:

1. Check if the current city is able to have two successors and two predecessors without going out of bounds
   if this is the case than the city will be placed in the middle of the Timeline
2. Check if the current city can have two predecessors
   This is done by checking if the current position is less than two times the jump value. When this is true the city can only have one predecessor. This means that the city will be placed at the first thumbnail, if the position is zero or else at the second thumbnail.
3. When the current city is able to have two predecessors the next step is to check if the current city is the last element
   If it is the last one, the city will get placed onto the last position, if not there is only the penultimate position left.

```
int jumpValue = Settings.TimelineJump;  // Timeline Jump Value
int position = value;                   // Current Position

if (position >= 2 * jumpValue && position <= Overall.Trees.Count - 2 * jumpValue -
     1) {
    TimelinePos = 2;            // Middle Position
}
else if (position < 2 * jumpValue) {
    TimelinePos = position == 0 ? 0 : 1;        // First or Second Position
} else {
    TimelinePos = position == Overall.Trees.Count - 1 ? 4 : 3; // Last or
    Penultimate Position
}
```

Listing 11: This code block shows the part of the CurDisplayTreeIndex setter that is responsible for selecting which time slot the current tree has in the timeline.

The update of the timeline `Data`-objects can be seen in the Listing 12. In this `for`-loop the thumbnails position is represented by the offset. This variable shows where the thumbnail is respectively to the current city thumbnail. Here are some examples to help understand the offset:

- `TimelinePos = 2, i = 3 -> offset = 1`:
  This means that the thumbnail will be one time slot in the future.
- `TimelinePos = 2, i = 0 -> offset = -2`:
  This means that the thumbnail will be two time slots in the past.
- `TimelinePos = 4, i = 0 -> offset = -4`:
  This means that the thumbnail will be four time slots in the past.
- `TimelinePos = 1, i = 4 -> offset = 3`:
  This means that the thumbnail will be three time slots in the future.

To get the new indexes of the trees the `CurDisplayedTreeIndex` (main city's tree index) gets added or subtracted by the timeline jump value multiplied by the offset-amount. If the outcome of this addition is out of bounds, the last index will be picked. If the outcome of the subtraction is out of bounds, the first index will be picked. After selecting the index the thumbnail gets updated by loading in the new tree and pointers.

```
for (int i = 0; i < Timeline.Length; i++)
{
    int offset = i - TimelinePos;
    int timelineTreeIndex = CurDisplayedTreeIndex + jumpValue * offset;

    if (timelineTreeIndex < 0) { timelineTreeIndex = 0; }
    if (timelineTreeIndex >= Overall.Trees.Count) { timelineTreeIndex = Overall.
    Trees.Count - 1; }

    Timeline[i].Tree = Overall.Trees[timelineTreeIndex];
    Timeline[i].PointsToMap = Overall.PointsToMaps[timelineTreeIndex];
    Timeline[i].PointedFromMap = Overall.PointedFromMaps[timelineTreeIndex];
    Timeline[i].Layer = 11 + i;
}
```

Listing 12: This code block shows the part of the `CurDisplayTreeIndex` setter that is responsible for the update of the timeline's `Data`-objects. This assures that the underlying data of each city in the timeline is updated.

4.3.2 Reordering the Timeline

After setting the `CurDisplayTreeIndex` property the `UpdateTimeline` method is always called. This method sets the `TimelineTexts`, enables and disables the `TimelineSlots` as well as updates the culling mask of every camera. By updating the culling mask of the cameras we display the right camera in the right time slot. By using the `TimelinePos` property we can find out where the main city should be displayed in the timeline. By calling this method the timeline gets updated and the user sees the newly generated cities in the timeline. As we can see in Listing 13 the reordering happens by giving the respective `Timeline`-object the right culling Mask.

```
int offset = Settings.TimelineJump;

switch (Data.TimelinePos)
{
    case 0: // First Slot
        ...
        Timeline1.cullingMask = 1 << 10; // main City
        Timeline2.cullingMask = 1 << 11;
        Timeline3.cullingMask = 1 << 12;
        Timeline4.cullingMask = 1 << 13;
        Timeline5.cullingMask = 1 << 14;
        break;
    case 1: // Second Slot
        ...
        Timeline1.cullingMask = 1 << 11;
        Timeline2.cullingMask = 1 << 10; // main City
        Timeline3.cullingMask = 1 << 12;
        Timeline4.cullingMask = 1 << 13;
        Timeline5.cullingMask = 1 << 14;
        break;
    case 2: // Middle Slot
        TimelineText1.text = "Time: " +
            (Data.CurDisplayedTreeIndex - 2 * offset + 1);
        TimelineText2.text = "Time: " +
            (Data.CurDisplayedTreeIndex - offset + 1);
        TimelineText3.text = "Time: Now";
        TimelineText4.text = "Time: " +
            (Data.CurDisplayedTreeIndex + offset + 1);
        TimelineText5.text = "Time: " +
            (Data.CurDisplayedTreeIndex + 2 * offset + 1);

        TimelineSlot1.enabled = false;
        TimelineSlot2.enabled = false;
        TimelineSlot3.enabled = true;
        TimelineSlot4.enabled = false;
        TimelineSlot5.enabled = false;

        Timeline1.cullingMask = 1 << 11;
        Timeline2.cullingMask = 1 << 12;
        Timeline3.cullingMask = 1 << 10; // main City
        Timeline4.cullingMask = 1 << 13;
        Timeline5.cullingMask = 1 << 14;
        break;
    case 3: // Penultimate Slot
        ...
        Timeline1.cullingMask = 1 << 11;
        Timeline2.cullingMask = 1 << 12;
        Timeline3.cullingMask = 1 << 13;
        Timeline4.cullingMask = 1 << 10; // main City
        Timeline5.cullingMask = 1 << 14;
        break;
    case 4: // Last Slot
        ...
        Timeline1.cullingMask = 1 << 11;
        Timeline2.cullingMask = 1 << 12;
        Timeline3.cullingMask = 1 << 13;
        Timeline4.cullingMask = 1 << 14;
        Timeline5.cullingMask = 1 << 10; // main City
        break;
}
```

Listing 13: The `UpdateTimeline` method, where the Timeline gets updated and reordered

5 Usage

We have already established all of the theoretical parts of the *Memory City* application's synchronised timeline view. We shall now examine the timeline's intended use in this section. Here, we take a look at a completed timeline and examine how it alters when we perform a time jump and how the timeline looks after altering the *timeline jump value*.

5.1 General Timeline

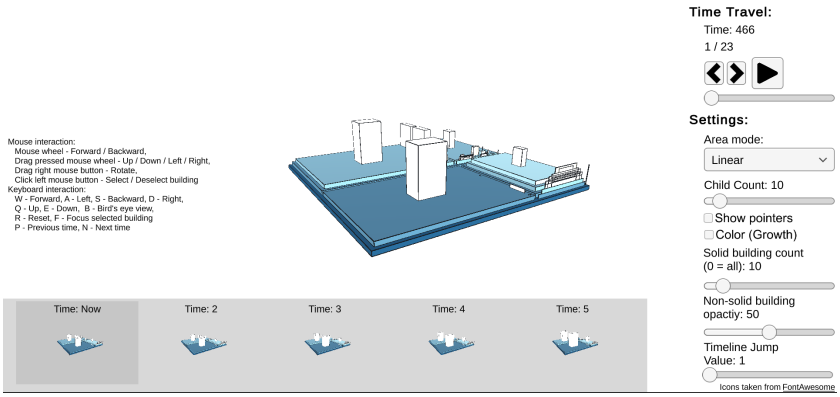


Figure 15: This image demonstrates the *Memory City* application with the synchronized timeline view.

In Figure 15 we can see the *Memory City* application with the synchronized timeline view. Because of the synchronized cameras the thumbnails, where different states of the cities are displayed, all show the camera from the same angle and use the same zoom level. If we now change the angle, the cameras adjust themselves accordingly. We can see this in Figure 16

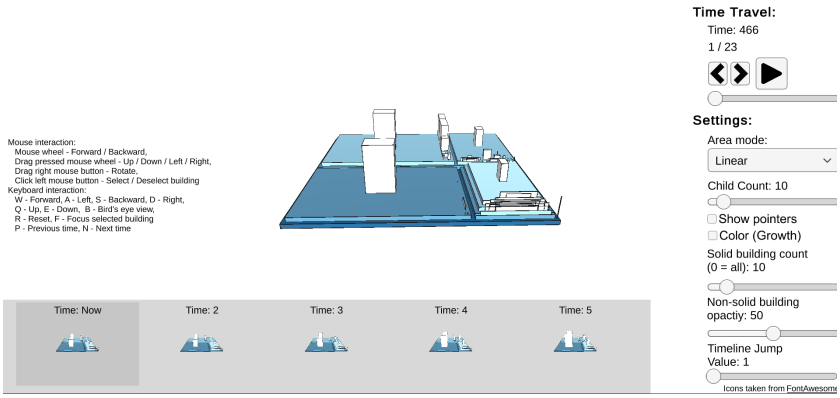


Figure 16: This image demonstrates the *Memory City* application with the synchronized timeline view when the camera has been moved.

5.2 Changing the Timeline Jump Value

When we now change the timeline jump value, our timeline should adjust itself. We can see the change in the timeline by looking at Figure 17. Here we see that the time slots of the timeline have been altered by having their period of time changed as well as their city has been updated to the respective state.

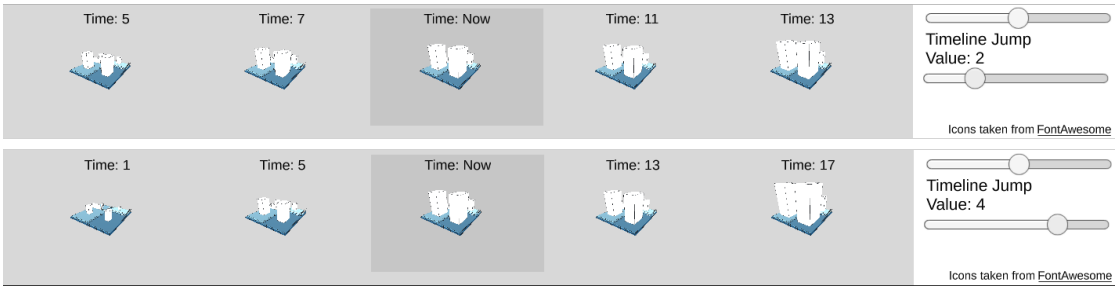


Figure 17: This image demonstrates the synchronized timeline view when the timeline jump value has been altered. On the top the timeline jump value is two and on the bottom its four.

5.3 Timeline with Pointers

To see the pointers in the timeline the only thing we have to do, is activate the pointers by clicking the "Show pointers"-button in the settings area and clicking onto a building from which we want to see the pointers. As we can see in Figure 18, in addition to the main city, also the cities in the thumbnails contain their respective pointer visualizations.

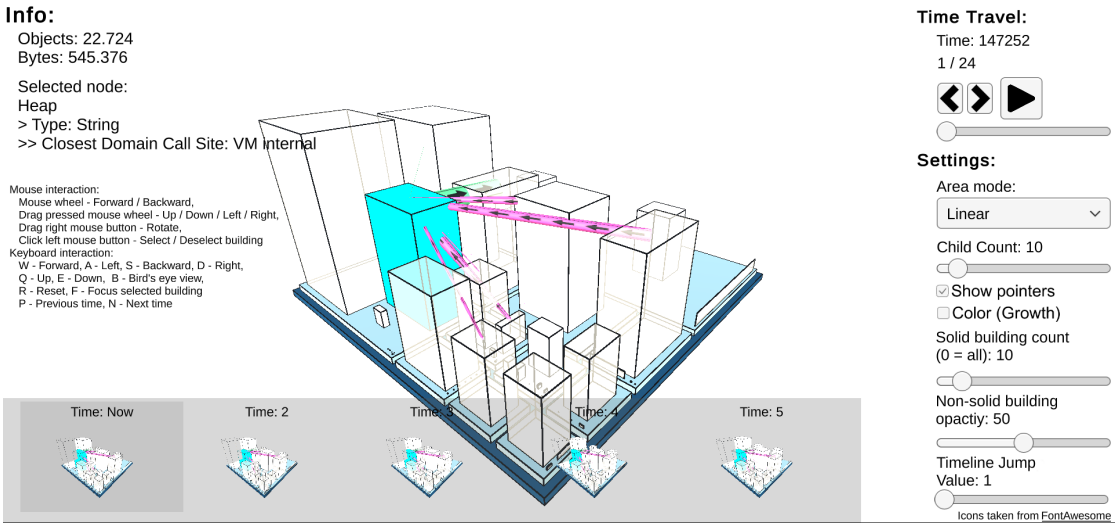


Figure 18: This image demonstrates a timeline view where the pointers are activated.

5.4 Timeline with Growth

In the given example, to see the growth better we adjust the *time jump* value to five and activate the setting for "Color (Growth)". As we now can see in Figure 19 the buildings are colored in our thumbnails and therefore we have it easier to find the memory anomalies.

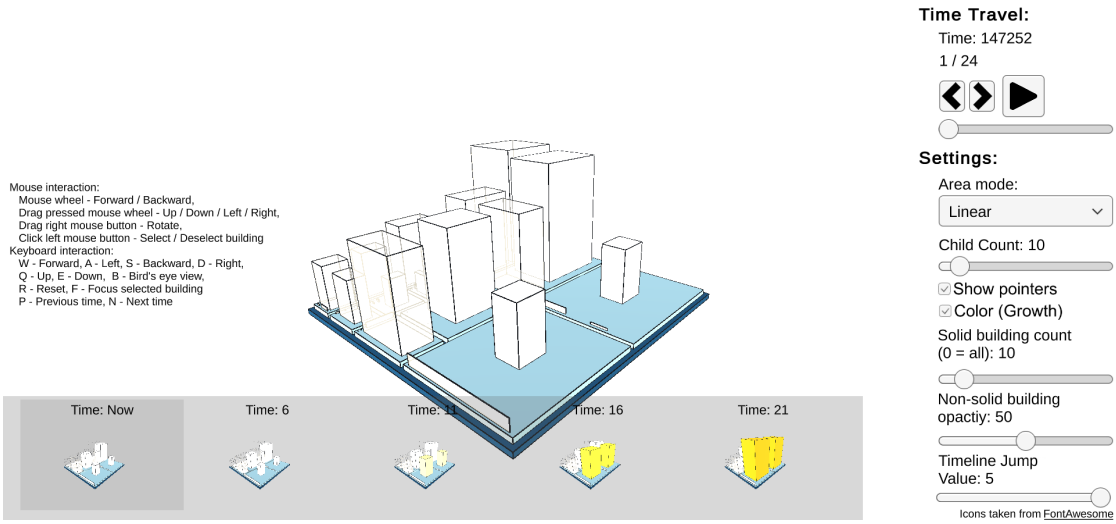


Figure 19: This image showcases a timeline view with growth coloring activated and a timeline jump value of five to see the evolution better.

5.5 Time travel through the Timeline

We can travel through time by clicking on the thumbnails of the cities. As we can see in Figure 20, we can go through the entire evolution of the heap memory and see that the timeline always adapts after every jump.

Figure 20: This image displaysvarious timelines for an application with 24 heap states. From top to bottom, the main city has been moved three heap states further. Thus, in the first line, the main city displays the very first tree and is thus on the very left of the timeline. Similarly, in the last line, the main city displays the very last tree and is thus on the very right of the timeline.

6 Conclusions and Future Work

The goal of this thesis was to introduce the reader to the *Memory City application* and explain how a synchronized timeline view could help the user to find memory leaks and anomalies faster. This thesis has shown how an architecture of such a solution could look like and explained how this problem has been solved by our approach.

This thesis explained how a synchronized timeline view may be implemented by the use of *Unity* and scripts written in C#. Further, it explained how with the help of this timeline the user had a simpler way to go through the different states of the heap. Additionally, we discussed how multiple cities can be generated and showed various approaches on how these cities might be placed on the screen at the same time. Moreover, the approach and implementation of camera synchronization, i.e., keeping the rotation and zoom level of all cameras filming the different cities synchronized, was elaborated. And finally the use of the timeline was showcased.

As for future work on the synchronized timeline view, the thumbnails do not show the same exact view like the main camera does of the *Memory Cities*. The reason behind this is that the *Render Textures* do not always have the same proportions as the screen does. As of now, the view dimensions of the render textures is always 16:9 but if the user uses a screen which has a different size it could look a little bit off.

Furthermore, it would be useful to reconsider how the camera itself works. With how it is right now, the camera is free and can move anywhere it wants. Thus, the user can just fly infinitely into one direction. It would be better if the camera was fixed on the main city and the movement was restricted to a certain area.

Listings

1 An newly created C# script for Unity. 7
2 The *SyncCamera* script is used to apply the movement of the main camera to the camera to which this script is attached to. 18
3 The constructor of the *Data* class where the main *city's* (Current) data and the data of the timeline cities (Timeline) are getting initialized 20
4 A part of the `DisplayCurrentTree` mehtod that shows where the different cities first calculate thes positions and sizes of their buildings (1) and then call the `DisplayNodeRecursive` method to display the buildings and districts. (2) 21
5 Calling the CreateMesh method and setting the layer of the created cube 22
6 Setting the culling mask of the cameras, ensures that only relevant buildings are recorded. 23
7 The selection handler when a building is clicked and the deselection handler when a building is deselected. 24
8 The `SelectedFullKeyString` property, which calls the `SelectionChanged` method for each city. 24
9 This listing presents the `SelectionChanged` method where the color of the selected building gets changed and the pointers are displayed when activated. 25
10 This code block is a part of the `DisplayCurrentNode` method, where the color of the current node is returned by the `GetBuildingColor` method. 26
11 This code block shows the part of the CurDisplayTreeIndex setter that is responsible for selecting which time slot the current tree has in the timeline. 27
12 This code block shows the part of the `CurDisplayTreeIndex` setter that is responsible for the update of the timeline's `Data`-objects. This assures that the underlying data of each city in the timeline is updated. 28
13 The `UpdateTimeline` method, where the Timeline gets updated and reordered . 29

List of Figures

1 An example of an visualization of the heap memory using the software city metaphor produced by the *Memory Cities* Application 1
2 A timeline displaying the past four years. 3
3 The visual attributes of a building in the *Memory City* application. 4
4 The basic structure of an *Unity* Program / Game. inspired by [5] 5
5 This screenshot of the *Memory City* application shows the application at use and has divided the UI into three sections 9
6 The general architecture and steps involved in integrating a timeline into the *Memory City* application. 10
7 The positions in which the "current" city can be. 11
8 Two example timelines one with a *timeline jump* value of one (top) and one with a *timeline jump* value of four (bottom). 12
9 The generation of cities while only using one scene and placing the cities far away from each other. 13

10 The generation of cities in multiple scenes and displaying them in the timeline view. 14
11 The generation of cities at the same spot utilizing layers. 15
12 On the top is the raw timeline canvas where we can see the structure and how the timeline will be placed together and on the bottom is an example of a "real" timeline. 17
13 The *timeline jump* value slider indicates how much the difference from one city to another should be. 18
14 A building with two pointers. 23
15 This image demonstrates the *Memory City* application with the synchronized timeline view. 31
16 This image demonstrates the *Memory City* application with the synchronized timeline view when the camera has been moved. 31
17 This image demonstrates the synchronized timeline view when the timeline jump value has been altered. On the top the timeline jump value is two and on the bottom its four. 32
18 This image demonstrates a timeline view where the pointers are activated. 32
19 This image showcases a timeline view with growth coloring activated and a timeline jump value of five to see the evolution better. 33
20 This image displaysvarious timelines for an application with 24 heap states. From top to bottom, the main city has been moved three heap states further. Thus, in the first line, the main city displays the very first tree and is thus on the very left of the timeline. Similarly, in the last line, the main city displays the very last tree and is thus on the very right of the timeline. 34

References

[1] Unity manual camera. https://docs.unity3d.com/Manual/Cameras.html. (last accessed on 25-April-2023).

[2] Unity manual canvas. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html. (last accessed on 25-April-2023).

[3] Unity manual scenes. https://docs.unity3d.com/Manual/CreatingScenes.html. (last accessed on 25-April-2023).

[4] Unity manual uses of layers. https://docs.unity3d.com/Manual/use-layers.html. (last accessed on 25-April-2023).

[5] Unity tutorial. https://www.tutorialspoint.com/unity/unity_tutorial.pdf. (last accessed on 25-April-2023).

[6] M. Weninger, L. Makor, and H. Mössenböck. Memory leak visualization using evolving software cities. *Softwaretechnik-Trends*, 39:44–46, 2019.

[7] M. Weninger, L. Makor, and H. Mössenböck. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Proceedings of the Working Conference on Software Visualization (VISSOFT) 2020*, pages 110–121, 2020.