Data Management 2021/2022 - Project Option 1:
Design and implement a software for visualizing the External Multi-Pass Sorting Algorithm (with varying size of the relation and varying number of frames in the buffer)

Marco De Luca - 2017104

January 17, 2023

Contents

1 Project Overview

The topic of the project was to design and implement a software to visualize the External Multi-pass Sorting Algorithm. The software can be used via browser and a Github Page was setup to run it without the need to download/install anything but one of the compatible browsers. Alternatively, it is possible to run the software by downloading the project repository and spawning a web server in the root.
The purpose of the project is to produce a software which can be used to better visualize and understand the External Multi-Pass algorithm. This influenced the software design, as the main design goal was to offer simplicity and high clarity while the algorithm is working. This led to the decision of implementing a simple scene inside which the frames and data blocks move while being always visible. Specifically, the user is not required to interact with the software during its execution except for the initial selection of the parameters.

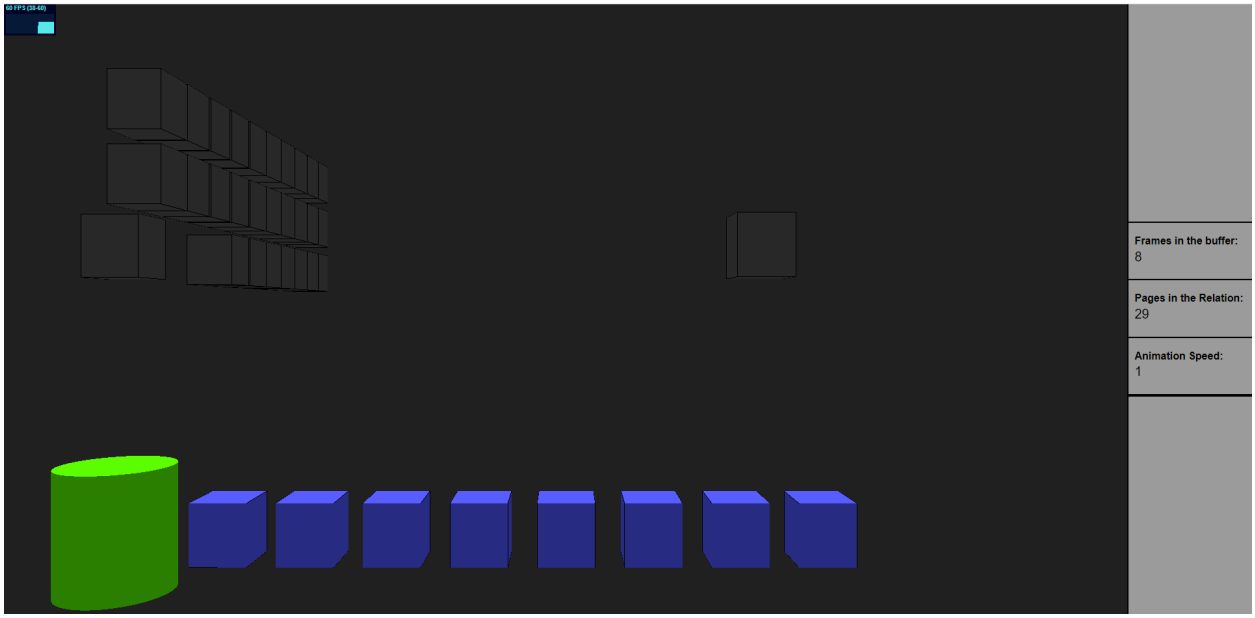

Figure 1: Overview of the Animation at run-time.

2 Design

2.1 UI

The first element I designed was the UI. Ever since the beginning of the project, I tried keeping everything as simple for the user as possible, allowing him to avoid a direct interaction with the simulation. In Fig. 2 there is a sketch of the UI: initially I had planned to have the canva occupying the whole page, with the input areas (pages, frames, speed) and the start button in the lower part.

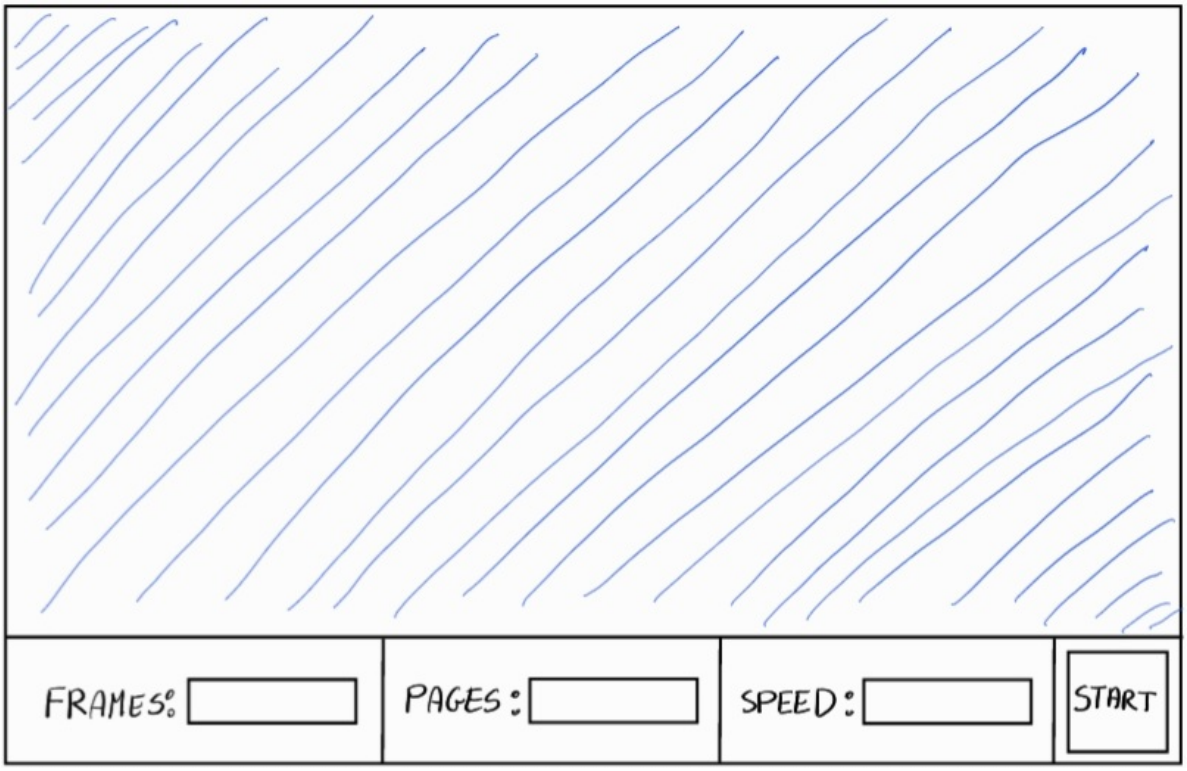

Figure 2: First version of the UI. The blue part is the canva.

Figure 3: Final version of the UI.

Once I started implementing the animation of the project I realized that since I was not going to allow an extremely high number of pages/frames there was no need to have a manual insertion of the input and instead this could have been implemented with sliders. In addition, I saw that the space was enough to allow me to move the UI on the right, having it vertical instead of horizontal. This eventually led to the final version of the UI shown in Fig. 3.

2.2 Elements on screen

Since we are working with data blocks, it made sense to represent them as cubes. In the bottom left part of the screen there is also a cylinder, which abstractly represents the DB. No other elements are used.

2.3 Subdivision of the space

I split the canva's space in four main areas, which are represented in Fig.4.

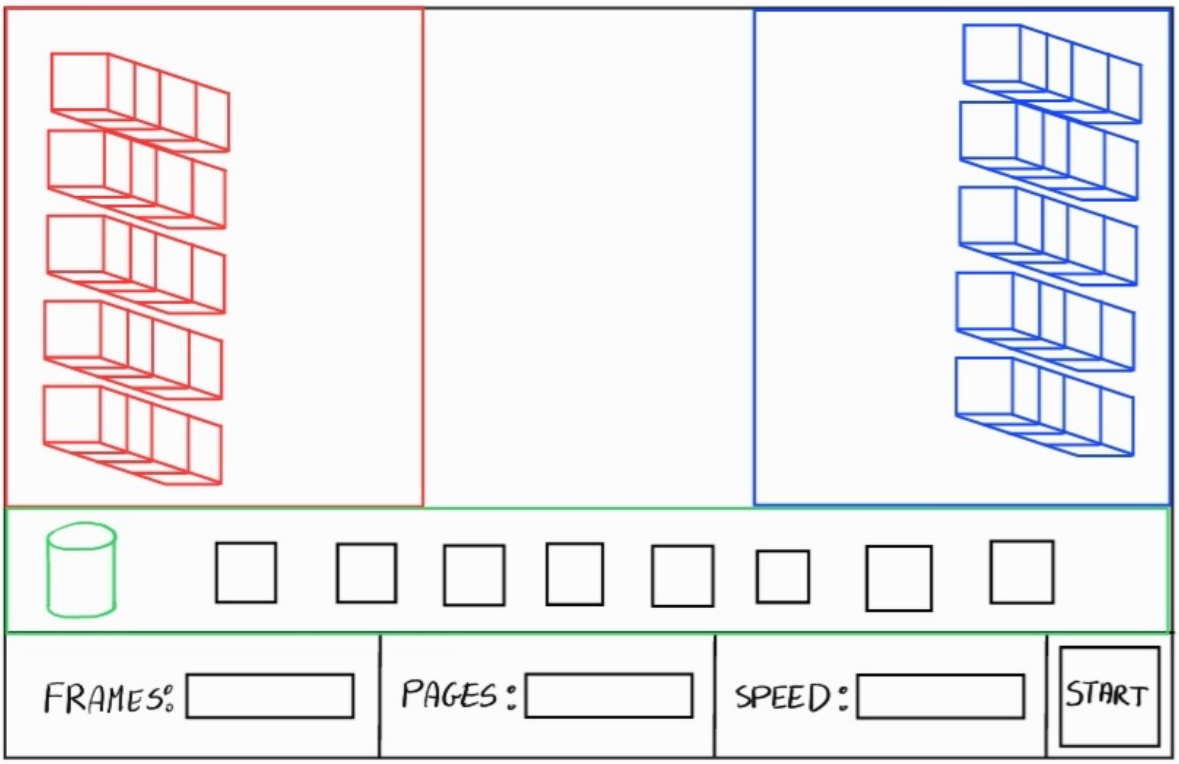

Figure 4: Subdivision of the space.

The **Red** area represents the "Input" space. This is the space which contains:

- During Step 0, the relation's pages before they are read;
- During Step i, the runs awaiting to be ordered.

The **Green** area represents the "Frames" space: This space contains the frames (as many as the user specified) and the pages/runs which are transferred here from the Red area. In addition to frames, in the left part there is also the DB cylinder, from which the relation page are extracted before being moved to the Red area.
The **Blue** area represents the "Output" space. This is the space which contains:

- During Step 0, the runs created by reading and sorting the relation's pages;

- During Step i, the runs obtained by merging the previous runs. Eventually it will contain only one run once the whole process is finished.

Finally, the **"RGB"** area (the one in the middle) is not specifically dedicated to anything, but it is traversed by the blocks moving from an area to the other.

2.4 Animation

2.4.1 Step 0

The first thing needed to do during step 0 is to load the relation's pages from the database into the free frames. To abstractly show that the pages come from the DB, there is an initial animation of the relation pages (the black blocks in Fig. 6) coming out of the DB cylinder and going into the Red area where they are disposed in lines of (at most) 10 blocks each. Then blocks need to be moved into the frames. In order to do this, each block is first moved on the Z axis and aligned with the frames. Then, the block is moved on the X axis to align it with the specific frame in which it'll be inserted. Lastly, the block is moved on the Y axis to be inserted inside the frame. Fig. 5 shows a sketch of both the initial movement from the DB and the final three movements performed by each block to be loaded into the frames.

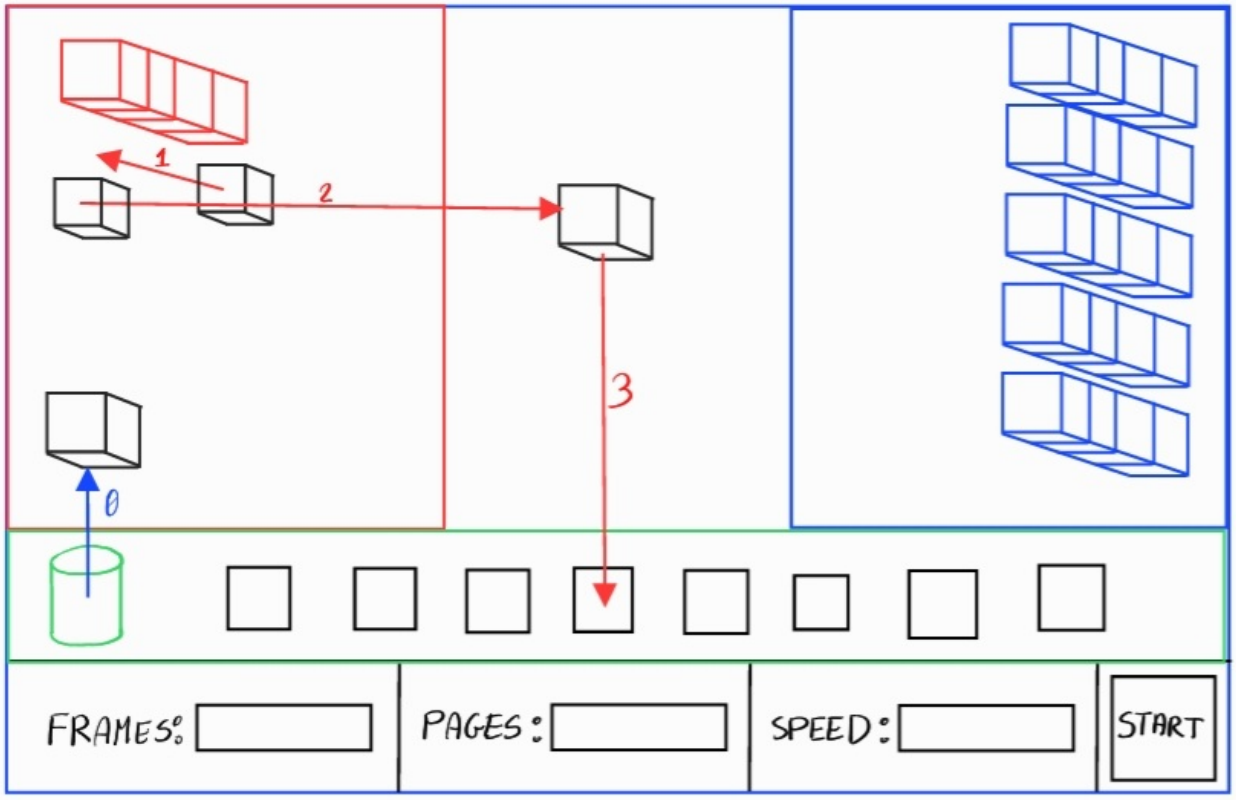

Figure 5: Sketch of Step 0. The page frames initially move from the DB cylinder to the red area (blue arrow labeled 0) in order to better display them. Then each page is loaded into the frame following the movement described by the red arrows labeled 1, 2 and 3. Although not shown in picture, the initial movement from the DB includes (in addition to the one described by the blue arrow) a second movement parallel and inverse to the red arrow 1. Arrow 1 is parallel to the Z axis, arrow 2 to the X axis, arrow 3 to the Y axis.

After loading the pages inside the frames, they need to be read to produce the runs. To simulate this procedure the frames blocks are gradually filled, becoming green (from black) over time. This is executed by choosing a block and increasing its "green fill" portion by 10%. After the same block has been selected 10 times, it has become fully green, and it means that every element of the block has been read and inserted in the run. In the real algorithm, we need to choose the opportune

element based on the fact that we want to sort the pages. Since this is a visual simulation, I simply choose the block at random.
Lastly, while reading the pages it is also needed to be creating the runs. This is executed by having the blocks "appear" inside the Blue area. The size of each block is initially 0%, then increases by 10% every time an element is read in the frames. This means that after reading 10 elements inside the frames there's one whole block of a run. To make this more understandable, the run's blocks color alternate between yellow and white.
Eventually, all the frames will only contain green blocks. This means that all the pages currently inside the frames have been read and sorted. If there are still pages to read in the Red area, frames are emptied and this procedure is repeated to create a new run (which will be placed directly under the previous one). If there are no more pages but there are more than just one run, we move on to execute step i. Finally, if there are no more pages and there is only one run, then the algorithm is completed in just one step.

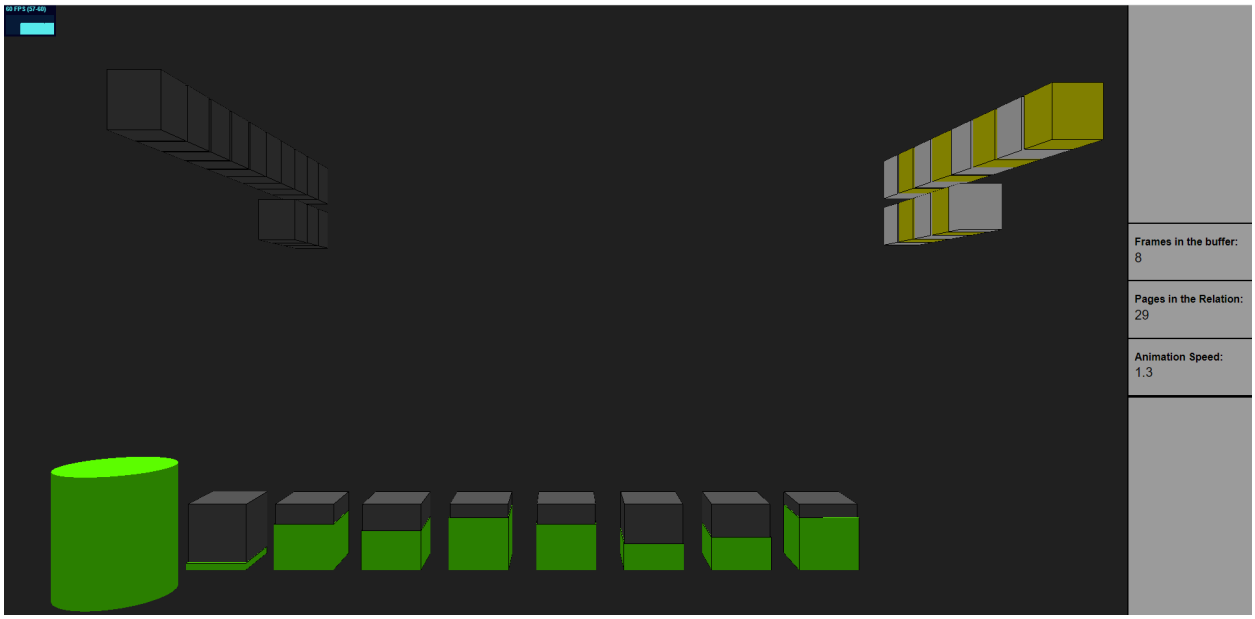

Figure 6: Step 0. Read the relation (black blocks on the left) using the frames (partially green filled blocks in the center) to create runs (blocks on the right). Every horizontal sequence of blocks on the right is a run. In this picture, we have a 29 pages long relation and 8 frames, so we'll create 3 8-blocks runs and a fourth 5-blocks run.

2.4.2 Step i

During step i, runs created in step i-1 need to be sorted together. The first thing needed is to move the runs created (which are in the blue area) to the left red area. Then, in step i not all the frames are used for reading but the last one is kept to write the output in it. This is the right-most frame, which is highlighted in a different color (purple in the final version). We generally don't need to use every frame, unless $number\ of\ runs \geq number\ of\ free\ frames - 1$.
To read a block from a run, we need to load it into a frame. Therefore the same 3-movements animation described in Fig. 5 is executed. Once the frames contain the runs block, their scansion is simulated in the same way as described in step 0. The only difference is that instead of creating a run block in the Blue area, the output frame is filled. Once the output frame is full, the block inside it is moved to the blue area to be aligned with the rest of its run. I didn't make a sketch of this, but the animation is analogue to the 3-movements animation used to load a block inside a frame, only this time the block leaves the frame to reach the run in the blue area.
Lastly, the output frame is filled using a different color then the other frames, to make it better

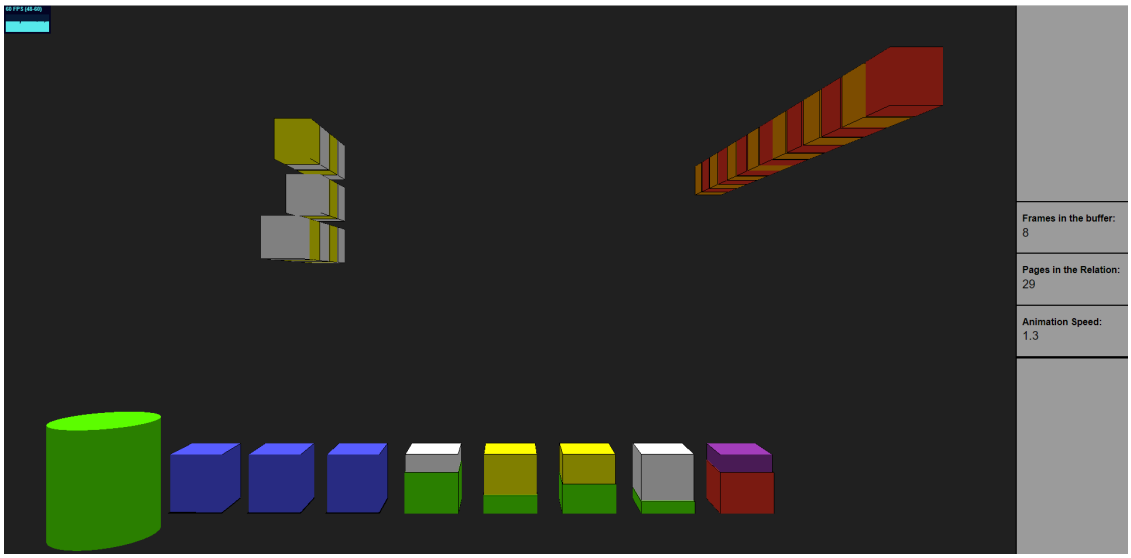

Figure 7: Step i. Read the runs created during step i-1 (yellow and white blocks on the left) using the frames (partially green filled blocks in the center) to fill the output frame (purple frame partially red filled). Eventually, when the output frame is full, copy it in memory by placing it together with the new runs (red and orange blocks on the right). Every horizontal sequence of blocks on the right is a run. In this picture, we have a 3 8-blocks runs and a fourth 5-blocks run, so we'll only need to perform step i once and we'll have a fully sorted 29-blocks run.

understandable that we are not performing the same operation in that frame (we're writing inside that frame instead of reading from it). More specifically, it alternates red and orange colors.
Once step i is completed, if there is more than one run, then they are moved from right to left and we iterate. The runs moved on the left also change color, becoming white and yellow from red and orange. If instead there is only one run, it means we're done because that run will be the full relation completely sorted.

2.4.3 Final result

Once the relation has been completely sorted, we have just one run on the right.

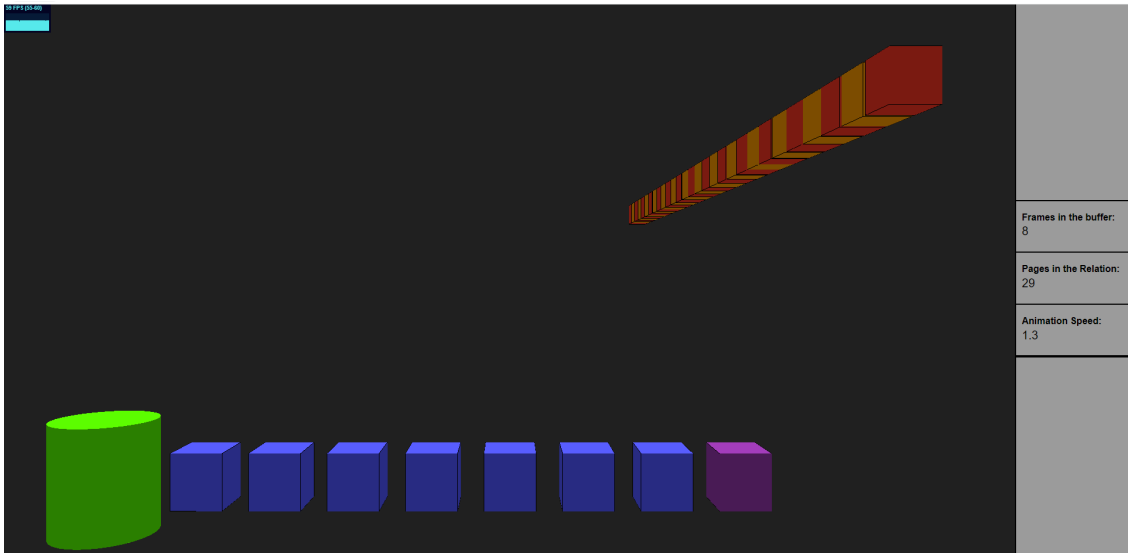

Figure 8: Final result. In this picture we have a 29-blocks run on the right containing the fully sorted 29 pages relation.

3 Software Architecture

The project consists of a simple HTML page containing the selectors for the initial setup and the canva used to render the 3d animation. Everything else was implemented in Javascript.

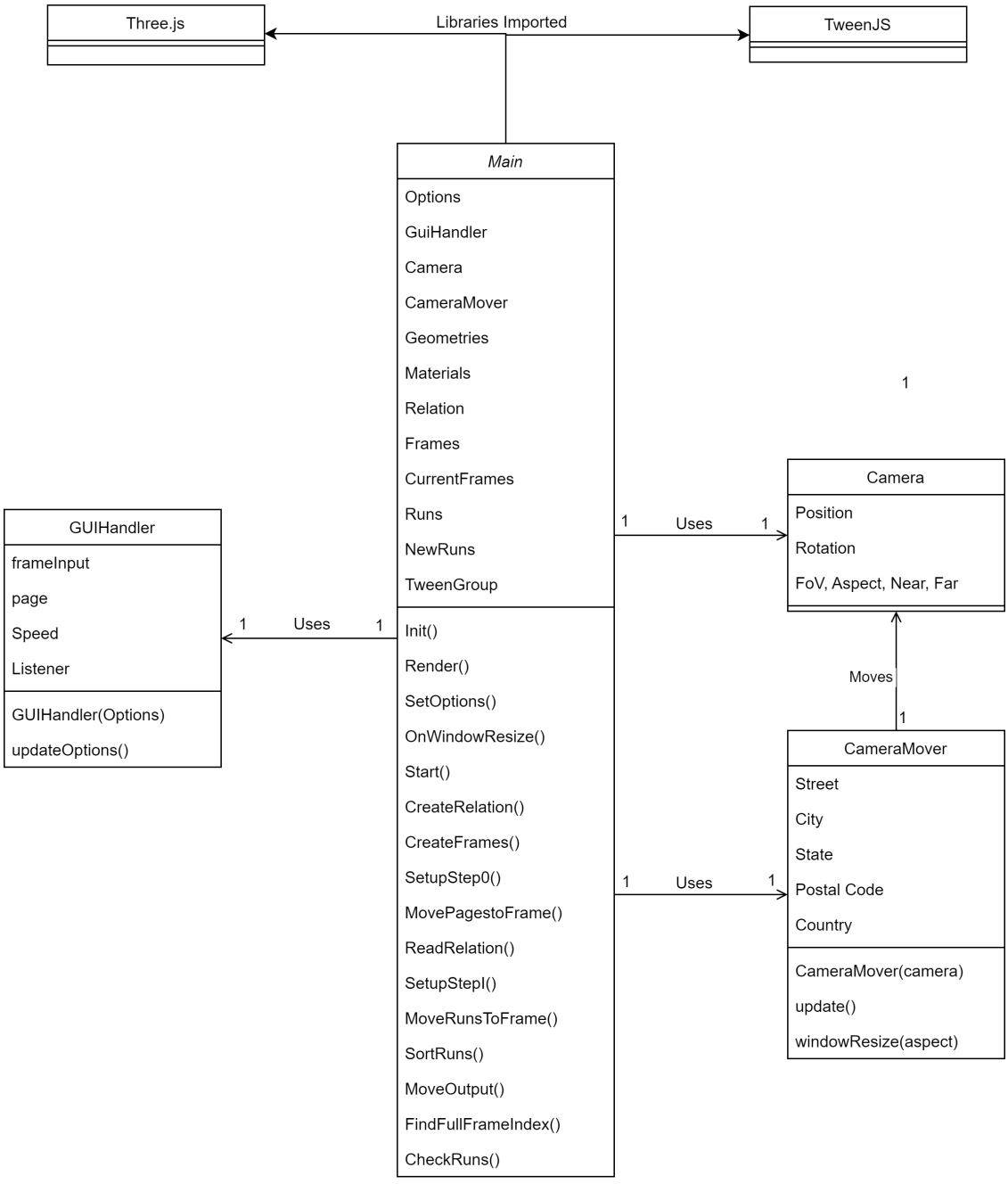

Figure 9: UML Diagram. The Main "class" was implemented following a more functional approach. Additionally, the Stats.module.js library is missing from this schema as it is not actually important in the project design in itself but it was only added to show the performances.

3.1 Javascript libraries used

3.1.1 Three.js

Three.js is a cross-browser JavaScript library and API which can be used to create and display animated 3D computer graphics in a web browser using WebGL. To code using Three.JS it is just sufficient to download the source code which is hosted in a repository on GitHub and then import it into the project. Generally speaking, to implement animations in Three.js it is sufficient to create a scene with a camera inside. After this, it is just needed to insert the desired models in the scene and actually implement the animation. Hence, Three.js was chosen because it allows to easily create 3D animations using JavaScript inside any compatible browser. Specifically, Three.js runs in all browsers supported by WebGL 1.0.

3.1.2 Tweenjs

Tweenjs is a JavaScript tweening engine. A tween (from in-between) is a concept that allows you to change the values of the properties of an object in a smooth way. We can decide how long it should take, and if there should be a delay, and what to do each time the tween is updated, whether it should repeat and other things.
Tweenjs was used to handle animations of the models created by Three.js.

3.1.3 Stats.module.js

This is a JavaScript Performance Monitor: this class provides a simple info box that helps in monitoring performances. It allows to see FPS, MS Latency (Milliseconds needed to render a frame) and MB of allocated memory.

3.2 Utilities developed

3.2.1 CameraMover.js

This class handles the camera's positioning and orientation. In the final version of the project, the camera is only moved at the beginning in order to have everything in the scene inside its field of view.

3.2.2 GUIHandler.js

This class handles the GUI in the HTML page. It listens to changes in the sliders and modifies the opportune variables, allowing the user to change the relation size, the number of frames and the animation speed. It also listens to the Start button.

3.3 Main.js

Since the project is quite small, I haven't created a dedicated class/library to manage the animation. Instead, I opted to just create the few animation functions directly inside the main file. Being the main, it also manages the initial setup, so it creates everything in the scene and makes the opportune calls to the utilities.

4 How the animation was coded

As previously said, the animation was fully coded inside the *main.js* file.

4.1 Creating the scene

When using Three.JS we first need to create a *Scene*: as its name recalls, this is a hierarchical object which contains all the other objects which will be displayed (lights and cameras as well). Then, we need to insert our objects inside the scene.

4.1.1 Lights

First of all we need *lights*, as otherwise it would all be dark. Three.js offers many kinds of lights, but in this scene I only added an ambient light and a directional light. The former is needed to make everything visible, while the latter helps the object have more of a 3d feeling, as otherwise their colors would be all exactly equal.

4.1.2 Objects: Geometry & Materials

After inserting the lights, we need the *objects*. To create an object, we need to define a geometry (cubes for our blocks, a cylinder for the DB) and a material. As an additional element, I also defined an EdgesGeometry for the cube to make them have black edges so that they are more easily visible as 3d cubes. Regarding materials, Three.js has many options, varying in the spectrum of its realism and its computational weight. When we have too many objects on screen, if they're also using complex materials it might take a while for the software to load and render them. Since quality of the materials was not really too needed, I made every material as a *MeshPhongMaterial*, which is not the most realistic material, but it is good enough while not being too heavy. I created one material for every color I used (plus a duplicate of the black which was needed for the edges), and they are all loaded at the beginning and accessed as needed when an object is created/changes color.

4.2 Setup Step 0

Once the relation pages and frames have been created, they need to be animated. Most animations I made are coded using tweenjs: this library allows to change the value inside a variable (say from X to Y) over a specified time T. It is very useful as it offers a lot of methods to manage the animation, such as OnStart, OnUpdate, OnComplete. Specifically, I used tweens to move the blocks over time. This means that every movement of a block has been implemented by creating a tween which modifies the block's position variable.

4.2.1 Load the pages from the DB

To perform this animation, every page is created inside the DB cylinder, so it's initially hidden to the user's sight. Then two tweens are applied to the cube's position value: the first tween moves it vertically on the Y axis, the second moves it horizontally on the Z axis. This procedure is iterated for every relation page the user has specified, and eventually the red area will contain all the pages. To avoid a single line of the relation (which in case of too many pages would mean having blocks very far from the camera), I made it so that the relation is split in lines of 10 pages.

4.2.2 Move the relation pages into the frames

As previously described in 2.4.1, moving the pages into the frames requires a 3-movements procedure. Therefore I used three tweens to move every page. The first tween moves the page on the X axis to

align it with the frames; the second tween moves the page on the Z axis to reach the desired frame's Z coordinate; the third tween drops the page inside the frame by moving it on the vertical Y axis.

4.3 Reading the relation

4.3.1 Read a page

To simulate a page being read, I decided to make it so that the block changes color over time. However, it is not possible to change just a portion of the block's material, unless we use textures but this would have been more complex to implement and computationally heavy, as every block would have needed to have the texture applied and edited. Since I was using very light cubes, I decided to just create an additional (green) block covering the (blue) frame containing the (black) page being read. Notably, this is not a very heavy solution as the blocks for the runs need to be created anyway, so only the frames' blocks are being duplicated. To show the page being read overtime, the green block is initially set with an height of 0, and every time the block is read (which happens randomly), its height is increased by 10%. After being read 10 times, the block is completely green, which means it has been completely read.

4.3.2 Creating the runs

While reading the pages, runs also need to be created. This is animated simultaneously to the reading: a cube is created to indicate the new block of the run. This block starts with a width of 0, and every time a page is read its width is increased by 10%. Once the block has a width of 100%, if the current pages haven't been all completely read, a new block is created. Notably, the blocks' color alternates between white and yellow to offer a better understanding. Otherwise, given the perspective it could be hard to distinguish the various blocks once the run becomes longer.

4.3.3 Once the frames are all completely green

When all the pages currently in the frames have been completely read, if there are still pages to read the current frames are emptied and we iterate. Additional runs are on a lower Y axis to separate them from the previous one. If there are no more pages, we check if the number of runs created is 1 or higher. If it is 1, then we're done. If it is higher than 1, we continue to step i.

4.4 Setup step i

After completing step $i-1$, if we need to execute step i it is because there are at least two runs on the right blue area. This means they first need to be moved from right to left, and this is done using a tween. In addition, the color of the right-most frame is also changed to purple to make it understandable that the algorithm is going to write in that frame instead of reading from it.
Once the runs are on the left, their blocks need to be moved inside the frames. This is done using tweens similarly to the way the movement of pages was animated in step 0. The only difference is that the right-most block will not be loaded with a block from a run. Additionally, blocks are not just picked in sequence as it was in step 0, but only one block at a time is taken from each run.

4.5 Creating the new runs

To simulate reading and sorting of the runs, the blocks inside the frames are read in the same way the pages were read during step 0. The difference is that in this case the new runs' block is not directly created in the blue area, but instead it is written in the output frame. Once this is full, the block is moved into the blue area together with the other blocks of the new run. As before, blocks alternate in color, only this time they alternate between red and orange.

4.5.1 Once the runs currently being scanned have beeen completely sorted together

When all the runs currently being analyzed have been sorted, if there are still runs to sort the current frames are emptied and we iterate. Additional new runs are on a lower Y axis to separate them from the previous one. If there are no more runs, we check if the number of new runs created is 1 or higher. If it is 1, then we're done. If it is higher than 1, we iterate step i.

5 Further Development and Project Issues

5.1 Real-Time interaction

While developing the project I was mainly focused on keeping the user experience as simple as possible, and the resulting work is completely non-interactive. I believe this to be a positive aspect of the final product, since it allows the user to clearly visualize the algorithm in function without losing focus because of the interaction.
On the other hand I also realize that some real-time interaction might be beneficial

5.1.1 Camera Movement

By allowing the user to move the camera, he could watch the runs while they're being created from an other point of view allowing him to focus on that specific part of the algorithm. This is not possible in the current version as the camera is locked.

5.1.2 Pausing the animation

It is not possible for the user to pause the animation while it is in progress. Similarly to the camera movement, this might be an issue as it prevents the user from analyzing specific moments of interest of the animation.

5.2 Animation Reset

The current version of the project does not allow the user to reset the animation. This means that the only way to stop the project from going on is to refresh the page, which is not user-friendly.

5.3 Run-time speed/frames change

5.3.1 Change in the number of free frames at run-time

This is probably one of the most complex updates to introduce. However, it is also an improvement which would greatly increase the realism of the application, since in a real case scenario the number of free frames might not be constant over the execution of the algorithm. In particular, one could implement an interactive version where the user is able to change the number of free frames at run-time. Alternatively, it would also be possible to have the number of free frames change randomly every X seconds.

5.3.2 Changing the animation speed at run-time

In the current version the user needs to specify the animation speed at the beginning and this remains constant throughout the whole execution. It would however be beneficial to allow the user to change the speed dynamically.

5.4 Increase the maximum number of frames/pages

If one were to allow the user to increase this number, then it would also be needed to either reduce the size of every block or to modify the camera, allowing it to move and possibly modifying the whole animation design. However, if we reduce the block size, they might be too many small objects on screen for the user to follow. As a consequence, he might struggle to correctly visualize what is happening.
On the other hand, we could keep the block size as it is, and move the camera around to follow the action. However, the project was not designed with this possibility in mind, therefore a similar approach would probably lead to a substantial overhaul of the project.

In my opinion, the current version already sort of minimizes the size of each block while maximizing the number of frames/pages on screen without adding too much cluttering. In addition, having more pages/frames on screen is not necessarily going to help visualize the algorithm better, as the fundamental concept of the algorithm remains the same. The only benefit I could see in increasing the maximum relation size is that it would allow the user to setup an execution with more than two or three steps.

5.5 Allow the user to personalize colors and/or block size

In the current version the user is unable to change anything in the animation. As previously said in 4.4, I tried to get the "right" block size to make the animation as understandable as possible. I also tried to pick colors in such a way for everything to be easily understandable and recognizable by the user, but indeed one might prefer different colors. In addition, the color palette I chose is not colorblind-friendly.

6 Browser Compatibility

I tested the project on Windows 10 and Windows 11. Tests were conducted both locally by spawning a Python3 server inside the root of the project and on the GitHub Page at the link down this section. Tests were conducted on the following browsers: **Google Chrome** Version 105.0.5195.127 (Official Build) (64 bits), **Microsoft Edge** Version 109.0.1518.55 (Official Build) (64 bit) and **Mozilla Firefox** Version 108.0.2 (Official Build) (64 bit).

7 Github Links

Github Repository Link: `https://github.com/MarcoDL99/DM-Multipass-Mergesort`
Github Pages Link: `https://MarcoDL99.github.io/DM-Multipass-Mergesort/`