AN EXPLORATION OF EDUCATIONAL ALGORITHM VISUALIZATIONS USING WEB TECHNOLOGIES

An Undergraduate Research Scholars Thesis

by

NKEMDI ANYIAM

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Dilma Da Silva

May 2022

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Nkemdi Anyiam, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

An Exploration of Algorithm Visualization for Educational Purposes

Nkemdi Anyiam
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Dilma Da Silva
Department of Computer Science and Engineering
Texas A&M University

Certain algorithms (such as those for dynamic programming (DP)) lack visualizations that can exhaustively explain each step while delivering intuitive animations, in part due to rigid layouts in the designs. In this paper, we show that these problems can be addressed using modern web technologies—namely HTML5/CSS3 and Javascript—by demonstrating an animation framework that lets developers create a timeline of animations that easily integrates into the flow of front-end web development. We also put forth and discuss design rationale and recommendations for algorithm visualizations in general.

The framework supports typical playback features like rewinding, changing playback speed, skipping, etc., and it allows developers to specify various parameters that let them fine-tune the animation sequences. Outside of that, we are free to incorporate any UI/UX designs that would aid students' overall comprehension, allowing a closer relationship between text explanations and graphics as well as connections between elements that would normally be isolated in panels.

To test the framework, we created a visualization of a DP algorithm for memoized weighted interval scheduling (WIS). WIS is tedious to solve by hand, so instructors typically skip iterations and expect students to have internalized the in-between steps. Our approach, however, takes user input and procedurally generates the visualization, including text explanations at every single step

of the way. Repetition can be crucial to understanding concepts in full, so by showing all of the parts that a professor would never have time to write down and providing an interface that supports useful playback controls, we have created a way to visualize algorithms that boosts intuitive design and supports different learning paces.

ACKNOWLEDGMENTS

Contributors

I would like to thank my faculty advisor, Dr. Dilma Da Silva, for her guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This work was made possible in part by the National Science Foundation under Grant Number NSF-SPX-1919181. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the National Science Foundation.

NOMENCLATURE

DP	Dynamic Programming
WIS	Weighted Interval Scheduling
WAAPI	Web Animations API
TAMU	Texas A&M University

1. INTRODUCTION

The area of visualizing algorithms has been studied since the 90's [1, 2], but many advancements have since occurred in tools and languages for web programming. We are revisiting the challenge of visualizing algorithms for the purposes of education, now using modern web support (including experimental features) from HTML5, CSS3, and JavaScript. More specifically, we are exploring a way to create a web animation framework that allows for the development of procedurally-generated visualizations that can be rewound, fast-forwarded, skipped through, and more—all without constraints on the normal front-end development and UI/UX design processes. Not only would tools built for this purpose be useful for clearly visualizing algorithms that would be difficult to fully demonstrate by hand, but they would also make it easier to explain challenging concepts that build *off* of said algorithms. For example, distributed systems went from being an expert topic to being one that is touched even in undergraduate studies, so it would be beneficial to have access to visualization learning tools that can help explain concepts in great details. After all, between the layout and design capabilities of HTML/CSS and the robust DOM-manipulation capabilities of JavaScript, anyone experienced in front-end web development technologies should be able to create visualizations that are both visually appealing and highly customizable. In this work, we will assess the feasibility of our goal by implementing our project, AnimTimline, and using it to develop a visualization for a dynamic programming algorithm for weighted interval scheduling. In this way, we are also putting forth and discussing design decisions and recommendations for animated algorithms in general as we test a way to exhaustively present dynamic programming algorithms with as close of a tie between the text and graphics as possible. Beyond that, the animation will represent the algorithm in a way that someone would actually solve by hand, making it more intuitive at a glance as the algorithm progresses. A professor would never have time to solve large problems by hand in front of a class without skipping certain steps or essentially fast-forwarding, but with our visualization, all of the text and every step are automatically generated based on the input, providing a level of repetition that would be excruciating to do by hand but is crucial to ensuring that students can see every single step in as much detail as they desire.

We will be using insight gained from other academic papers as guidance in determining what is and is not appropriate in terms of visualization best practices. For example, the proposal of a "type by task taxonomy (TTT) of information visualizations" [3] to address what actions a user should be able to perform to accomplish particular tasks within the visualization gives useful ideas on how to ensure that the user experience aligns with the type of data they are interacting with. Most pertinent, we propose, would be to "keep a history of actions to support undo, replay, and progressive refinement" [3]. For an educational tool, figuring out the best way to incorporate history into these visualizations is vital to ensuring that students can step forward and backwards through steps in the algorithms as they please.

2. BACKGROUND

2.1 What Is Dynamic Programming?

Let us go through a brief overview of what exactly “dynamic programming” is. In order to understand it, it may be helpful to have a basic understanding of “recursion”. Oftentimes, large problems can be broken down into smaller subproblems, and when those subproblems are solved, they are combined into the overall solution. Those subproblems may be broken down into even smaller subproblems as well. A common way to demonstrate this is using the "Fibonacci sequence", which is the sequence of numbers $a_1,\ a_2,\ a_3,\ ...,\ a_n$ such that $a_1 = 1$, $a_2 = 1$, and $a_n = a_{n-1}+a_{n-2}, n > 2$. The first eight numbers in the sequence are $1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21$. Except for the first two, each "Fibonacci number" is simply the sum of the two numbers preceding it. Suppose we wanted to compute the eighth Fibonacci number, which is 21; one way to solve it would be to use the following routine $\textsc{Fib}(n)$, where n starts at 8:

```
1: if $n == 1$ or $n == 2$ then
2:     return $1$
3: else
4:     $a = \textsc{Fib}(n-1)$
5:     $b = \textsc{Fib}(n-2)$
6:     return $a + b$
7: end if
```

Notice that in lines 4 and 5, we call the routine inside itself to compute previous Fibonacci numbers. This is recursion—we are computing the eighth number by breaking the problem down into finding the seventh and sixth numbers, which will break the problem down into finding the sixth and fifth number and fifth and fourth number respectively, and so on. When the "base case" is reached, which is when n is 1 or 2, we return 1 and go back up the recursion tree until we finally determine that $\textsc{Fib}(7)$ is 13 and $\textsc{Fib}(6)$ is 8. Thus, we return $a+b$, which is 21, which is our eighth

Fibonacci number.

This works perfectly fine, but the main issue with this implementation is that it computes the same problem multiple times. For example, recall that the subproblems in $\textsc{Fib}(8)$ were to compute $\textsc{Fib}(7)$ and then $\textsc{Fib}(6)$. $\textsc{Fib}(7)$'s subproblems are $\textsc{Fib}(6)$ and $\textsc{Fib}(5)$. We can immediately see that we are computing $\textsc{Fib}(6)$ twice, and each subproblem of the subproblems similarly computes numbers that we have already computed. This is incredibly inefficient and has what is called an "exponential time complexity". To put this in perspective, using this naive approach for $\textsc{Fib}(8)$ solves a total of 129 problems even though there were only 8 Fibonacci numbers total!

This is where dynamic programming (DP) comes into play. Unlike simple recursion, we can use DP to break problems into *overlapping* subproblems, and one technique is to use "memoization" (not "memorization"). Memoization involves saving the results of subproblems into some sort of data structure so that we do not have to reevaluate said subproblems in case we run into them again. For example, after $\textsc{Fib}(7)$ is solved, we already know that $\textsc{Fib}(6)$ was also solved in the process since that was one of $\textsc{Fib}(7)$'s subproblems; therefore, we can just store $\textsc{Fib}(6)$'s result in a table. Then, once we call $\textsc{Fib}(6)$ for the second time since it is $\textsc{Fib}(8)$'s next subproblem, we check the table to see if $\textsc{Fib}(6)$ was already computed, see that it is indeed there, and just use that. The same process is repeated at every step of the way so that we end up solving for every Fibonacci number exactly once instead of multiple times. At the start of a given call, we check the table to see if this Fibonacci number has already been found; if so, just use that value; if not, solve for the number and fill in the table entry with the result. It is clear to see that this is significantly faster on average than the naive approach; the downside is that we have to use space for the table.

There are more nuances to dynamic programming that deal with the conditions necessary to be eligible for DP and various types of DP problems that we will not discuss here. For more details, seek sources like [4] or online learning resources.

2.2 What Is Weighted Interval Scheduling?

We will give a brief explanation of interval scheduling and weighted interval scheduling, but the interested reader will seek more thorough explanations from sources like [4] or online references. In general, interval scheduling is a class of problems involving tasks—or jobs—whose intervals may or may not overlap over a given stretch of time. Each job is at least defined by its starting time and finishing time, and two jobs overlap (or are "incompatible") if their allotted times intersect. Given a set of jobs, suppose we want to figure out the maximum number of jobs we can perform without overlapping. Additionally, suppose we do not care about the importance of each job—in other words, they hold the same weight. Finding the maximum amount of jobs in this case is simple: First, we sort the jobs by their finish times such that jobs that finish first are placed before jobs that finish later, regardless of start times, giving us a sorted list of jobs $J_1,\ J_2,\ ...,\ J_n$. Then, starting from J_1, we work our way to the right and keep adding jobs to our chosen set. If a job does not overlap with the jobs we have chosen so far, we add it to our set; otherwise, we leave it alone. This is a "greedy" approach because we just pick any compatible jobs we see without question, and it works without fail (see page 120 of [4] for a proof by induction).

Things may not always be this simple, however. Suppose that the jobs are *not* of equal weight; now each job has any value above 0 associated with it—its weight. Now suppose that we want to find the maximum weight we can achieve without overlapping jobs. Currently, there is no greedy algorithm that is guaranteed to give the correct answer, so the alternative is finding every possible combination of compatible jobs and choosing the highest found total weight. We first figure out each job's most recent compatible job—that is, for each job J_j, we need to find the job J_i such that $i < j$, J_i does not overlap J_j (i.e. J_i finishes before J_j starts, so no overlap), and J_i finishes later than any other jobs preceding J_j in the sorted list. The index of each job's compatible job is stored in a separate table. Next, we start at the last-finishing job J_n and consider two possibilities. The first possibility is that J_n is part of the optimal sequence of jobs that produces the maximum total weight, in which case we add J_n's weight and then compute the rest of the chain behind it from this job's most recent compatible job (i.e., the most recently finishing job that does not overlap it). The second possibility is that J_n is NOT part of the optimal sequence of jobs that produces the maximum total weight, in which case we try to compute the chain using the job that finished most recently behind us (it does not have to be compatible since it does not need to worry about overlapping J_n). Then, for each possibility's subproblem, we check the same two possibilities on whatever job we end up on. Just as with $\textsc{Fib}()$, the final answer works its way back up to the top after the base cases are reached. Of course, such a task would have an exponential time complexity because we would end up resolving subchains of jobs several times in an effort to solve larger chains—just like with the naive approach for finding Fibonacci numbers. As you have likely surmised, this is a great use case for dynamic programming—once we find the maximum weight for a particular chain of jobs, we can store that result in a table so that we can simply use that value when the same chain of jobs is queried.

3. RELATED WORK

Work related to this project consists of research that similarly attempts to facilitate the creation of educational visualizations for various algorithms.

3.1 SREC

One such work is SRec, which is a system for animating recursive algorithms in Java programs to assist in algorithm courses [5]. Functionally, it incorporates several animation controls akin to those present on a VCR set, including pause, play, stepping forward and backward, and variable speeds. Such features allow the user to fine-tune their viewing experience as they try to better their understanding of the material being taught, which is important for students who learn at different paces or need to repeat parts. SRec also allows the user to jump over recursive calls in the same way that debuggers allow you to "step over", which would certainly be a useful feature for instructors who feel like the students have a good enough understanding to warrant skipping additional repetitions. In terms of graphically representing recursion, one of the ways it allows the user to view a recursive algorithm is a trace, where entering and exiting invocations is indicated using indentations (this is similar to how tools displaying the file organization in a computer show folders nested in the graphical interface). This is particularly effective for layouts that focus on taking up vertical space rather than horizontal space, as will be seen with our project.

In a later paper, the researchers extended the original SRec system by supporting the representation of dynamic programming algorithms [6]. We will not discuss the implementation details here, but an important thing to note is that the graphical representations are similar to that of the original system: simple blocks connected by lines. It seems that neither of the versions of SRec provide extensive textual details or similarly detailed graphics that would facilitate teaching/learning; we believe that it would be greatly beneficial if the animation exhaustively presented dynamic programming algorithms with as close of a tie between the text and graphics as possible, which was easily achievable with AnimTimeline in our project.

3.2 VizAlgo

VizAlgo is a similarly effective learning tool, allowing "students to experiment and explore the ideas with respect to their individual needs" [7]. As depicted in the paper, the initial version of VizAlgo includes panels that display the algorithm pseudocode, the algorithm visualization, and information about the algorithm itself to describe what it is actually doing. The user can choose from a list of algorithms, and the system features playback controls such as "Next", "Stop", "Play", and "Delay".

Over time, the platform featured more visualizations according to [8], including some dynamic programming algorithms. However, the layout of information appears to be quite limited; while there is more accompanying text than SRec, the "Information" panel only provides basic details about the algorithm being displayed, and there is no text truly explaining what is happening at each step. It also does not display tabular data in an aesthetically pleasing manner; it instead lists the entries as you would with typing in a standard word document, using spaces to separate columns rather than colored blocks or lines. Our visualization for weighted interval scheduling shows that no such limitations need to exist when constructing an effective visualization for a complicated algorithm.

3.3 VisuAlgo

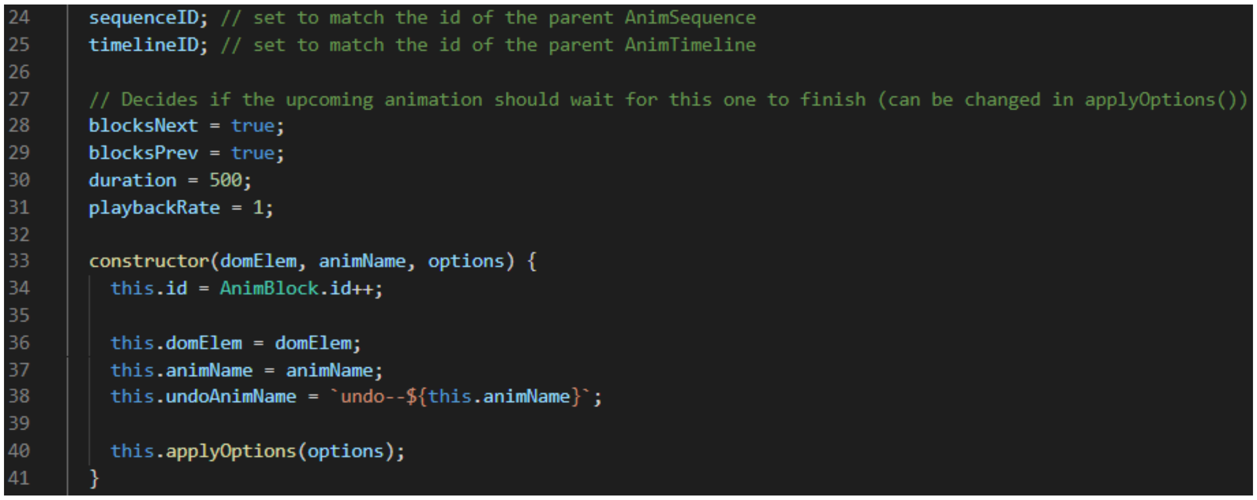
The last relevant work we will briefly discuss is VisuAlgo, which is freely available online and makes good use of graphics, text, and animations in its visualizations. It features animations for some dynamic programming algorithms, but they do not provide a visual trace that the user can use to see the progression of the entire algorithm at any given point, nor do the complementary text descriptions live very close to the graphics themselves, instead being disconnected in a box to the side. Additionally, the arrays used for memoization are not displayed for the user, which is a missed opportunity to visualize an incredibly important part of the computational process to the user. With AnimTimeline, developers can depict such things with ease by using already-existing web development layout tools and generating an animation timeline to manipulate them.

4. SYSTEM ARCHITECTURE

AnimTimeline consists of several parts. In this section, we will break them down in *great* detail and explain how they interact with each other. This description should give other developers ideas on how to implement similar systems and what important decisions to take into consideration to achieve bug-free solutions. Proficiency in JavaScript and a fair understanding of HTML and CSS are desirable in order to understand the implementation details, but high-level explanations are given when appropriate. We start with the smallest component of a given AnimTimeline—the AnimBlock.

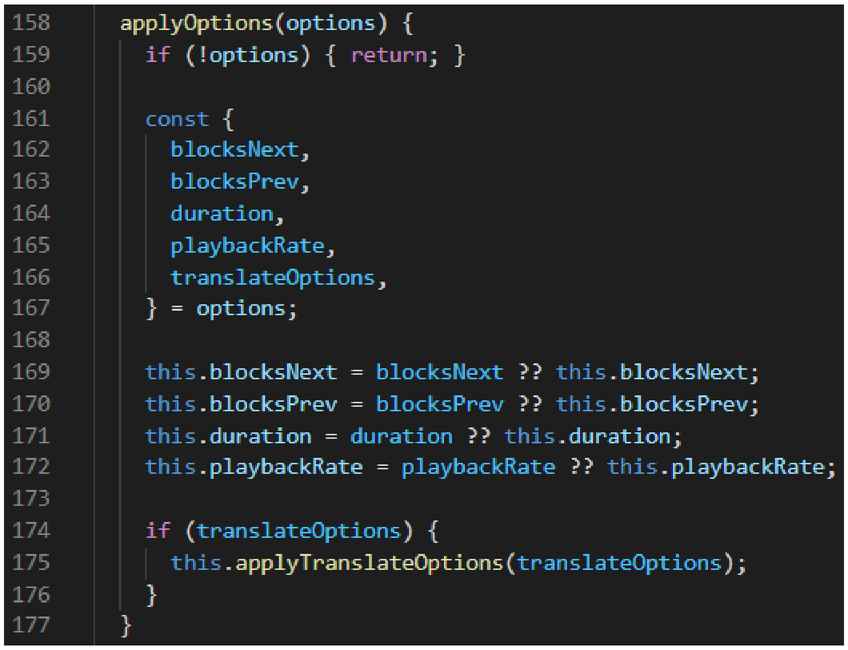
4.1 AnimBlock

The AnimBlock attaches an animation to one HTMLElement; it is the essential building block of the AnimTimeline framework. **Figure 4.1** shows the constructor for the AnimBlock class. *domElem* references the aforementioned HTMLElement object. The *animName* parameter is a string that holds the name of the animation that will be performed on *domElem* when stepping forward in the timeline. *undoAnimName* holds the name of the reverse of *animName*; it is performed when stepping backwards. The *options* parameter accepts an object that will be used to set a variety of properties of the animation. The options are all applied in the *applyOptions()* method (see **Figure 4.2**); further explanations will come later.



```
sequenceID; // set to match the id of the parent AnimSequence
timelineID; // set to match the id of the parent AnimTimeline

// Decides if the upcoming animation should wait for this one to finish (can be changed in applyOptions())
blocksNext = true;
blocksPrev = true;
duration = 500;
playbackRate = 1;

constructor(domElem, animName, options) {
  this.id = AnimBlock.id++;

  this.domElem = domElem;
  this.animName = animName;
  this.undoAnimName = `undo--${this.animName}`;

  this.applyOptions(options);
}
```

Figure 4.1: *The constructor for the AnimBlock class.*

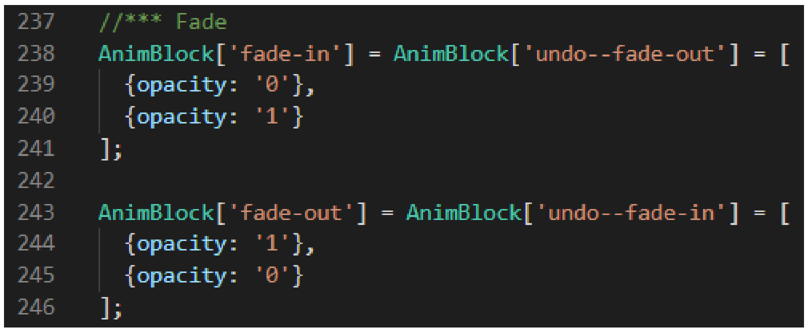


```
applyOptions(options) {
  if (!options) { return; }

  const {
    blocksNext,
    blocksPrev,
    duration,
    playbackRate,
    translateOptions,
  } = options;

  this.blocksNext = blocksNext ?? this.blocksNext;
  this.blocksPrev = blocksPrev ?? this.blocksPrev;
  this.duration = duration ?? this.duration;
  this.playbackRate = playbackRate ?? this.playbackRate;

  if (translateOptions) {
    this.applyTranslateOptions(translateOptions);
  }
}
```

Figure 4.2: *AnimBlock applyOptions() method*

4.1.1 Naming animations effectively

AnimBlock comes with several preset animations, and the first major decision was how to treat animation names. Take the two preset animations named "fade-in" and "fade-out", for example. *fade-in* un-hides an element and increases its opacity from 0 to 100, while *fade-out* decreases the opacity from 100 to 0 and then hides the element from the document. Naturally, these two animations can be called reversions of each other. Thus, if stepping forward in a timeline invokes *fade-in*, then stepping backward should invoke *fade-out*; the converse is also true. This seems straightforward, but we immediately run into a problem when generalizing this relationship. Humans understand what the reverse of either operation should be, but in code, how do we convey that going backwards from *fade-out* should invoke *fade-in* or that going backwards from *enter-wipe-from-left* should invoke *exit-wipe-to-left* and vice versa? We clearly cannot rely on the names alone; we need to find a consistent method of mapping animations to their reverse operations, which we did by creating aliases for every preset animation (demonstrated in **Figure 4.3**).[1] For example, we can map *undo–fade-out* to *fade-in* and map *undo–fade-in* to *fade-out*. The prefix "undo–" (two hyphens at the end) is unique since we do not use it as part of any preset animation name.



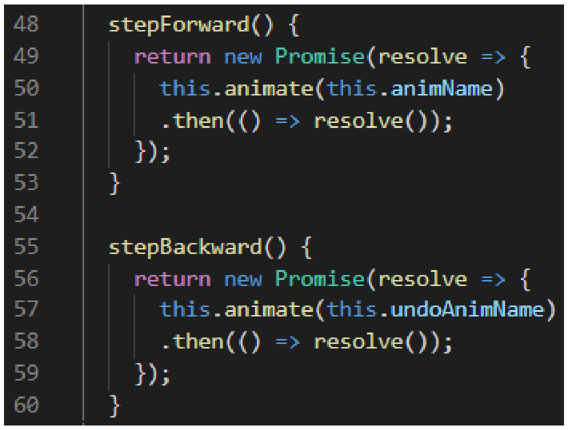
```
//*** Fade
AnimBlock['fade-in'] = AnimBlock['undo--fade-out'] = [
  {opacity: '0'},
  {opacity: '1'}
];

AnimBlock['fade-out'] = AnimBlock['undo--fade-in'] = [
  {opacity: '1'},
  {opacity: '0'}
];
```

Figure 4.3: *The presets for fading animations. Presets are formed by assigning arrays of keyframes to static properties of AnimBlock.*

As a result, every time we step backward, we simply append "undo–" to the beginning of the initial animation name. This allows us to assume that passing in a valid preset name to the AnimBlock constructor will generate the valid reverse operation name (see **Figure 4.1**, line 38). Thus, when rewinding an animBlock, we just animate with that reverse name (see **Figure 4.4**, line 57). This approach also provides a level of abstraction by only requiring a developer to specify the forward animation name when defining an animBlock. For example, if they want to perform *fade-in* and then *fade-out*, they need only define two blocks with those given names. Then, when

[1]Note that because the aliases are just references to the same memory locations, and because we are using static properties (of which there is only one on the underlying prototype), the additional memory usage is negligible.

stepping backwards, "undo–" will automatically be appended to form *undo–fade-in* and *undo–fade-out*, which are equivalent to *fade-out* and *fade-in* respectively, without the developer ever having to think about it.

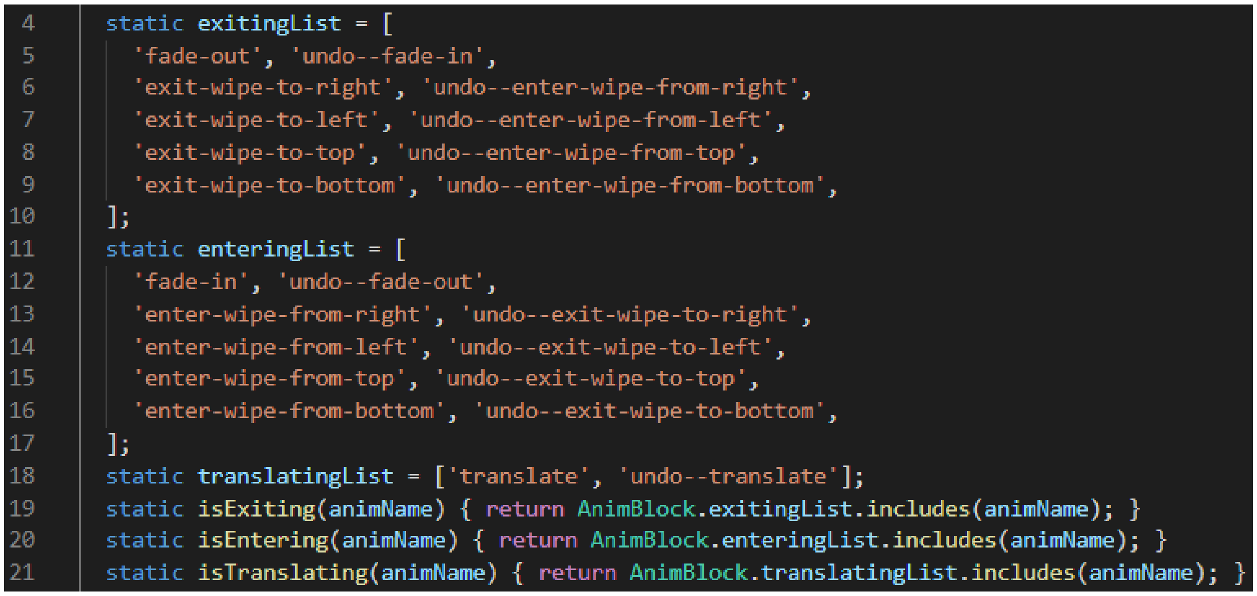


```
stepForward() {
  return new Promise(resolve => {
    this.animate(this.animName)
    .then(() => resolve());
  });
}

stepBackward() {
  return new Promise(resolve => {
    this.animate(this.undoAnimName)
    .then(() => resolve());
  });
}
```

Figure 4.4: *AnimBlock step methods.*

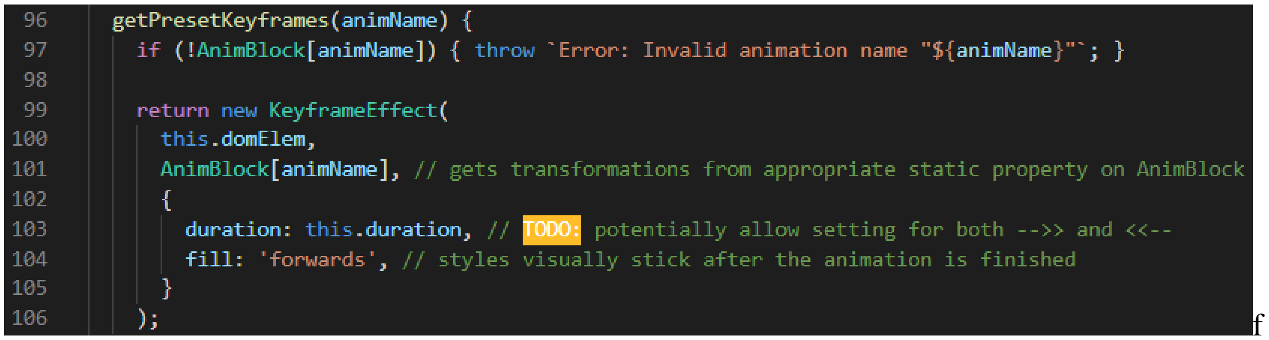
4.1.2 Performing an animation

The next major challenge was deciding how to effectively perform the animations and store any relevant data. The animations in AnimTimeline utilize the Web Animations API (WAAPI), whose specifications are detailed on the MDN Web Docs site [9]. The API provides extensive JavaScript functionality for animating DOM elements, filling the "gap between declarative CSS animations and transitions, and dynamic JavaScript animations". With the basic AnimBlock, there are four main classifications for animations: Entering, Exiting, Translation, or none of those. This grouping is important for defining what additional actions to perform before, during, and after an animation is performed. The easiest way to keep track of the animation names' corresponding types is with static arrays, as shown in **Figure 4.5**. With an exception that will be discussed later in **subsection 4.2.2**, all animation names that fall under Entering, Exiting, or none are defined by premade keyframes. We saw an example of the syntax in **Figure 4.3**, which shows how the fading animation keyframes are defined. Each array of object literals is, of course, a list of keyframes, which can be passed into the constructor of a KeyframeEffect object (part of the WAAPI); a resulting KeyframeEffect is then used to animate the transitions. **Figure 4.6** shows the AnimBlock method *getPresetKeyframes()*, which generates a KeyframeEffect object when given a valid preset animation name. Thanks to how we set up the naming system (as explained in **subsection 4.1.1**), we can simply pass in a name and access the corresponding property belonging to the AnimBlock prototype as shown in line 101. This ease is also partly due to the fact that we used the computed member access operator (i.e., []) to add static properties to the AnimBlock instead of the member access operator (i.e., .)—it allows us to use actual strings as property names, which are not limited by the constraints of JavaScript variable naming rules. Besides the DOM element on which we want to perform the animation (line 100) and the array of keyframes, the KeyframeEffect constructor also accepts an object literal with additional options (lines 102—105). In line 104, we set the *fill* property to *'forwards'*, which, in traditional CSS animations, makes the new visual state of the element stick after the animation is completed.



```
static exitingList = [
  'fade-out', 'undo--fade-in',
  'exit-wipe-to-right', 'undo--enter-wipe-from-right',
  'exit-wipe-to-left', 'undo--enter-wipe-from-left',
  'exit-wipe-to-top', 'undo--enter-wipe-from-top',
  'exit-wipe-to-bottom', 'undo--enter-wipe-from-bottom',
];
static enteringList = [
  'fade-in', 'undo--fade-out',
  'enter-wipe-from-right', 'undo--exit-wipe-to-right',
  'enter-wipe-from-left', 'undo--exit-wipe-to-left',
  'enter-wipe-from-top', 'undo--exit-wipe-to-top',
  'enter-wipe-from-bottom', 'undo--exit-wipe-to-bottom',
];
static translatingList = ['translate', 'undo--translate'];
static isExiting(animName) { return AnimBlock.exitingList.includes(animName); }
static isEntering(animName) { return AnimBlock.enteringList.includes(animName); }
static isTranslating(animName) { return AnimBlock.translatingList.includes(animName); }
```

Figure 4.5: *AnimBlock main animation classifications.*



```
getPresetKeyframes(animName) {
  if (!AnimBlock[animName]) { throw `Error: Invalid animation name "${animName}"`; }

  return new KeyframeEffect(
    this.domElem,
    AnimBlock[animName], // gets transformations from appropriate static property on AnimBlock
    {
      duration: this.duration, // TODO: potentially allow setting for both -->> and <<--
      fill: 'forwards', // styles visually stick after the animation is finished
    }
  );
```

f

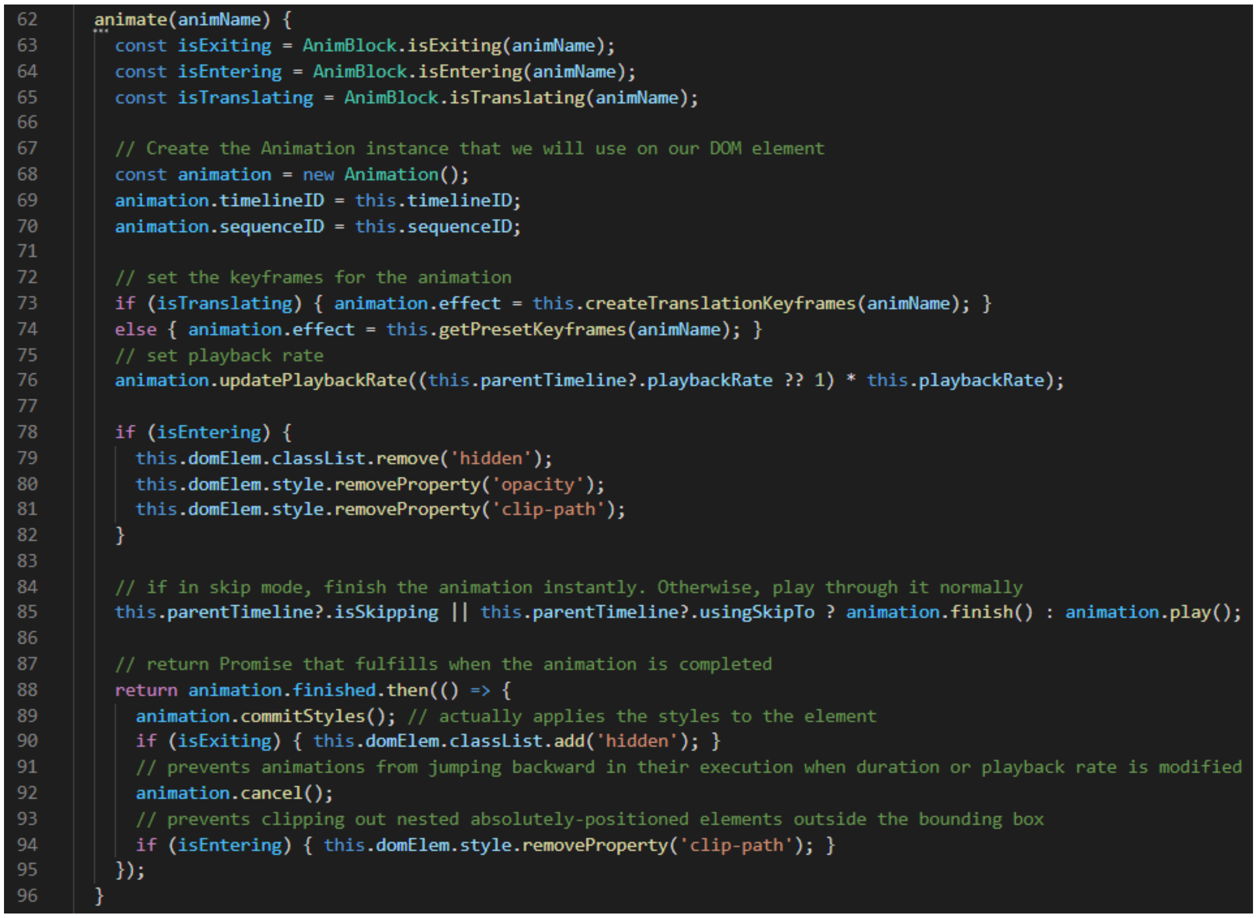
Figure 4.6: *AnimBlock getPresetKeyframes().*

4.1.3 Post-animation operations

As we will later see, the AnimTimeline framework relies heavily on Promises[2] because they facilitate working with asynchronous operations and allow us to "await" results; this is necessary for effectively coordinating the sequences of animations (whether in serial or in parallel). **Figure 4.7** shows the full AnimBlock *animate()* method, which is called every time a block is played. Let us examine lines 88—95, which are executed after an animation is completed. In the WAAPI, the Animation *commitStyles()* method forces the resulting style transitions on an element to stick; this is accomplished by adding inline styles to the HTML code. Thus, since the fill mode is set to *forwards*, we guarantee that once an animation is completed, the element will retain its changes. Line 92's purpose is far less intuitive. An oddity with the way animations are implemented in CSS/JavaScript is that keyframes seem to be tied to an element even after *commitStyles()* is called. If the playback speed on an animation is increased in the middle of an animation, one would expect that it be over and done with upon completion, but this is not necessarily the case; if the playback speed is then reverted, the animation may actually jump backward in its execution and then continue as if retroactively deciding that the speedup never happened at all. This is obviously problematic considering the fact that one of the primary features of any playback system is the ability to fast-forward at any time. This can be entirely circumvented by calling the *cancel()* method, which definitively cuts all ties with the active keyframes. Thus, once an animation is completed,

[2]*Promise* is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value citeMDNdoc-promise.

the styles are committed, and the animation is canceled to stop it from being active in any possible capacity.



```javascript
animate(animName) {
  const isExiting = AnimBlock.isExiting(animName);
  const isEntering = AnimBlock.isEntering(animName);
  const isTranslating = AnimBlock.isTranslating(animName);

  // Create the Animation instance that we will use on our DOM element
  const animation = new Animation();
  animation.timelineID = this.timelineID;
  animation.sequenceID = this.sequenceID;

  // set the keyframes for the animation
  if (isTranslating) { animation.effect = this.createTranslationKeyframes(animName); }
  else { animation.effect = this.getPresetKeyframes(animName); }
  // set playback rate
  animation.updatePlaybackRate((this.parentTimeline?.playbackRate ?? 1) * this.playbackRate);

  if (isEntering) {
    this.domElem.classList.remove('hidden');
    this.domElem.style.removeProperty('opacity');
    this.domElem.style.removeProperty('clip-path');
  }

  // if in skip mode, finish the animation instantly. Otherwise, play through it normally
  this.parentTimeline?.isSkipping || this.parentTimeline?.usingSkipTo ? animation.finish() : animation.play();

  // return Promise that fulfills when the animation is completed
  return animation.finished.then(() => {
    animation.commitStyles(); // actually applies the styles to the element
    if (isExiting) { this.domElem.classList.add('hidden'); }
    // prevents animations from jumping backward in their execution when duration or playback rate is modified
    animation.cancel();
    // prevents clipping out nested absolutely-positioned elements outside the bounding box
    if (isEntering) { this.domElem.style.removeProperty('clip-path'); }
  });
}
```

Figure 4.7: *AnimBlock animate() method.*

Notice that the post-animation actions are performed within a Promise. In the WAAPI, *Animation.finished* provides a Promise that resolves when the Animation object in question finishes animating. It is *imperative* that we use this Promise and *not* the *onfinish* onevent property. The latter attaches an event listener that listens for the *finished* event, but the issue is that event handlers utilize normal callback functions. This is incredibly problematic because callback functions used with event listeners go in what is called the "callback queue", while callbacks related to Promises—such callbacks being termed "microtasks"—go into the "microtasks queue". Callbacks have a lower priority than microtasks, which means that even if the animation for an element finishes, the post-animation operations we need to perform would execute *after* any other Promise operations are finished. This would cause undesired results when AnimBlocks are performed on the same element within one sequence of animations—additional modifications are stacked on top of an element before the *previous* AnimBlocks finish their jobs.

4.1.4 Animation classification considerations

As mentioned in **subsection 4.1.2**, we categorize the animations into Exiting, Entering, Translating, or none. In the *animate()* method (**Figure 4.7**), it becomes clear why this is necessary.

4.1.4.1 Exiting

Handling an element that is exiting is straightforward—simply hide the element in some way after the animation styles have been committed. In **Figure 4.7** line 90, this is done by adding the class *hidden* to the element. In a separate CSS stylesheet, this is defined with the style rule "*display: none!important*", which makes the element inaccessible (for certain elements, the *hidden* class instead has "*visibility: hidden*". This will be addressed later in **section 5.2**).

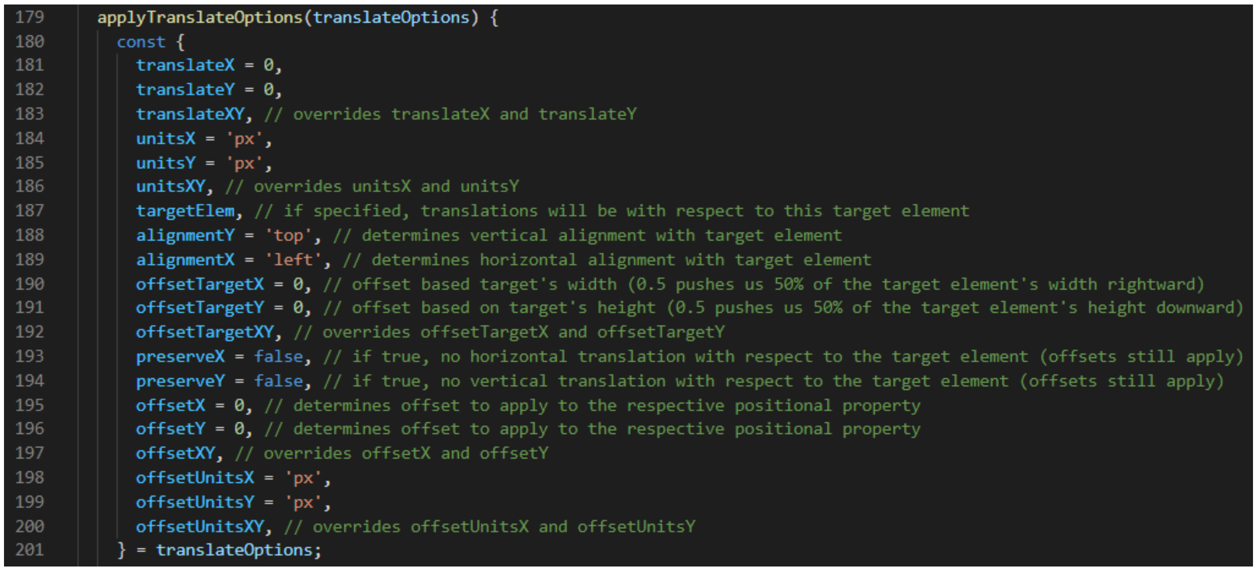
4.1.4.2 Entering

Examine lines 78—82 of the *animate()* method in **Figure 4.7**. Before the actual animation begins on line 85, these lines are executed if the animation name indicates that the DOM element will be entering. If an animation name is associated with a DOM element entering (e.g., *fade-in*), then we need to ensure that it is not invisible to the document—that is, its CSS *display* property is not set to *none* (or, alternatively, that the *visibility* property is not set to *hidden*)—when the animation starts. For example, if we want to fade an element into view, we need to make sure that it is not hidden so that we actually see it transition from 0 opacity to 100 opacity. The first, obvious step is to remove the *hidden* class that the element presumably currently has. The second step is less apparent. One potential problem with performing animations and eventually calling *commitStyles()* is that this does not just magically change the visual style of the element; it actually just adds the changes as inline styles in the HTML code. Suppose we performed the exiting animation *fade-out* on some element. Upon inspecting the HTML using the browser's developer tools, we will see "opacity: 0" added to the *style* attribute. This is an issue when we want to perform an entering animation that modifies a CSS property other than opacity. For example, our preset animation *enter-wipe-from-left* modifies the CSS *clip-path* property to create a linear wiping effect to reveal the element. The opacity will still be set to 0, so the element will never actually be visible. Of course, the same is true if we use an exiting wipe effect followed by an entering fade effect—the clipping path on the element would render the element invisible even though opacity was increased to 1. Generally speaking, performing exiting and entering animations that modify different CSS properties can cause unexpected graphical bugs. This can be solved by manually removing inline styles as we do Lines 80 and 81. In this way, regardless of an element's state prior to entering, no conflicting properties will prevent the element from being visible.

In regards to using the *clip-path* property to simulate wiping effects for entrances, we were careful not to overlook the relationship between the bounding box of an element and any nested elements with absolute positioning. One of our preset wiping entrance animations is *enter-wipe-from-left*, which essentially transitions a clipping mask over the element that starts on the left edge and expands to the right edge. This naturally makes use of the element's width to form the size of the clipping path, but absolutely positioned elements are, by definition, outside of the normal flow of the document—they do not contribute to the width of the parent element. Thus, the final clipping path encompassing the parent element ends up cutting out any absolutely positioned child elements that may lie outside of the element (text boxes are a perfect example of such children). To prevent this from being an issue, any inline styling for *clip-path* is removed after the animation is completed if the animation was an Entering type (see **Figure 4.7**, line 94).

4.1.4.3 Translating

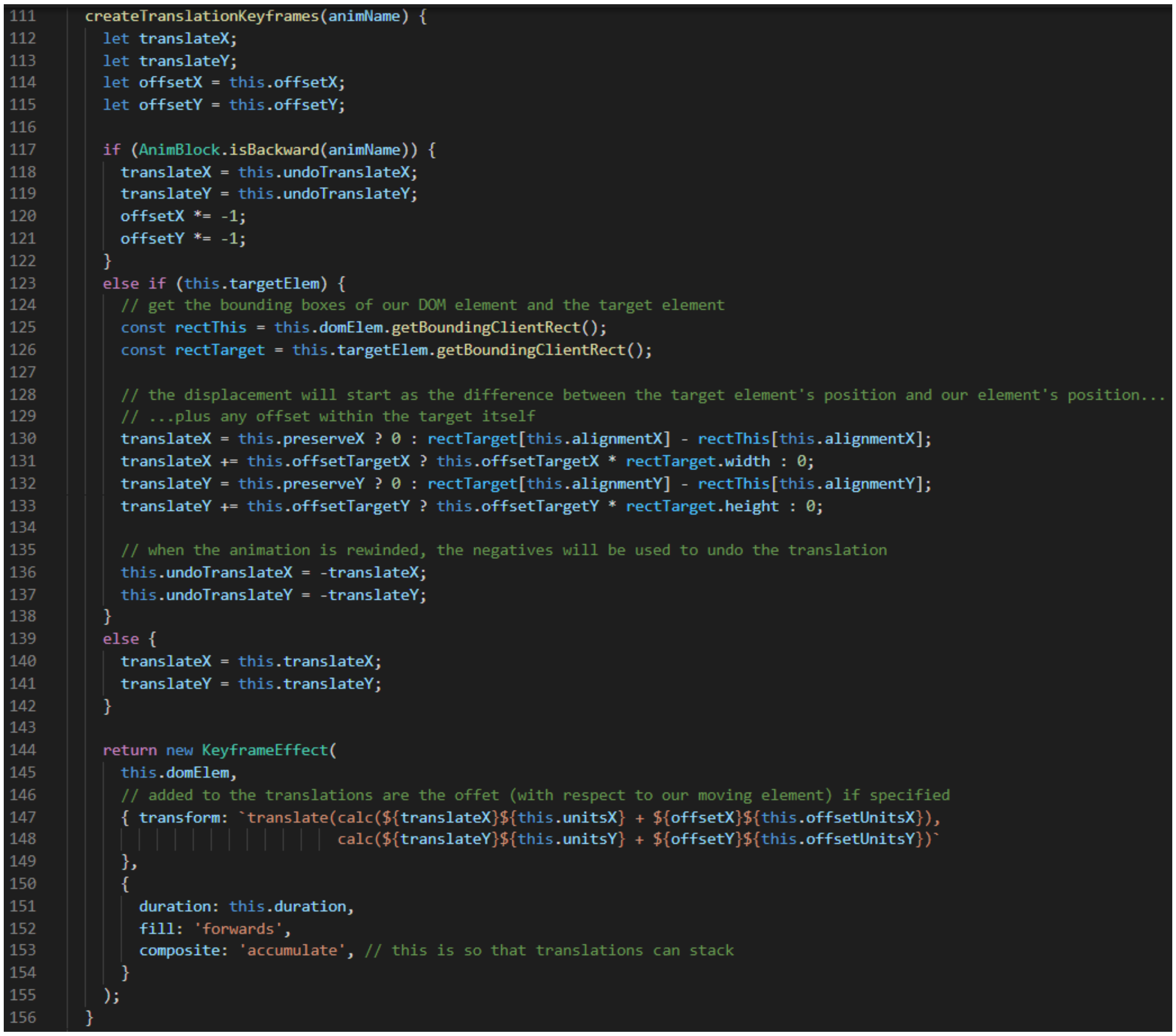
This type of animation is the most complicated. It does not have its own preset animation; rather, as shown in **Figure 4.7**, line 73, we generate the necessary KeyframeEffect using the method *createTranslationKeyframes()* instead of *getPresetKeyframes()*. In order to indicate that a translation should be performed, the developer must pass in *"translate"* as the animation name when constructing the AnimBlock instance. The developer can then use the constructor's *options* argument (see **Figure 4.1**) to specify several aspects of the translation itself. **Figure 4.8** shows all of the possible properties we allow one to set (for brevity, the logic for applying the values using the *this* keyword is not shown). The comments in the figure explain each option; the important thing to understand is that this extensive list of options—from the distance an object should move to even the units that should be used in the calculations—allows developers to generate movement animations that range in complexity from very simple to quite specific.



```
applyTranslateOptions(translateOptions) {
  const {
    translateX = 0,
    translateY = 0,
    translateXY, // overrides translateX and translateY
    unitsX = 'px',
    unitsY = 'px',
    unitsXY, // overrides unitsX and unitsY
    targetElem, // if specified, translations will be with respect to this target element
    alignmentY = 'top', // determines vertical alignment with target element
    alignmentX = 'left', // determines horizontal alignment with target element
    offsetTargetX = 0, // offset based target's width (0.5 pushes us 50% of the target element's width rightward)
    offsetTargetY = 0, // offset based on target's height (0.5 pushes us 50% of the target element's height downward)
    offsetTargetXY, // overrides offsetTargetX and offsetTargetY
    preserveX = false, // if true, no horizontal translation with respect to the target element (offsets still apply)
    preserveY = false, // if true, no vertical translation with respect to the target element (offsets still apply)
    offsetX = 0, // determines offset to apply to the respective positional property
    offsetY = 0, // determines offset to apply to the respective positional property
    offsetXY, // overrides offsetX and offsetY
    offsetUnitsX = 'px',
    offsetUnitsY = 'px',
    offsetUnitsXY, // overrides offsetUnitsX and offsetUnitsY
  } = translateOptions;
```

Figure 4.8: *AnimBlock applyTranslateOptions() method.*

The logic for creating keyframes for a translation is executed in lines 144—155 of the *createTranslationKeyframes()* method, which is displayed in **Figure 4.9**. Ultimately, any translation is reduced to traveling some distance in the X and Y directions plus some offsets, all of which have units attached.



```
111    createTranslationKeyframes(animName) {
112      let translateX;
113      let translateY;
114      let offsetX = this.offsetX;
115      let offsetY = this.offsetY;
116
117      if (AnimBlock.isBackward(animName)) {
118        translateX = this.undoTranslateX;
119        translateY = this.undoTranslateY;
120        offsetX *= -1;
121        offsetY *= -1;
122      }
123      else if (this.targetElem) {
124        // get the bounding boxes of our DOM element and the target element
125        const rectThis = this.domElem.getBoundingClientRect();
126        const rectTarget = this.targetElem.getBoundingClientRect();
127
128        // the displacement will start as the difference between the target element's position and our element's position...
129        // ...plus any offset within the target itself
130        translateX = this.preserveX ? 0 : rectTarget[this.alignmentX] - rectThis[this.alignmentX];
131        translateX += this.offsetTargetX ? this.offsetTargetX * rectTarget.width : 0;
132        translateY = this.preserveY ? 0 : rectTarget[this.alignmentY] - rectThis[this.alignmentY];
133        translateY += this.offsetTargetY ? this.offsetTargetY * rectTarget.height : 0;
134
135        // when the animation is rewinded, the negatives will be used to undo the translation
136        this.undoTranslateX = -translateX;
137        this.undoTranslateY = -translateY;
138      }
139      else {
140        translateX = this.translateX;
141        translateY = this.translateY;
142      }
143
144      return new KeyframeEffect(
145        this.domElem,
146        // added to the translations are the offet (with respect to our moving element) if specified
147        { transform: `translate(calc(${translateX}${this.unitsX} + ${offsetX}${this.offsetUnitsX}),
148                               calc(${translateY}${this.unitsY} + ${offsetY}${this.offsetUnitsY})`
149        },
150        {
151          duration: this.duration,
152          fill: 'forwards',
153          composite: 'accumulate', // this is so that translations can stack
154        }
155      );
156    }
```

Figure 4.9: *AnimBlock createTranslationKeyframes() method.*

Notice that in addition to specifying the fill mode (as we discussed in **subsection 4.1.2**), we also specified *composite: 'accumulate'* in line 153. The KeyframeEffect *composite* property is, at the time of this writing, an experiment technology that "resolves how an element's animation impacts its underlying property values", according to the MDN Web Docs site. What this means does not particularly matter; just know that *accumulate* means that every change to a given style property adds on to the previous changes. Here, with handling movements, this makes it far easier to manage the positional states of elements after several translations—moving using relative directions (e.g., moving 5 rem in the X direction) is possible without having to worry about positions resetting, and stepping backward through each translation AnimBlock entails merely subtracting previous movements to zero out the displacements.

When stepping forward with a translation, there are two modes of movement: The element is either moving relative to its current position (lines 139—142) or moving to another target element (lines 123—138). The first case is simple—the local variables *translationX* and *translationY* will be set to the translation values that the developer set up during the animBlock's creation. In the second case, we can actually move our element to another target element; this is done by subtracting the distances between the edges of both elements' bounding rectangles (which are DOMRect instances). We can control further details of the movement, such as the horizontal and vertical alignment of the element with respect to the target element (which just uses the *top* or *bottom* and *left* or *right* properties of the bounding rectangle objects in the subtractions), the offset of our element within the target element (for example, an offset of *-0.5* for *offsetTargetX* will subtract 50% of the target element's width from our element's x translation), and whether we want to preserve one of the axis positions (perhaps we only want our element to move vertically to line up with the target element but not travel the horizontal distance). If the element is moving backward, all we need to do is subtract the distance moved during the forward translation to, as we said before, "zero out the displacement." This is done by simply storing the negation of *translateX* and *translateY*, negating any offset values, and using these values while creating the keyframes.

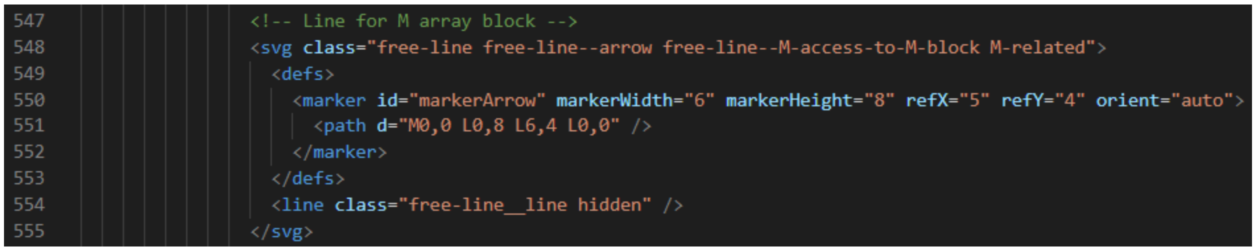
Whether the element is moving with respect to itself or another element or rewinding its translation, the final translation is created in lines 144—155, taking into account units and offsets specified.[3]

4.2 AnimBlockLine

A useful feature to have in a visualization is the ability to draw lines between any two elements on the page; better yet would be the ability to have the endpoints of said line update whenever the two elements change their positions. This is realized with a special class called AnimBlock-

[3]If moving to another target element, *unitsX* and *unitsY* are strictly set to *'px'* regardless of what the developer tries to set them to. This is because the positional properties in DOMRect are always in pixel units, so using any other unit while creating the keyframes would be incorrect.

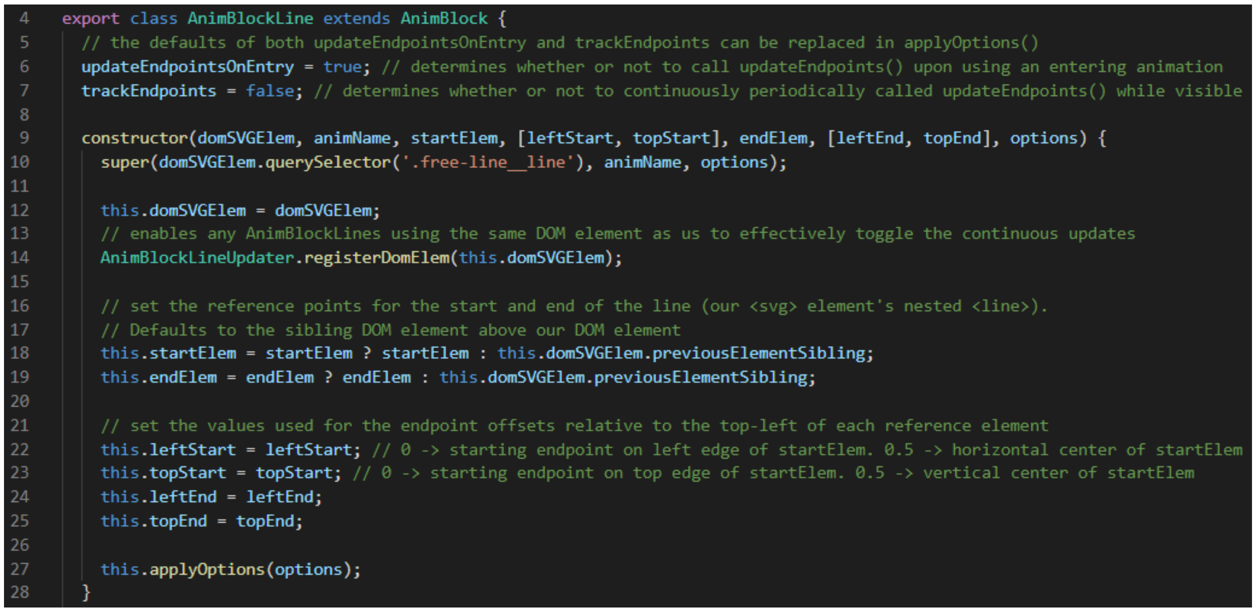
Line, which is a subclass of AnimBlock. The only HTML element to which AnimBlockLine is applicable is an <svg> element containing a <line>. Optionally, there can be a <marker>, which adds a triangular endpoint to the line. An example is shown in **Figure 4.10**; all such structures will be called "free-line"s because they are free to point to any elements on the page.



```
<!-- Line for M array block -->
<svg class="free-line free-line--arrow free-line--M-access-to-M-block M-related">
  <defs>
    <marker id="markerArrow" markerWidth="6" markerHeight="8" refX="5" refY="4" orient="auto">
      <path d="M0,0 L0,8 L6,4 L0,0" />
    </marker>
  </defs>
  <line class="free-line__line hidden" />
</svg>
```

Figure 4.10: *Example of a free-line in the HTML*

The AnimBlockLine constructor and some data members are shown in **Figure 4.11**. The *domSVGElem* parameter is the <svg> element that we want to animate. *startElem* and *endElem* reference two other DOM elements, which will serve as the start and end points of the free-line. As explained in the comments in **Figure 4.11**, *leftStart*, *topStart*, *leftEnd*, and *topEnd* can be used to offset the position of the free-line's endpoints relative to the target elements. The call to *register-DomElem()* will be explained later in this section; for now, know that it links all AnimBlockLine instances that share a common <svg> element to a single updater. The process of pointing an SVG line between two elements on the page and handling updating the endpoints is somewhat complex, but we will briefly go over how we decided to implement it for the interested reader.

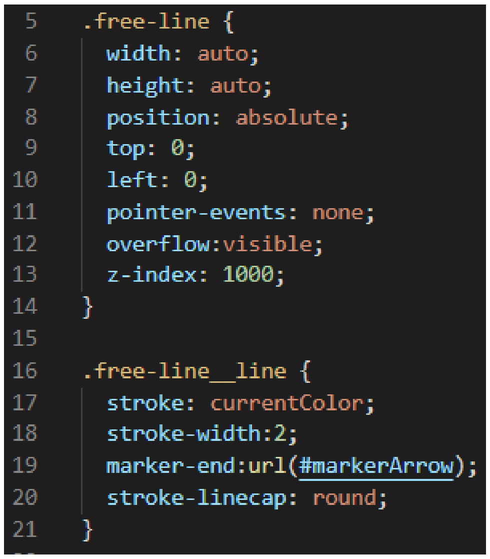


```
export class AnimBlockLine extends AnimBlock {
  // the defaults of both updateEndpointsOnEntry and trackEndpoints can be replaced in applyOptions()
  updateEndpointsOnEntry = true; // determines whether or not to call updateEndpoints() upon using an entering animation
  trackEndpoints = false; // determines whether or not to continuously periodically called updateEndpoints() while visible

  constructor(domSVGElem, animName, startElem, [leftStart, topStart], endElem, [leftEnd, topEnd], options) {
    super(domSVGElem.querySelector('.free-line__line'), animName, options);

    this.domSVGElem = domSVGElem;
    // enables any AnimBlockLines using the same DOM element as us to effectively toggle the continuous updates
    AnimBlockLineUpdater.registerDomElem(this.domSVGElem);

    // set the reference points for the start and end of the line (our <svg> element's nested <line>).
    // Defaults to the sibling DOM element above our DOM element
    this.startElem = startElem ? startElem : this.domSVGElem.previousElementSibling;
    this.endElem = endElem ? endElem : this.domSVGElem.previousElementSibling;

    // set the values used for the endpoint offsets relative to the top-left of each reference element
    this.leftStart = leftStart; // 0 -> starting endpoint on left edge of startElem. 0.5 -> horizontal center of startElem
    this.topStart = topStart; // 0 -> starting endpoint on top edge of startElem. 0.5 -> vertical center of startElem
    this.leftEnd = leftEnd;
    this.topEnd = topEnd;

    this.applyOptions(options);
  }
```

Figure 4.11: *AnimBlockLine constructor*

4.2.1 Setting free-line endpoints

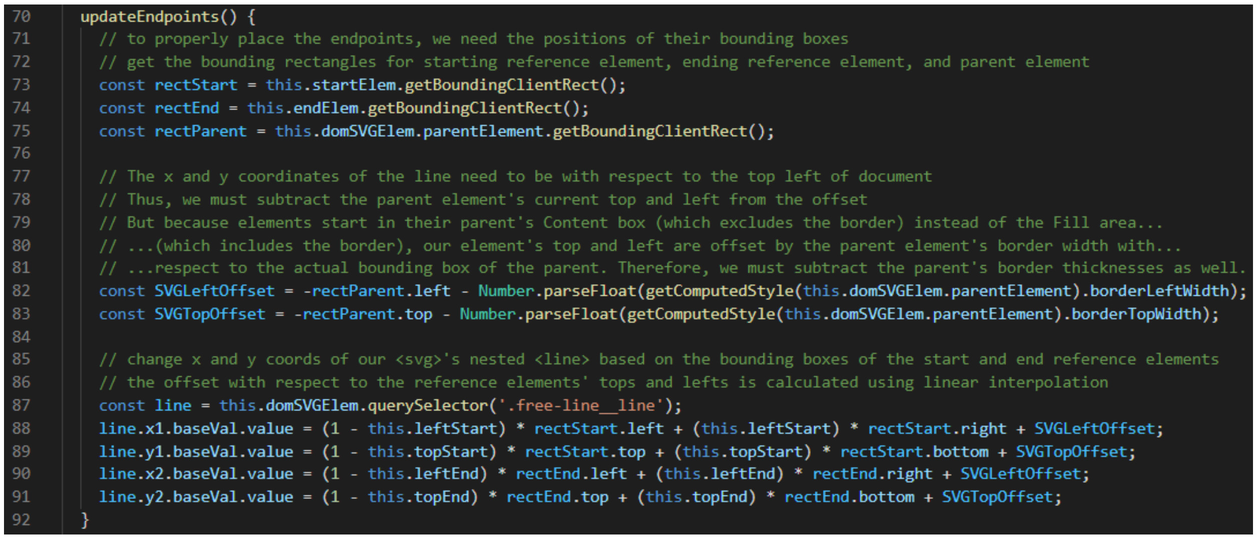
The first step in properly setting the endpoints of a free-line is to set up the CSS for the <svg> element. Let us examine the CSS code snippet in **Figure 4.12**; the <svg> element is denoted by the "free-line" class, while the inner <line> element is denoted by the "free-line__line" class. We will not discuss the various problematic iterations of the code, but setting the *height* and *width* properties to *auto* and absolutely positioning the element to the top and left ensures that the <line>'s endpoints can be properly positioned without A) letting the <svg> elements take up the entire screen size or B) necessitating placing all free-line elements within one designated area in the HTML code. Setting *pointer-events* to *none* prevents the <svg> elements from being selectable with right-click (this is because the elements get in the way when trying to utilize the browser developer tools). As for the CSS pertaining to <line>, the only important thing to note is that, because two HTML elements cannot have the same ID, the IDs and URLs for every <marker> and <line> are generated later using JavaScript (the code will not be shown here).



```css
.free-line {
  width: auto;
  height: auto;
  position: absolute;
  top: 0;
  left: 0;
  pointer-events: none;
  overflow:visible;
  z-index: 1000;
}

.free-line__line {
  stroke: currentColor;
  stroke-width:2;
  marker-end:url(#markerArrow);
  stroke-linecap: round;
}
```

Figure 4.12: *Partial CSS code snippet for a free-line*

With the CSS set up, we can examine the AnimBlockLine method *updateEndpoints()* as shown in **Figure 4.13**. Explanations are given in the comments of the code shown in the figure. Essentially, we use the differences between the bounding boxes of the <svg> element and the two target elements, the overall position on the document, and other influential values such as border widths and offsets to compute the appropriate pixel positions of the <line>'s endpoints' coordinates. This leaves the question of when the endpoints should be set, and that is partly answered by the AnimBlockLine's member variable *updateEndpointsOnEntry*. If set to *true* (which it is by default), then *updateEndpoints()* will be called whenever the AnimBlockLine performs an entering animation (recall the discussion about entering animations in **subsubsection 4.1.4.2**); we will see this when we describe the method *handleUpdateSettings()*.

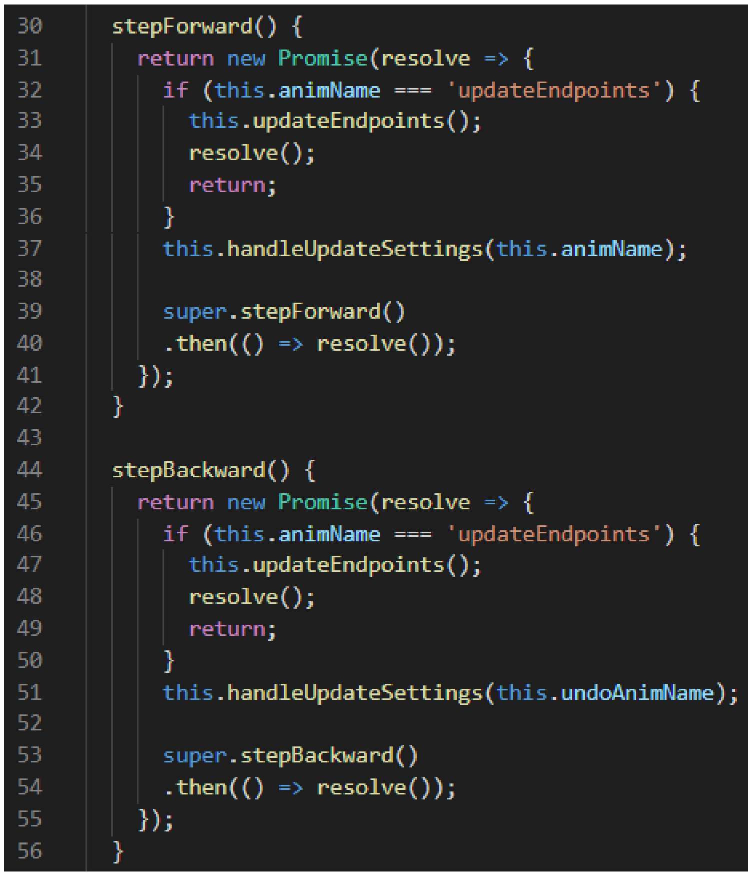


```
updateEndpoints() {
  // to properly place the endpoints, we need the positions of their bounding boxes
  // get the bounding rectangles for starting reference element, ending reference element, and parent element
  const rectStart = this.startElem.getBoundingClientRect();
  const rectEnd = this.endElem.getBoundingClientRect();
  const rectParent = this.domSVGElem.parentElement.getBoundingClientRect();

  // The x and y coordinates of the line need to be with respect to the top left of document
  // Thus, we must subtract the parent element's current top and left from the offset
  // But because elements start in their parent's Content box (which excludes the border) instead of the Fill area...
  // ...(which includes the border), our element's top and left are offset by the parent element's border width with...
  // ...respect to the actual bounding box of the parent. Therefore, we must subtract the parent's border thicknesses as well.
  const SVGLeftOffset = -rectParent.left - Number.parseFloat(getComputedStyle(this.domSVGElem.parentElement).borderLeftWidth);
  const SVGTopOffset = -rectParent.top - Number.parseFloat(getComputedStyle(this.domSVGElem.parentElement).borderTopWidth);

  // change x and y coords of our <svg>'s nested <line> based on the bounding boxes of the start and end reference elements
  // the offset with respect to the reference elements' tops and lefts is calculated using linear interpolation
  const line = this.domSVGElem.querySelector('.free-line__line');
  line.x1.baseVal.value = (1 - this.leftStart) * rectStart.left + (this.leftStart) * rectStart.right + SVGLeftOffset;
  line.y1.baseVal.value = (1 - this.topStart) * rectStart.top + (this.topStart) * rectStart.bottom + SVGTopOffset;
  line.x2.baseVal.value = (1 - this.leftEnd) * rectEnd.left + (this.leftEnd) * rectEnd.right + SVGLeftOffset;
  line.y2.baseVal.value = (1 - this.topEnd) * rectEnd.top + (this.topEnd) * rectEnd.bottom + SVGTopOffset;
}
```

Figure 4.13: *AnimBlockLine updateEndpoints()*

4.2.2 Stepping with AnimBlockLine

Stepping with AnimBlockLine is not too complicated; *stepForward()* and *stepBackward()* are shown in **Figure 4.14**. One special animation name that is exclusive to AnimBlockLine is *updateEndpoints*, which causes the endpoints of the free-line to update. We mentioned in **subsection 4.1.2** that there was an exception to the rule that non-translation animation names are defined by premade keyframes, and this is it. Presumably, the *updateEndpoints* name is used when either of the target elements has moved and we want the endpoints to follow them. Besides that, AnimBlockLine has access to the same animation options as AnimBlock because it is a subclass; it can simply use *super* to call the standard methods if anything besides *updateEndpoints* is used (e.g., *enter-wipe-from-left*, *highlight*, *fade-out*, etc.). Regardless of the animation name, stepping with AnimBlockLine will call *handleUpdateSettings()*, whose job will be explained next.



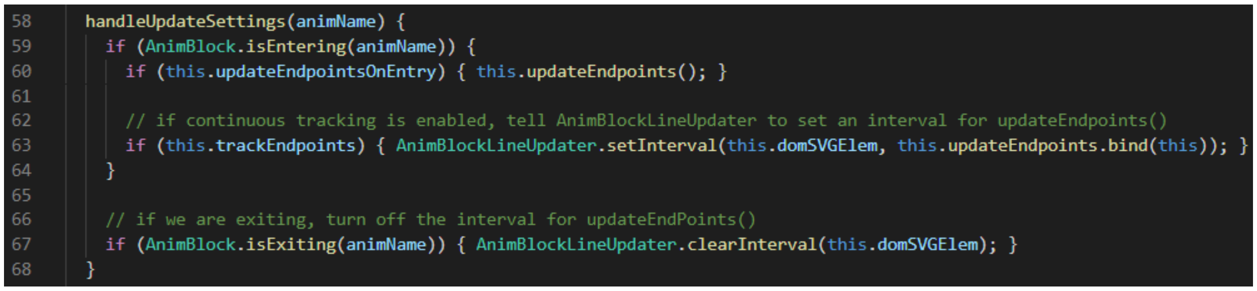
```
stepForward() {
  return new Promise(resolve => {
    if (this.animName === 'updateEndpoints') {
      this.updateEndpoints();
      resolve();
      return;
    }
    this.handleUpdateSettings(this.animName);

    super.stepForward()
    .then(() => resolve());
  });
}

stepBackward() {
  return new Promise(resolve => {
    if (this.animName === 'updateEndpoints') {
      this.updateEndpoints();
      resolve();
      return;
    }
    this.handleUpdateSettings(this.undoAnimName);

    super.stepBackward()
    .then(() => resolve());
  });
}
```

Figure 4.14: *AnimBlockLine stepping methods*

4.2.3 Continuously updating endpoints

While it is useful to be able to update the endpoints of a free-line while stepping through the timeline, this does not account for the case where the target elements' positions depend on factors outside of the AnimTimeline, e.g., scrolling. As we will see when we go over the WIS visualization in **section 5.2**, the left side of the screen displaying the time graph is fixed, while the right side of the screen displaying the job cards can be vertically scrolled when the tree becomes tall enough. If there are lines pointing between both halves of the screen (which there are), we need to make sure they maintain their endpoints' targets even as positions of the targets change with respect to each other.

One of AnimBlocks' data members, shown in **Figure 4.11**, is *trackEndpoints*, which is *false* by default. If it is set to *true*, then the free-line will update its endpoints constantly. This is handled in *handleUpdateSettings()*, shown in **Figure 4.15**.

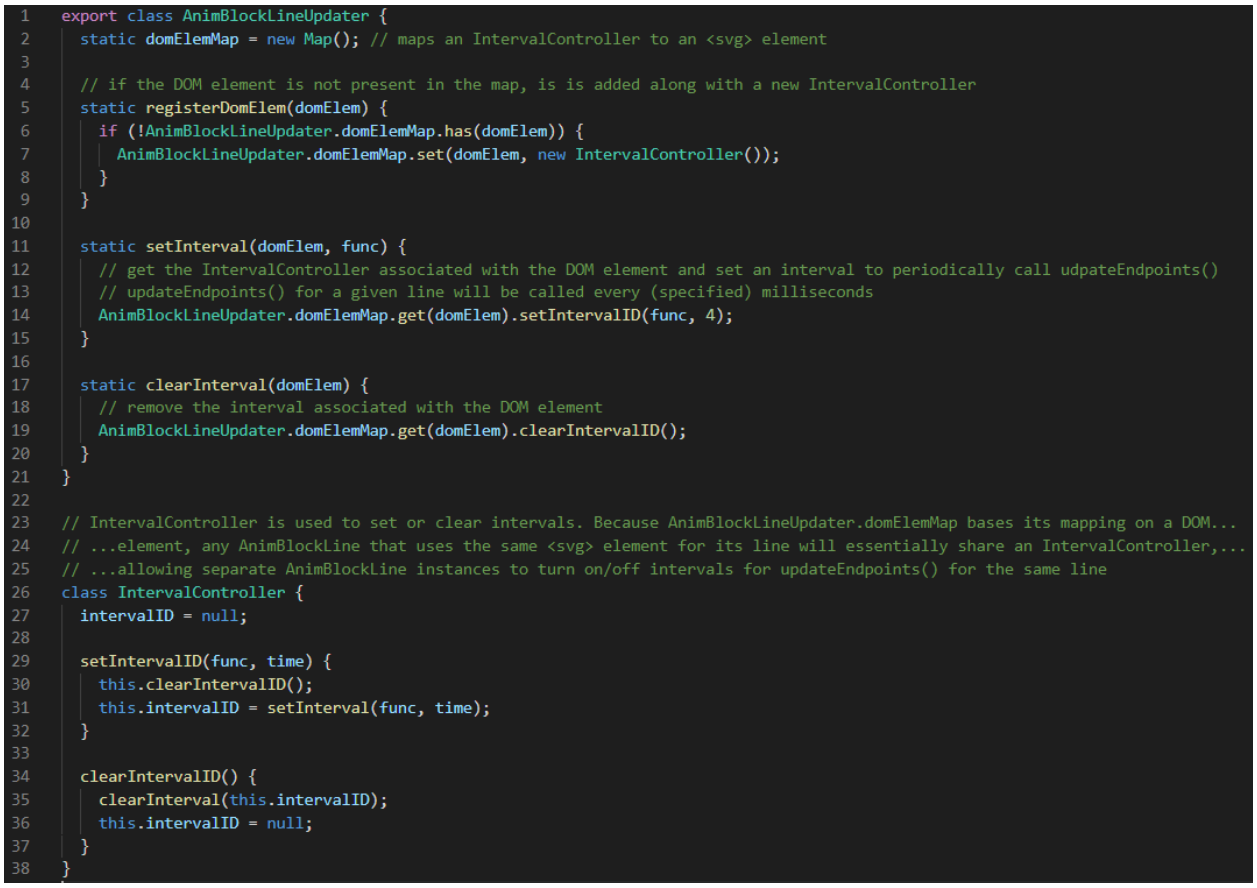


```
58  handleUpdateSettings(animName) {
59    if (AnimBlock.isEntering(animName)) {
60      if (this.updateEndpointsOnEntry) { this.updateEndpoints(); }
61  
62      // if continuous tracking is enabled, tell AnimBlockLineUpdater to set an interval for updateEndpoints()
63      if (this.trackEndpoints) { AnimBlockLineUpdater.setInterval(this.domSVGElem, this.updateEndpoints.bind(this)); }
64    }
65  
66    // if we are exiting, turn off the interval for updateEndPoints()
67    if (AnimBlock.isExiting(animName)) { AnimBlockLineUpdater.clearInterval(this.domSVGElem); }
68  }
```

Figure 4.15: *AnimBlockLine handleUpdateSettings()*

In this method, the first task is to check whether or not the free-line is performing an entering animation (discussed in **subsubsection 4.1.4.2**). If so, then—as we hinted at the end of **subsection 4.2.1**—if the member variable *updateEndPointsOnEntry* is *true*, *updateEndpoints()* will be called in line 60. After this we check if *trackEndpoints* is true in line 63. If so, then we set an interval that continuously calls *updateEndpoints()* using a class called AnimBlockLineUpdater.

What AnimBlockLineUpdater is will be explained shortly, but first, we will consider a scenario that highlights its necessity. One free-line will most likely be the subject of multiple different AnimBlockLine instances; at the very least, it will likely enter and later exit (which constitutes two animBlockLines). Every instance is independent, which means that they cannot coordinate their usage of *updateEndpoints()*. This means that it will not suffice to simply use standard JavaScript functions *setInterval()* and *clearInterval()* to respectively continuously call *updateEndpoints()* and stop continuously calling it. This is because each animBlockLine only has access to its own *updateEndPoints()* method. If we, later in a timeline, want to stop a line from continuously updating, we need to find a way to access that other animBlockLine's interval data in order to clear the interval. In general, every AnimBlockLine that shares the same <svg> element needs to be linked such that they can stop each other's continuous updates if necessary. This is where *AnimBlockLineUpdater* comes into play. The entirety of AnimBlockLineUpdater.js is shown in **Figure 4.16**



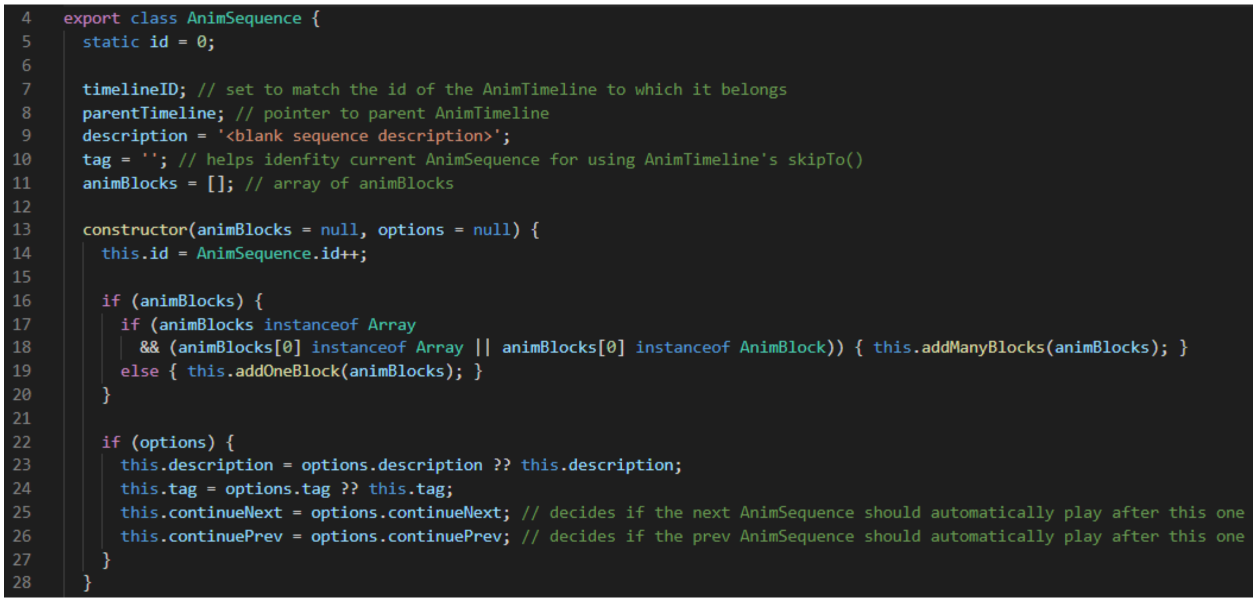
```
1   export class AnimBlockLineUpdater {
2     static domElemMap = new Map(); // maps an IntervalController to an <svg> element
3   
4     // if the DOM element is not present in the map, is is added along with a new IntervalController
5     static registerDomElem(domElem) {
6       if (!AnimBlockLineUpdater.domElemMap.has(domElem)) {
7         AnimBlockLineUpdater.domElemMap.set(domElem, new IntervalController());
8       }
9     }
10  
11    static setInterval(domElem, func) {
12      // get the IntervalController associated with the DOM element and set an interval to periodically call udpateEndpoints()
13      // updateEndpoints() for a given line will be called every (specified) milliseconds
14      AnimBlockLineUpdater.domElemMap.get(domElem).setIntervalID(func, 4);
15    }
16  
17    static clearInterval(domElem) {
18      // remove the interval associated with the DOM element
19      AnimBlockLineUpdater.domElemMap.get(domElem).clearIntervalID();
20    }
21  }
22  
23  // IntervalController is used to set or clear intervals. Because AnimBlockLineUpdater.domElemMap bases its mapping on a DOM...
24  // ...element, any AnimBlockLine that uses the same <svg> element for its line will essentially share an IntervalController,...
25  // ...allowing separate AnimBlockLine instances to turn on/off intervals for updateEndpoints() for the same line
26  class IntervalController {
27    intervalID = null;
28  
29    setIntervalID(func, time) {
30      this.clearIntervalID();
31      this.intervalID = setInterval(func, time);
32    }
33  
34    clearIntervalID() {
35      clearInterval(this.intervalID);
36      this.intervalID = null;
37    }
38  }
```

Figure 4.16: *AnimBlockLineUpdater*

The comments in the code explain the implementation, but we will briefly go over it here. First, let us look at *IntervalController*, whose definition begins on line 27. Its only job is is to set and clear an interval, but one IntervalController instance will be mapped exclusively to one free-line <svg>, effectively mapping it to every AnimBlockLine using that <svg>. This effect starts with creating *domElemMap* as a Map instance in line 2; it is used to map one IntervalController to one free-line in the method *registerDomElem()* (which is called in line 14 of the AnimBlockLine constructor shown in **Figure 4.11**). The AnimBlockLineUpdater method *setInterval()*, which is the method we saw in line 63 of *handleUpdateSettings()* in **Figure 4.15**, sets an interval using the IntervalController associated with the <svg> element. The *func* parameter is always the *updateEndpoints()* method of the animBlockLine being active at the time. Because of this, another animBlockLine sharing the same <svg> element can clear the same interval using the last method *clearInterval()* because they will ultimately utilize the same IntervalController. Thus, in line 67 of *handleUpdateSettings()* (**Figure 4.15**), if the free-line is exiting, we can clear any potential interval associated with the <svg> to prevent unnecessary updates to an invisible element.

4.3 AnimSequence

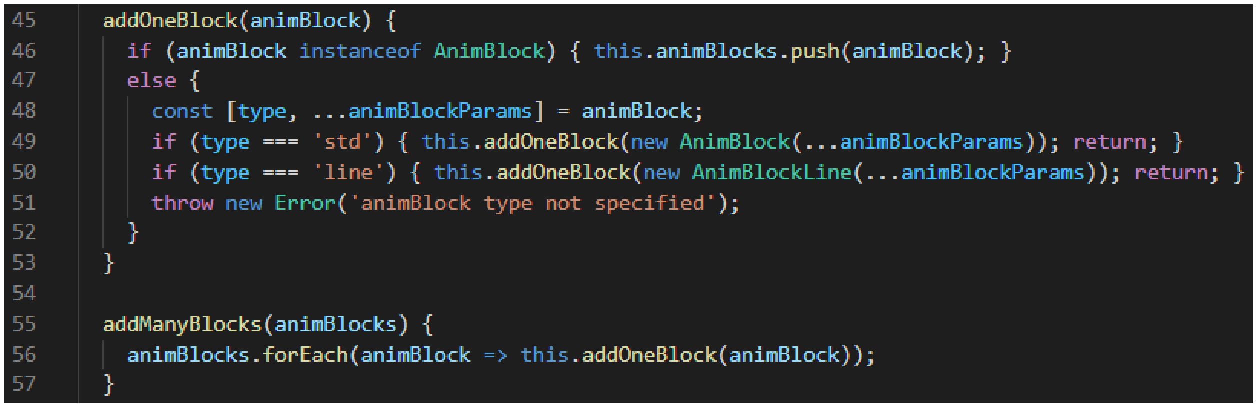
At a high-level view, an AnimSequence instance is essentially just a list of AnimBlock instances. It is necessary to have this separate class, however, because it provides several useful functionalities, such as rewinding a whole sequence of animations, printing descriptions for debugging purposes, and being able to "skip" currently-running animations by forcing them to finish instantly (this last feature will be explored later). The constructor for AnimSequence is shown in **Figure 4.17**; its properties and inner workings will be explained as we continue forward.



```
export class AnimSequence {
  static id = 0;

  timelineID; // set to match the id of the AnimTimeline to which it belongs
  parentTimeline; // pointer to parent AnimTimeline
  description = '<blank sequence description>';
  tag = ''; // helps idenfity current AnimSequence for using AnimTimeline's skipTo()
  animBlocks = []; // array of animBlocks

  constructor(animBlocks = null, options = null) {
    this.id = AnimSequence.id++;

    if (animBlocks) {
      if (animBlocks instanceof Array
        && (animBlocks[0] instanceof Array || animBlocks[0] instanceof AnimBlock)) { this.addManyBlocks(animBlocks); }
      else { this.addOneBlock(animBlocks); }
    }

    if (options) {
      this.description = options.description ?? this.description;
      this.tag = options.tag ?? this.tag;
      this.continueNext = options.continueNext; // decides if the next AnimSequence should automatically play after this one
      this.continuePrev = options.continuePrev; // decides if the prev AnimSequence should automatically play after this one
    }
  }
```

Figure 4.17: *AnimSequence constructor*

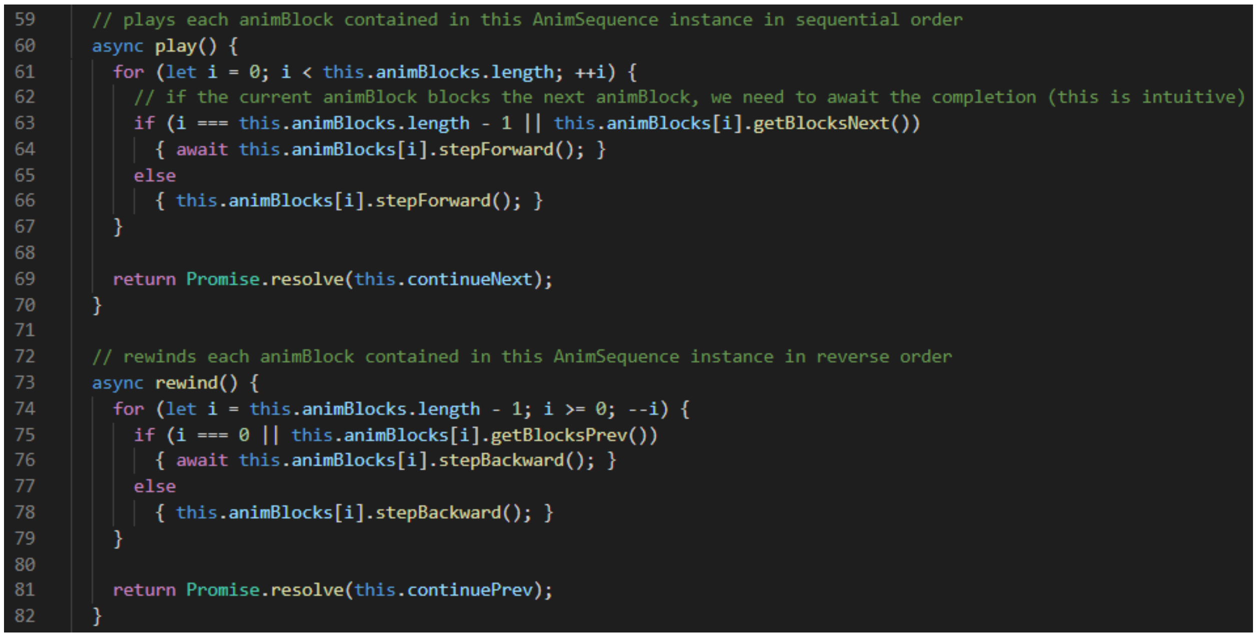
One way to add animBlocks to an animSequence is using either of the AnimSequence methods *addOneBlock()* or *addManyBlocks()*, shown in **Figure 4.18**. When adding an animBlock, we can either instantiate it using the AnimBlock constructor and the *new* keyword (in which case, the animBlock is just pushed to the list in line 46) or provide an array of parameters headed by the block type (an example of this is shown in **section 5.2**), which will be used to create an AnimBlock (or AnimBlockLine) instance. This is purely a preferential decision.



```
addOneBlock(animBlock) {
  if (animBlock instanceof AnimBlock) { this.animBlocks.push(animBlock); }
  else {
    const [type, ...animBlockParams] = animBlock;
    if (type === 'std') { this.addOneBlock(new AnimBlock(...animBlockParams)); return; }
    if (type === 'line') { this.addOneBlock(new AnimBlockLine(...animBlockParams)); return; }
    throw new Error('animBlock type not specified');
  }
}

addManyBlocks(animBlocks) {
  animBlocks.forEach(animBlock => this.addOneBlock(animBlock));
}
```

Figure 4.18: *AnimSequence methods for adding animBlocks*

Like an animBlock, an animSequence can be played forward or backward. This is done with the AnimSequence *async* methods *play()* and *rewind()* respectively; both methods are displayed in **Figure 4.19**. The general idea is that *play()* steps forward through each of the animSequence's animBlocks in sequential order (lines 61—67), while *rewind()* starts from the animSequence's last animBlock and steps backward through each of them in reverse sequential order. This satisfies the need to play a visualization in discrete chunks, which we will later see when we look at the AnimTimeline class.



```
59  // plays each animBlock contained in this AnimSequence instance in sequential order
60  async play() {
61    for (let i = 0; i < this.animBlocks.length; ++i) {
62      // if the current animBlock blocks the next animBlock, we need to await the completion (this is intuitive)
63      if (i === this.animBlocks.length - 1 || this.animBlocks[i].getBlocksNext())
64        { await this.animBlocks[i].stepForward(); }
65      else
66        { this.animBlocks[i].stepForward(); }
67    }
68
69    return Promise.resolve(this.continueNext);
70  }
71
72  // rewinds each animBlock contained in this AnimSequence instance in reverse order
73  async rewind() {
74    for (let i = this.animBlocks.length - 1; i >= 0; --i) {
75      if (i === 0 || this.animBlocks[i].getBlocksPrev())
76        { await this.animBlocks[i].stepBackward(); }
77      else
78        { this.animBlocks[i].stepBackward(); }
79    }
80
81    return Promise.resolve(this.continuePrev);
82  }
```

Figure 4.19: *AnimSequence play() and rewind() methods*

Notice that the loops in both *play()* and *rewind()* make a decision about how exactly to call an animBlock's stepping method; for the sake of brevity and without loss of generality, let us look only at *play()*. In line 64, the call to *stepForward()* is preceded by the keyword *await*. In an *async* function, *await* causes the function execution to pause until the awaited Promise is settled. This makes sense because it forces each upcoming animBlock to wait for the previous one to finish before beginning its own animation. However, we often want to play animations in parallel, so we need a way to specify whether a given animBlock should *not* block the next one. Recall the AnimBlock *applyOptions()* method shown in **Figure 4.2**; two of the options were *blocksNext* and *blocksPrev*, which are both set to *true* by default. For a given animBlock A, if *blocksNext* is set to *false*, then the next animBlock B in the sequence will *not* wait for A to finish before beginning its own animation, effectively letting A and B play in parallel.[4] *blocksPrev* does the same thing, but when we are stepping backward. Thus, in line 64 in *play()* (**Figure 4.19**), we only wait for an animBlock A to finish before playing the next animBlock B if *A.blocksNext* is *true*. This setup is important for 2 reasons: 1) It allows us to play several animations in parallel by having multiple animBlocks in a row with *false* blocking settings, and 2) Having separate options for blocking forward (*blocksNext*) and blocking backward (*blocksPrev*) means that we can customize the flow of animations in both directions. Point 2 is significant because it addresses the issue that some sequences are not useful to play exactly the same way both forward and in reverse. For example, in the context of an algorithm visualization, one sequence may, say, reveal several components in serial, but it may not be useful to undo those reveals in series when rewinding the sequence. Rather, undoing the entire sequence (or parts of it) in parallel would save time and lead to a better experience for users, who may often want to play and rewind parts of a visualization repeatedly. Of course, how to take advantage of this feature to make the best animation sequences is up to the discretion of the developer.

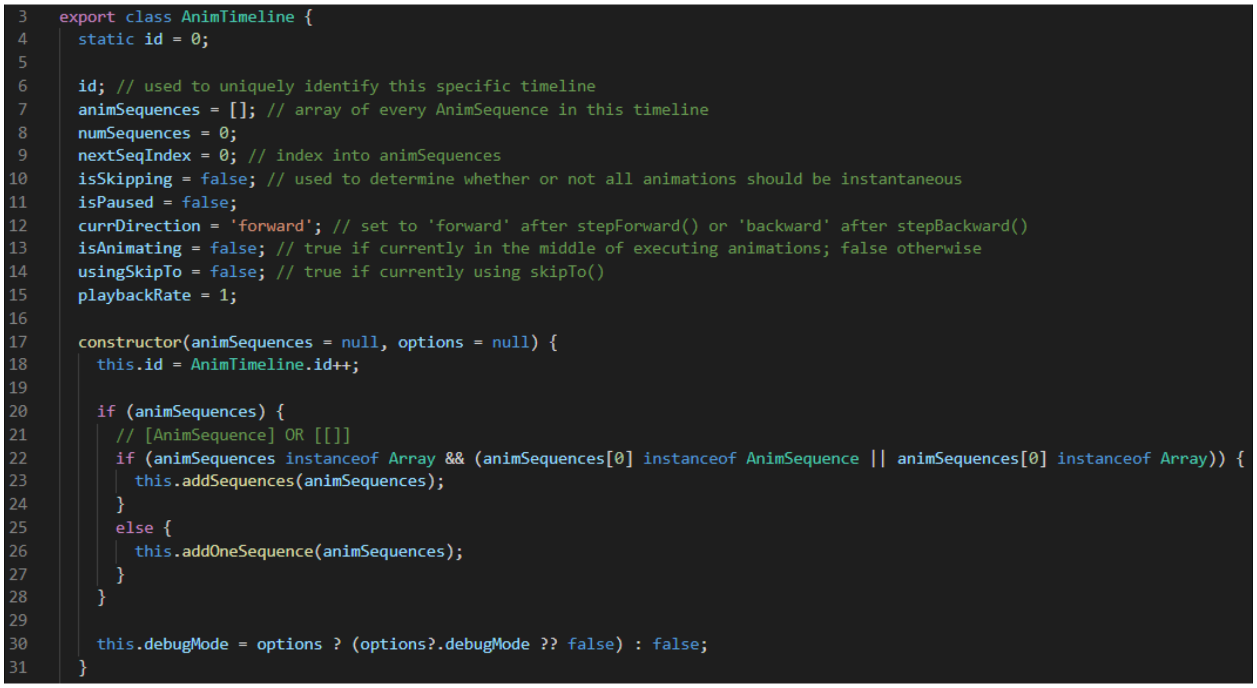
Once an animSequence has finished playing all of its animBlocks, it returns a Promise con-

[4]If an animBlock is the last one in the sequence, then its animation will be awaited regardless of its *blocksNext* value. The same holds for the first animBlock in a sequence regarding its *blocksPrev* value. This is to prevent the animSequence from returning its own Promise before its first or last animBlock has actually finished.

taining the value of *continueNext* (in the case of *play()*) or *continuePrev* (in the case of *rewind()*). Respectively, these member variables tell the parent AnimTimeline instance whether to autoplay the next or previous animSequence; we will see this when we examine the AnimTimeline class.

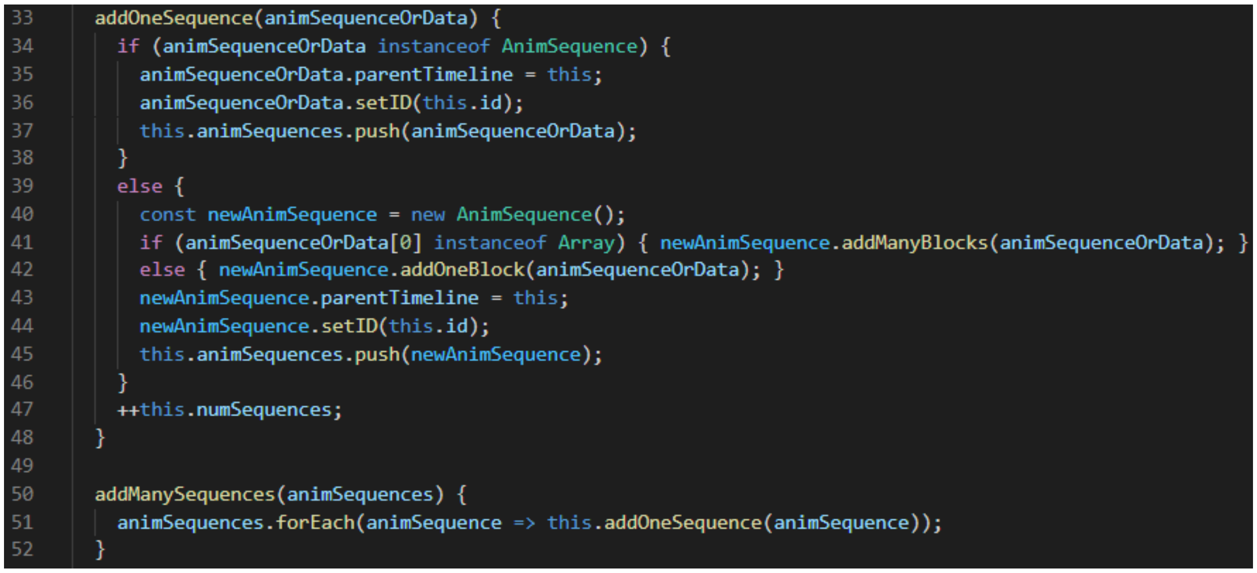
4.4 AnimTimeline

One or more AnimSequence instances can be grouped into an AnimTimeline instance, whose job is to control the playback of the entire timeline. The constructor for the class and the member variables are shown in **Figure 4.20**. One of the ways to add animSequences to an animTimeline is using either of the AnimTimeline methods *addOneSequence()* or *addManySequences()*, shown in **Figure 4.21**. The various conditionals just account for some of the different ways in which developers may add data for animSequences.



```
export class AnimTimeline {
  static id = 0;

  id; // used to uniquely identify this specific timeline
  animSequences = []; // array of every AnimSequence in this timeline
  numSequences = 0;
  nextSeqIndex = 0; // index into animSequences
  isSkipping = false; // used to determine whether or not all animations should be instantaneous
  isPaused = false;
  currDirection = 'forward'; // set to 'forward' after stepForward() or 'backward' after stepBackward()
  isAnimating = false; // true if currently in the middle of executing animations; false otherwise
  usingSkipTo = false; // true if currently using skipTo()
  playbackRate = 1;

  constructor(animSequences = null, options = null) {
    this.id = AnimTimeline.id++;

    if (animSequences) {
      // [AnimSequence] OR [[]]
      if (animSequences instanceof Array && (animSequences[0] instanceof AnimSequence || animSequences[0] instanceof Array)) {
        this.addSequences(animSequences);
      }
      else {
        this.addOneSequence(animSequences);
      }
    }

    this.debugMode = options ? (options?.debugMode ?? false) : false;
  }
```

Figure 4.20: AnimTimeline constructor



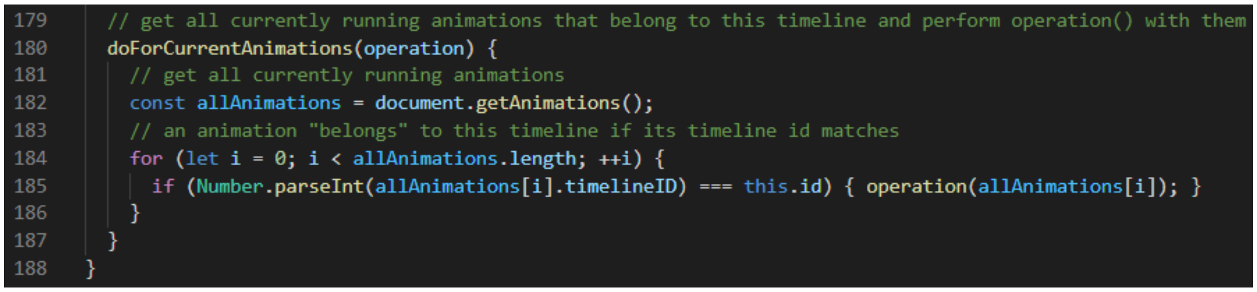
```
  addOneSequence(animSequenceOrData) {
    if (animSequenceOrData instanceof AnimSequence) {
      animSequenceOrData.parentTimeline = this;
      animSequenceOrData.setID(this.id);
      this.animSequences.push(animSequenceOrData);
    }
    else {
      const newAnimSequence = new AnimSequence();
      if (animSequenceOrData[0] instanceof Array) { newAnimSequence.addManyBlocks(animSequenceOrData); }
      else { newAnimSequence.addOneBlock(animSequenceOrData); }
      newAnimSequence.parentTimeline = this;
      newAnimSequence.setID(this.id);
      this.animSequences.push(newAnimSequence);
    }
    ++this.numSequences;
  }

  addManySequences(animSequences) {
    animSequences.forEach(animSequence => this.addOneSequence(animSequence));
  }
```

Figure 4.21: *AnimTimeline methods for adding animSequences*

Though the most crucial feature of AnimTimeline is the ability to step forward and backward through its animSequences, it will be easier to understand the relationships between the various playback functionalities if we examine some of the other features first. Let us start with figuring out which running animations belong to a particular animTimeline.

4.4.1 Animation identification

There are multiple scenarios within the AnimTimeline class where we need to perform some operation on every running animation, so the desired operation is passed to a separate method *doForCurrentAnimations()*, shown in **Figure 4.22**. With the WAAPI, we can obtain a list of all Animation instances that are currently in effect by calling *document.getAnimations()* as shown in line 182. However, we cannot just perform the operation on every animation in that array because there could be other animations running in the entire document that do *not* belong to the animTimeline. To solve this, all we need to do is assign some form of identification to each Animation instance upon being created. Upon every instantiation of AnimBlock (as well as AnimSequence), the new AnimBlock instance receives a reference to the parent AnimTimeline as well as a matching ID (the code has been excluded from this paper because it is not necessary to display). In line 69 of AnimBlock's *animate()* method (**Figure 4.7**), after the Animation instance has been created, we give it a *timelineID* property and match its value to the parent animTimeline's ID. Then, in lines 184—186 of *doForCurrentAnimations()* (**Figure 4.22**), we only perform *operation()* on Animation instances that we know belong to the animTimeline in question.

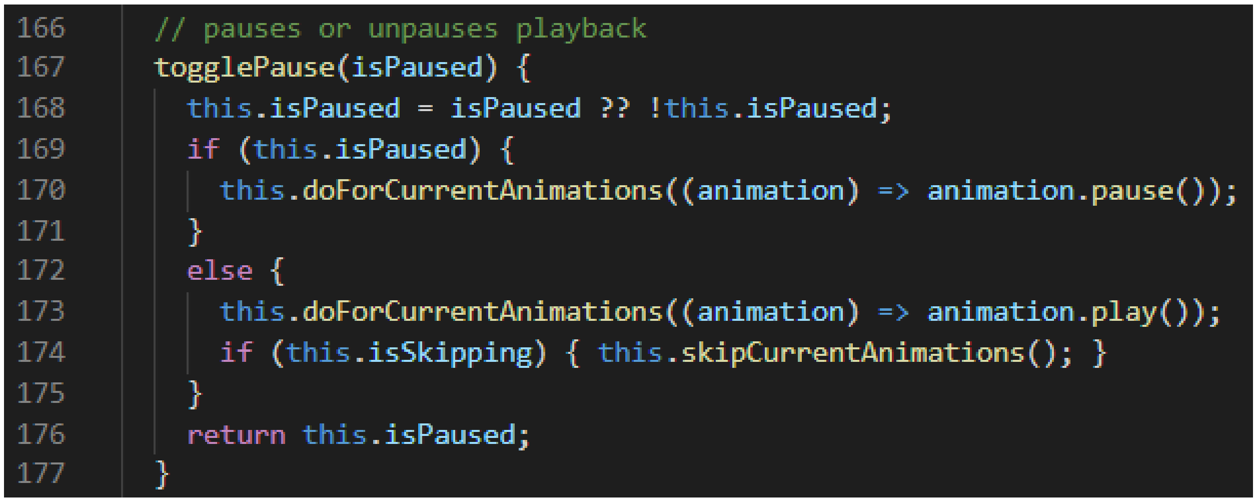


```
179   // get all currently running animations that belong to this timeline and perform operation() with them
180   doForCurrentAnimations(operation) {
181     // get all currently running animations
182     const allAnimations = document.getAnimations();
183     // an animation "belongs" to this timeline if its timeline id matches
184     for (let i = 0; i < allAnimations.length; ++i) {
185       if (Number.parseInt(allAnimations[i].timelineID) === this.id) { operation(allAnimations[i]); }
186     }
187   }
188 }
```

Figure 4.22: *AnimTimeline doForCurrentAnimations()*

4.4.2 Pausing playback

Oftentimes, a user may want to pause in the middle of an animation sequence to examine intermediate steps; thus, pausing is an essential feature for any algorithm visualization. With AnimTimeline, pausing is done with the *togglePause()* method (shown in **Figure 4.23**), which will toggle the current pause setting unless a Boolean value (*true* or *false*) is specifically passed in. If *this.isPaused* becomes *true* in line 168, then we need to call the Animation method *pause()* on every currently-running animation that belongs to the animTimeline. Of course, we can just use *doForCurrentAnimations()* as shown in line 170. If *this.isPaused* is set to *false*, we simply call *play()* on every active animation in line 173. In addition to resuming the animations, we also need to consider whether skipping is enabled (which will be discussed in the next subsection). We defined the behavior such that if playback is paused in the middle of a sequence and then skipping is enabled, the animations remain paused. If skipping is still enabled when playback is eventually resumed, it would be counterintuitive to have the rest of the current animSequence play at normal speed; thus, in this situation, the remaining animations to be played in the sequence are skipped in line 174.

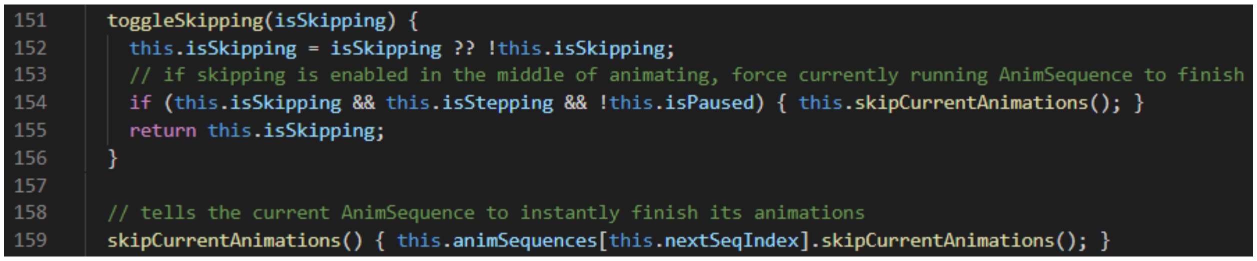


```
// pauses or unpauses playback
togglePause(isPaused) {
  this.isPaused = isPaused ?? !this.isPaused;
  if (this.isPaused) {
    this.doForCurrentAnimations((animation) => animation.pause());
  }
  else {
    this.doForCurrentAnimations((animation) => animation.play());
    if (this.isSkipping) { this.skipCurrentAnimations(); }
  }
  return this.isPaused;
}
```

Figure 4.23: *AnimTimeline togglePause()*

4.4.3 Enabling skipping

AnimSequences can be "skipped", which means instantly finishing the sequence of animations. This is possible by using the Animation method *finish()*—which, as the name suggests, instantly brings an animation to its ending—on every animation belonging to the animSequence. In AnimTimeline, skipping can be enabled with the *toggleSkipping()* method (**Figure 4.24**), which sets the member variable *isSkipping*.



```
toggleSkipping(isSkipping) {
  this.isSkipping = isSkipping ?? !this.isSkipping;
  // if skipping is enabled in the middle of animating, force currently running AnimSequence to finish
  if (this.isSkipping && this.isStepping && !this.isPaused) { this.skipCurrentAnimations(); }
  return this.isSkipping;
}

// tells the current AnimSequence to instantly finish its animations
skipCurrentAnimations() { this.animSequences[this.nextSeqIndex].skipCurrentAnimations(); }
```

Figure 4.24: *AnimTimeline toggleSkipping()*

With skipping enabled, stepping forward (or backward) will play the next (or previous) animSequence instantly. We see that this is true by examining line 85 in the AnimBlock *animate()* method (**Figure 4.7**)—if the parent animTimeline's *isSkipping* is *true*, then the animation will call *finish()* rather than *play()*. Now consider the scenario where we enable skipping while an animSequence is in the middle of playing; we would expect the rest of the sequence to instantly finish rather than continue to play normally. However, by this time, line 85 for the currently-playing animBlock(s) has already been executed, so *finish()* will not be called for their animations. This means that those animations that are currently active would play at normal speed, and *then* the upcoming animBlocks would skip their animations. To account for this, in line 154 of *toggleSkipping()* (**Figure 4.24**), we call the method *skipCurrentAnimations()* if skipping was enabled while animations were running. This in turn calls the current animSequence's own *skipCurrentAnimations()* method, which is shown in **Figure 4.25**. Here, as in *doForCurrentAnimations()*, we obtain a list of all of the active animations and filter them by their ID. In this case, we find all of the active animations belonging to the animSequence and call *finish()*.[5] With this, we achieve the expected behavior that enabling skipping in the middle of animating skips the current animations as well as the upcoming ones in the sequence.

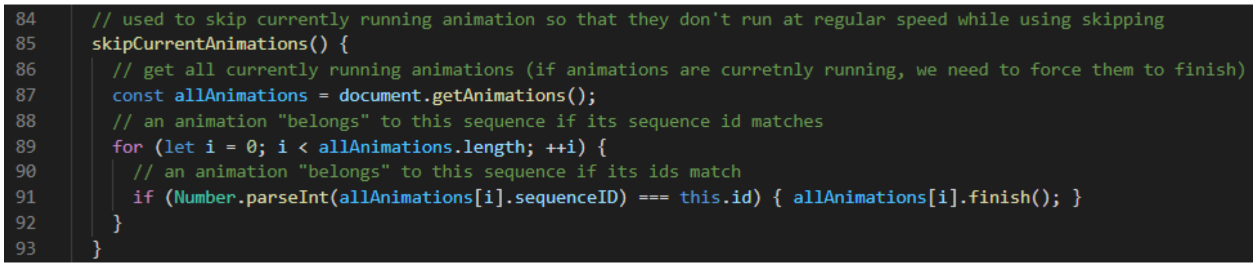

```
// used to skip currently running animation so that they don't run at regular speed while using skipping
skipCurrentAnimations() {
  // get all currently running animations (if animations are curretnly running, we need to force them to finish)
  const allAnimations = document.getAnimations();
  // an animation "belongs" to this sequence if its sequence id matches
  for (let i = 0; i < allAnimations.length; ++i) {
    // an animation "belongs" to this sequence if its ids match
    if (Number.parseInt(allAnimations[i].sequenceID) === this.id) { allAnimations[i].finish(); }
  }
}
```

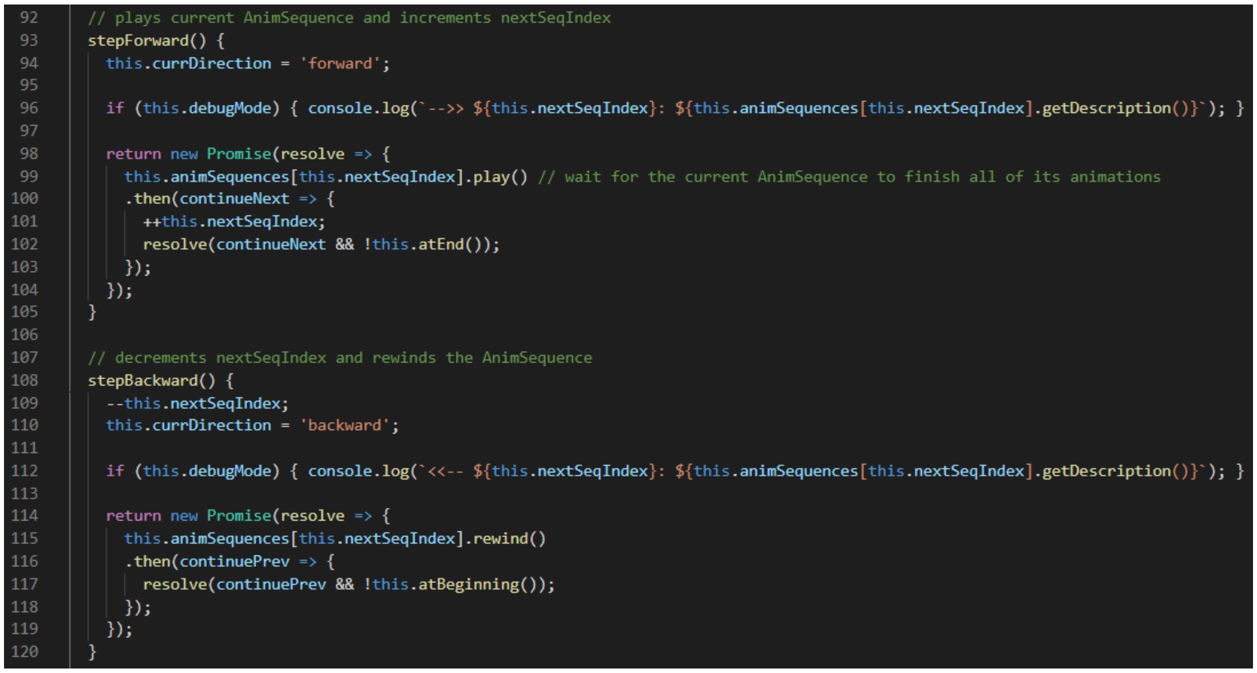
Figure 4.25: *AnimSequence skipCurrentAnimations()*

4.4.4 Stepping with AnimTimeline

The most crucial feature is the ability to step forwards or backward through an animTimeline's child animSequences. This can be done with the appropriately-named AnimTimeline methods *stepForward()* and *stepBackward()*, which are shown in **Figure 4.26** (we will soon see that we do not directly call these methods when using the framework though). The basics of the implementations are fairly simple. In *stepForward()* in lines 98—104, we return a Promise; within that Promise, we play the next animSequence, after which we increment the index into the animTime-

[5]It would also be possible to do this using *doForCurrentAnimations()* within the animTimeline. There was no particular reason for having the functionality within AnimSequence, but it shows that it may be feasible to run multiple animSequences within the same animTimeline in parallel and retain control over both. That is not explored in this paper, however.

line's list of AnimSequence instances. In *stepBackward()*, we reverse this by decrementing the index and then returning a Promise in which we rewind the most-recently-played animSequence.

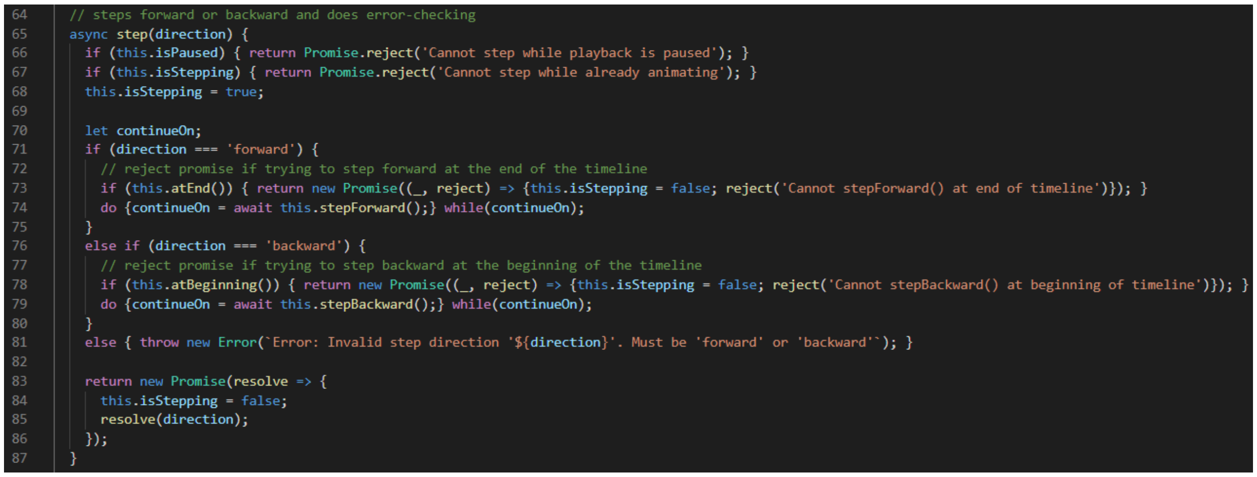


```
// plays current AnimSequence and increments nextSeqIndex
stepForward() {
  this.currDirection = 'forward';

  if (this.debugMode) { console.log(`-->> ${this.nextSeqIndex}: ${this.animSequences[this.nextSeqIndex].getDescription()}`); }

  return new Promise(resolve => {
    this.animSequences[this.nextSeqIndex].play() // wait for the current AnimSequence to finish all of its animations
    .then(continueNext => {
      ++this.nextSeqIndex;
      resolve(continueNext && !this.atEnd());
    });
  });
}

// decrements nextSeqIndex and rewinds the AnimSequence
stepBackward() {
  --this.nextSeqIndex;
  this.currDirection = 'backward';

  if (this.debugMode) { console.log(`<<-- ${this.nextSeqIndex}: ${this.animSequences[this.nextSeqIndex].getDescription()}`); }

  return new Promise(resolve => {
    this.animSequences[this.nextSeqIndex].rewind()
    .then(continuePrev => {
      resolve(continuePrev && !this.atBeginning());
    });
  });
}
```

Figure 4.26: *AnimTimeline step methods*

That much is straightforward, but notice that the calls to *then()* (which is a Promise method that executes after the Promise is settled) receive values for *continueNext* and *continuePrev*. We briefly saw these variable names at the end of **section 4.3**; they are the fulfillment values of the Promises returned in the AnimSequence methods *play()* and *rewind()* respectively (see **Figure 4.19**). As we mentioned back then, *continueNext* and *continuePrev*, which are *false* by default, determine whether the animSequence following the current one should be played automatically. Thus, the Promises returned by *stepForward()* and *stepBackward()* in AnimTimeline resolve to the values of *continueNext* and *continuePrev* respectively (unless there is no upcoming animSequence, in which case *false* is used since there is nothing to continue to).

All of this becomes relevant in the AnimTimeline method *step()*, shown in **Figure 4.27** which performs either *stepForward()* or *stepBackward()* depending on the value passed to the *direction* parameter; *step()* is the method that we intend for developers to call within their code. The first thing to notice is that *step()* returns a rejected Promise if playback is paused or animations are already currently in progress in lines 66 and 67, which is intuitive. Next, the member variable *isStepping*, which keeps track of whether or not a sequence is currently playing, is set to *true* in line 68 (hence the check for *isStepping* in line 67). Now take a look at lines 74 and 79—this is where the continuation values for automatically playing upcoming sequences are used. While the resolved value from the helper stepping method is true, we keep stepping—simple enough.



```
64  // steps forward or backward and does error-checking
65  async step(direction) {
66    if (this.isPaused) { return Promise.reject('Cannot step while playback is paused'); }
67    if (this.isStepping) { return Promise.reject('Cannot step while already animating'); }
68    this.isStepping = true;
69
70    let continueOn;
71    if (direction === 'forward') {
72      // reject promise if trying to step forward at the end of the timeline
73      if (this.atEnd()) { return new Promise((_, reject) => {this.isStepping = false; reject('Cannot stepForward() at end of timeline')}); }
74      do {continueOn = await this.stepForward();} while(continueOn);
75    }
76    else if (direction === 'backward') {
77      // reject promise if trying to step backward at the beginning of the timeline
78      if (this.atBeginning()) { return new Promise((_, reject) => {this.isStepping = false; reject('Cannot stepBackward() at beginning of timeline')}); }
79      do {continueOn = await this.stepBackward();} while(continueOn);
80    }
81    else { throw new Error(`Error: Invalid step direction '${direction}'. Must be 'forward' or 'backward'`); }
82
83    return new Promise(resolve => {
84      this.isStepping = false;
85      resolve(direction);
86    });
87  }
```

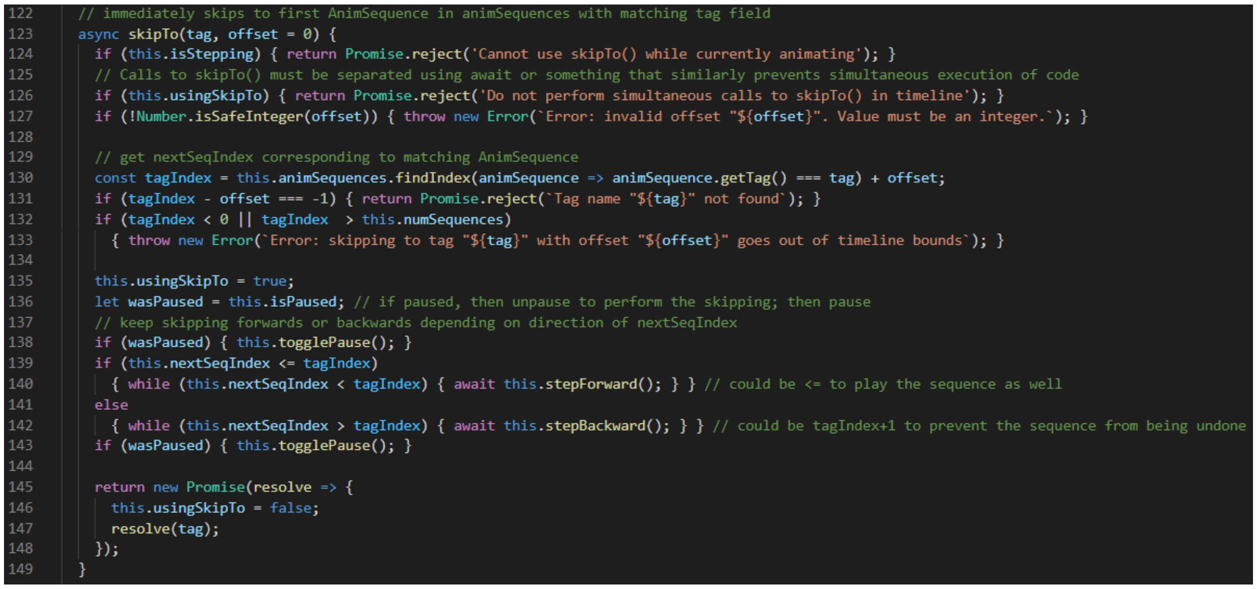
Figure 4.27: *AnimTimeline step()*

4.4.5 Jumping around in a timeline

A useful feature for any visualization of a sequence of events would be the ability to jump to any point in the timeline. In AnimTimeline, this is done using the *skipTo()* method, shown in **Figure 4.28**. Not only can the method be used to allow users to revisit different points without having to go through every sequence in-between, but it is also useful for debugging purposes (*skipTo()* made it substantially easier for us to jump to different parts within the WIS visualization during the development phase).

4.4.5.1 Basics of skipTo()

If we look back to AnimSequence's data members in **Figure 4.17**, we will notice that one of them is *tag*. These *tag* values can be used to skip to various animSequences in the animTimeline. As shown in line 130 of *skipTo()*, we first find the index of the first animSequence in the animTimeline that matches the desired tag. If a matching animSequence is found, then we set the animTimeline’s member variable *usingSkipTo* to *true*; in addition to *isSkipping*, *usingSkipTo* was one of the values that can force animations to finish immediately in the AnimBlock *animate()* method (**Figure 4.7**, line 85). Then, depending on whether the target animSequence is ahead of or behind the current place in the animTimeline, the method directly calls *stepForward()* or *stepBackward()* until we reach the target animSequence, effectively skipping through potentially several animation sequences instantly. *skipTo()* also allows an offset to be specified, which adds or subtracts from the index of the animSequence denoted by *tag*.



```
122  // immediately skips to first AnimSequence in animSequences with matching tag field
123  async skipTo(tag, offset = 0) {
124    if (this.isStepping) { return Promise.reject('Cannot use skipTo() while currently animating'); }
125    // Calls to skipTo() must be separated using await or something that similarly prevents simultaneous execution of code
126    if (this.usingSkipTo) { return Promise.reject('Do not perform simultaneous calls to skipTo() in timeline'); }
127    if (!Number.isSafeInteger(offset)) { throw new Error(`Error: invalid offset "${offset}". Value must be an integer.`); }
128
129    // get nextSeqIndex corresponding to matching AnimSequence
130    const tagIndex = this.animSequences.findIndex(animSequence => animSequence.getTag() === tag) + offset;
131    if (tagIndex - offset === -1) { return Promise.reject(`Tag name "${tag}" not found`); }
132    if (tagIndex < 0 || tagIndex  > this.numSequences)
133      { throw new Error(`Error: skipping to tag "${tag}" with offset "${offset}" goes out of timeline bounds`); }
134
135    this.usingSkipTo = true;
136    let wasPaused = this.isPaused; // if paused, then unpause to perform the skipping; then pause
137    // keep skipping forwards or backwards depending on direction of nextSeqIndex
138    if (wasPaused) { this.togglePause(); }
139    if (this.nextSeqIndex <= tagIndex)
140      { while (this.nextSeqIndex < tagIndex) { await this.stepForward(); } } // could be <= to play the sequence as well
141    else
142      { while (this.nextSeqIndex > tagIndex) { await this.stepBackward(); } } // could be tagIndex+1 to prevent the sequence from being undone
143    if (wasPaused) { this.togglePause(); }
144
145    return new Promise(resolve => {
146      this.usingSkipTo = false;
147      resolve(tag);
148    });
149  }
```

Figure 4.28: *AnimTimeline skipTo()*

4.4.5.2 Considerations

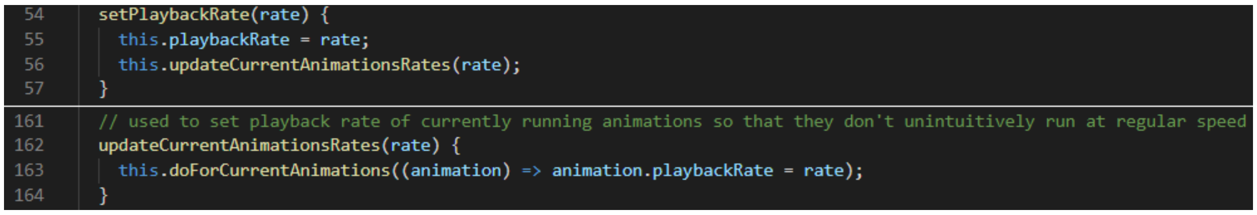
To prevent *skipTo()* from breaking the code, there are some combinations of actions that must be taken into consideration. The most obvious two are that we should not be allowed to use the method if animations are currently in progress (line 124) or if *skipTo()* is already in use (line 126); both restrictions prevent simultaneous execution of code. The third, less obvious consideration is what should happen when *skipTo()* is called while playback is paused (but not in the middle of a sequence of animations). We believe that it is intuitive to still jump to the target animSequence (if there is one) but keep the playback paused. Thus, in lines 136, 138, and 143, if playback was paused, we temporarily resume it so that the actual skipping can occur; then, it is paused again.

4.4.6 Printing sequence descriptions

Besides *skipTo()*, another debugging feature that should be available to developers is the ability to set and print descriptions for every animation sequence. With AnimTimeline, this can be done easily. In the AnimTimeline constructor (see **Figure 4.20**), we set *debugMode* to *true* (line 30) using the *options* parameter. If debugging mode is enabled, then *stepForward()* and *stepBackward()* print the current animSequence's description to the console along with the step number (lines 96 and 112). As for how to set an animSequence's description, one of the data members in the AnimSequence class is *description* (see **Figure 4.17**). We can set each description to whatever helps us keep track of our place in the timeline, which expedites the debugging process.

4.4.7 Adjusting playback rate

Any animated visualization should allow the user to adjust the playback rate at any time. In AnimTimeline, the structure of the code makes this simple—as with enabling skipping (**Figure 4.24**), any newly created Animation instances will receive the correct playback rate (see **Figure 4.7**, line 76), while *doForCurrentAnimations()* is used to update any animations that are already active (see the AnimTimeline method *setPlaybackRate()* in **Figure 4.29**).



```
setPlaybackRate(rate) {
  this.playbackRate = rate;
  this.updateCurrentAnimationsRates(rate);
}
```

```
// used to set playback rate of currently running animations so that they don't unintuitively run at regular speed
updateCurrentAnimationsRates(rate) {
  this.doForCurrentAnimations((animation) => animation.playbackRate = rate);
}
```

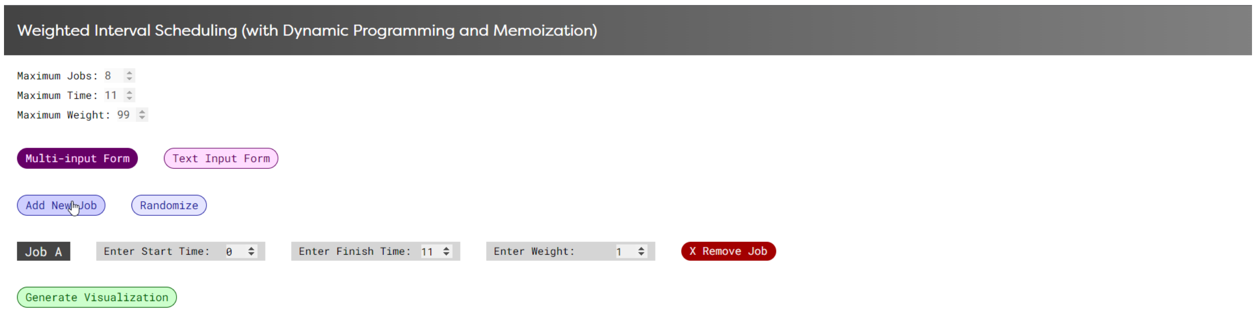
Figure 4.29: *AnimTimeline setPlaybackRate()*

5. THE WIS VISUALIZATION

To test the utility of AnimTimeline, we created a visually-rich, animated visualization for weighted interval scheduling (WIS) using dynamic programming (DP) and memoization (see **section 2.2** for a brief overview of what that is). While the primary objective is to show off the effectiveness of the tools offered by our framework, we should also examine the design and experience of the whole visualization. Thus, we will start our demonstration from the beginning—inputting jobs. Next, we will demonstrate the visualization itself, which was, of course, created using the AnimTimeline framework, and we will take note of the design decisions employed in the process. It is possible that the interface of the web page will have changed after this paper is finished, but the basic structure will remain the same.

5.1 Inputting Jobs

The first thing we see when loading the web page is a screen that allows us to input the job parameters (see **Figure 5.1**). At the top of the page, we can see the constraints for the job inputs (they cannot be modified by the user); these could be considered arbitrary limitations because our system can technically function with any number of jobs and with no limitation on the time and weight, but most traditional hand-written demonstrations of WIS use a maximum of 8 jobs and 11 units of time. We applied the same limitations for the sake of familiarity and to ensure that the visualization looks clean (it would be hard to justify allowing a user to input 500 jobs, for example). The input form shown in **Figure 5.1** is one of two forms that can be used to fill out the job parameters; we will call it the "multi-input" job form.


Weighted Interval Scheduling (with Dynamic Programming and Memoization)

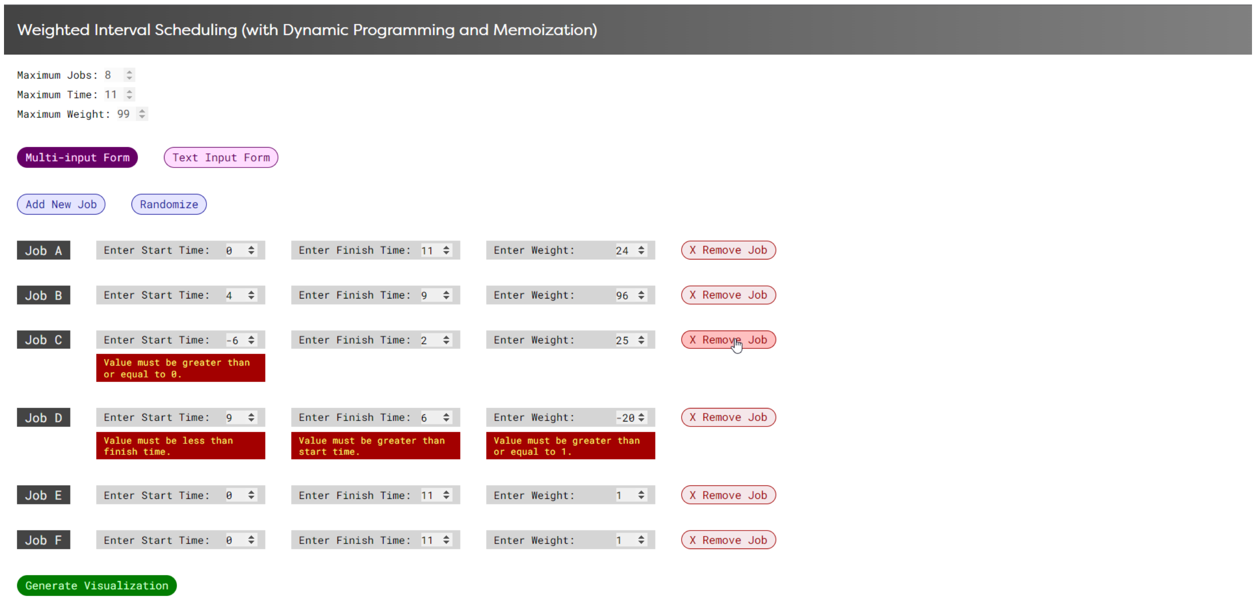
Maximum Jobs: 8
Maximum Time: 11
Maximum Weight: 99

Job A
Enter Start Time: 0
Enter Finish Time: 11
Enter Weight: 1

Figure 5.1: *Main menu*

5.1.1 Multi-input job form

The Add New Job button can be used to add new rows, and we can modify each job's start time, finish time, and weight. The Randomize button will generate a random number of jobs with random inputs, which is useful when we just want any job inputs without having to add them manually. Job input rows can be deleted using the Remove button, but the button will be disabled if there is only one job row currently. To maximize the intuitiveness of the form, extensive form validation is performed every time the user modifies a field, and errors are displayed under each corresponding problematic field as shown in **Figure 5.2**. As long as errors are present, the Generate Visualization button will be disabled; the user can fix this by correcting the inputs or by simply deleting the problematic rows.



Weighted Interval Scheduling (with Dynamic Programming and Memoization)

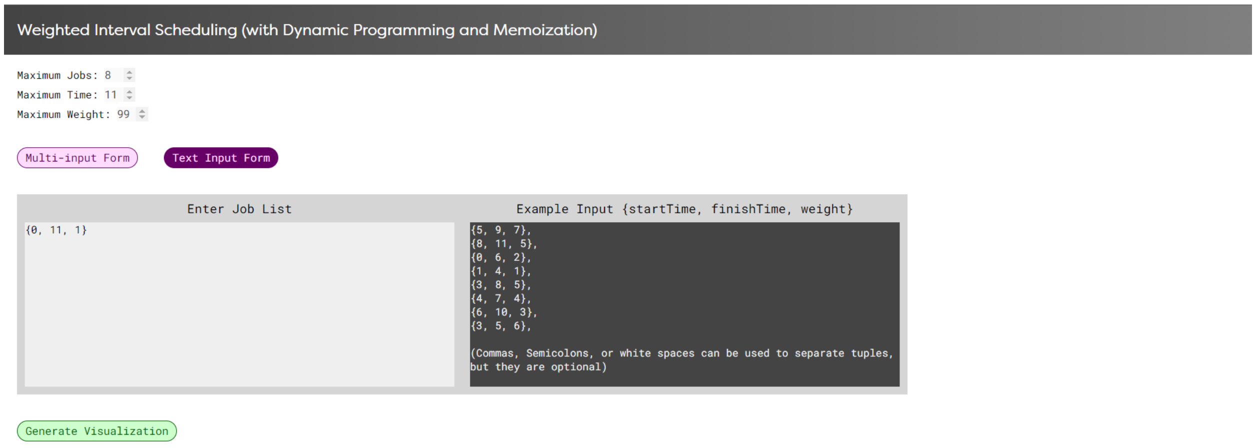
Maximum Jobs: 8
Maximum Time: 11
Maximum Weight: 99

Job A	Enter Start Time: 0	Enter Finish Time: 11	Enter Weight: 24	X Remove Job
Job B	Enter Start Time: 4	Enter Finish Time: 9	Enter Weight: 96	X Remove Job
Job C	Enter Start Time: -6 Value must be greater than or equal to 0.	Enter Finish Time: 2	Enter Weight: 25	X Remove Job
Job D	Enter Start Time: 9 Value must be less than finish time.	Enter Finish Time: 6 Value must be greater than start time.	Enter Weight: -20 Value must be greater than or equal to 1.	X Remove Job
Job E	Enter Start Time: 0	Enter Finish Time: 11	Enter Weight: 1	X Remove Job
Job F	Enter Start Time: 0	Enter Finish Time: 11	Enter Weight: 1	X Remove Job

Figure 5.2: *Form validation for the multi-input form*

5.1.2 *Text input form*

Though the multi-input form is intuitive, it is not practical in the scenario where a professor (or student) wants to run the same set of jobs upon each visit. This is where the second job form, the text input form, comes into play; it is shown in **Figure 5.3**. It can be switched to by pressing the Text Input Form button.



Weighted Interval Scheduling (with Dynamic Programming and Memoization)

Maximum Jobs: 8
Maximum Time: 11
Maximum Weight: 99

Enter Job List

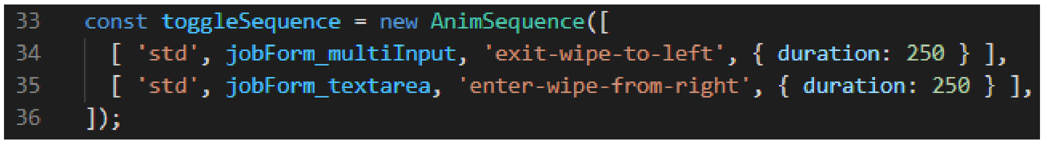
```
{0, 11, 1}
```

Example Input {startTime, finishTime, weight}

```
{5, 9, 7},
{8, 11, 5},
{0, 6, 2},
{1, 4, 1},
{3, 8, 5},
{4, 7, 4},
{6, 10, 3},
{3, 5, 6},

(Commas, Semicolons, or white spaces can be used to separate tuples,
but they are optional)
```

Figure 5.3: *Text input job form*

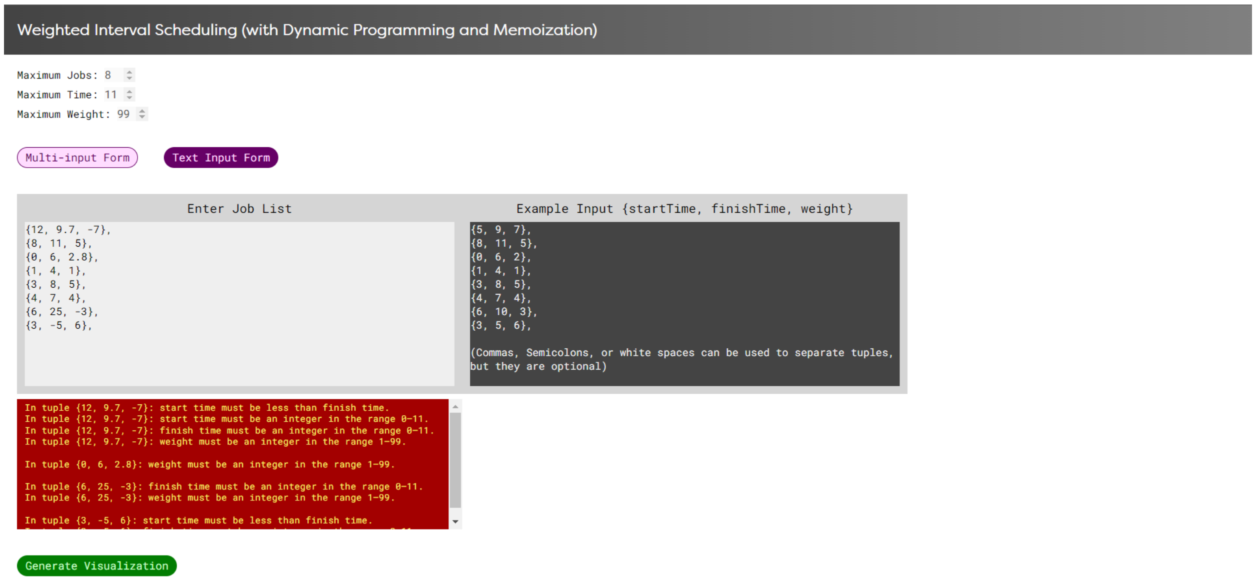
It is worth noting that toggling the job form mode plays a wiping animation to hide the current form and another to reveal the other form; this was done using a standalone animSequence with two animBlocks (shown in **Figure 5.4**). It was not necessarily our intention to let AnimSequence function independently of AnimTimeline, but it (as well as AnimBlock) is actually quite useful on its own.



```
const toggleSequence = new AnimSequence([
  [ 'std', jobForm_multiInput, 'exit-wipe-to-left', { duration: 250 } ],
  [ 'std', jobForm_textarea, 'enter-wipe-from-right', { duration: 250 } ],
]);
```

Figure 5.4: *An animSequence for animating the switch between job form modes*

In any case, with this job form, we can type or paste in several tuples, which will be used to set up the job data. **Figure 5.3** explains the format in the Example Input box. As with the multi-input form, the best user experience entails giving a detailed description for invalid inputs, so we utilized regex to display a list of errors related to the input, separated by tuple (exemplified in **Figure 5.5**).



Weighted Interval Scheduling (with Dynamic Programming and Memoization)

Maximum Jobs: 8
Maximum Time: 11
Maximum Weight: 99

Enter Job List

```
{12, 9.7, -7},
{8, 11, 5},
{0, 6, 2.8},
{1, 4, 1},
{3, 8, 5},
{4, 7, 4},
{6, 25, -3},
{3, -5, 6},
```

Example Input {startTime, finishTime, weight}

```
{5, 9, 7},
{8, 11, 5},
{0, 6, 2},
{1, 4, 1},
{3, 8, 5},
{4, 7, 4},
{6, 10, 3},
{3, 5, 6},

(Commas, Semicolons, or white spaces can be used to separate tuples,
but they are optional)
```

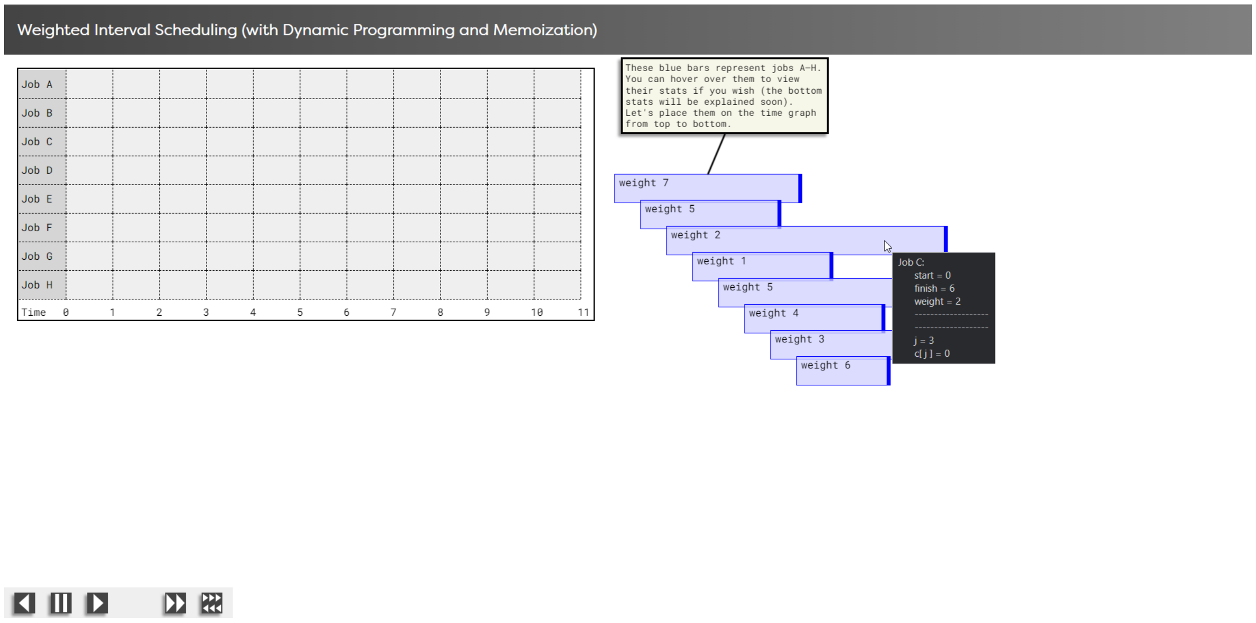
```
In tuple {12, 9.7, -7}: start time must be less than finish time.
In tuple {12, 9.7, -7}: start time must be an integer in the range 0-11.
In tuple {12, 9.7, -7}: finish time must be an integer in the range 0-11.
In tuple {12, 9.7, -7}: weight must be an integer in the range 1-99.

In tuple {0, 6, 2.8}: weight must be an integer in the range 1-99.

In tuple {6, 25, -3}: finish time must be an integer in the range 0-11.
In tuple {6, 25, -3}: weight must be an integer in the range 1-99.

In tuple {3, -5, 6}: start time must be less than finish time.
```

Figure 5.5: *Form validation for text area form*

5.2 Playing the Visualization

Once we finish inputting our jobs, we can start the visualization by clicking the Generate Visualization button. At this point, it would be more useful to demonstrate the final product on the web page instead of walking through every single step using static images, so the walk-through here will be brief and will not cover the entire visualization. Instead, a video demonstration can be viewed at **https://www.youtube.com/playlist?list=PLQ9MSztsBAbDZUZQhdBzz4mGxKW5m1fsL**. Because the links for the website itself and the GitHub repository may change, those will not be provided in this paper; instead, they can be found in the description of the aforementioned YouTube video, where they can be updated.

When we begin the visualization, we are met with an empty time graph—whose number of rows depends on the number of jobs—and a stack of blue job bars. We immediately see AnimBlockLine come into play when we see a line extend upward from the topmost bar and then end where a text box appears shortly afterward (see **Figure 5.6**).


Weighted Interval Scheduling (with Dynamic Programming and Memoization)

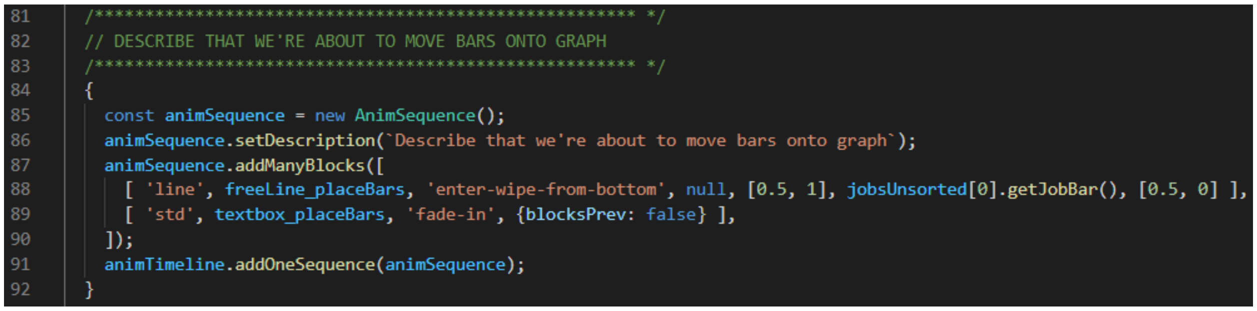
```
These blue bars represent jobs A–H.
You can hover over them to view
their stats if you wish (the bottom
stats will be explained soon).
Let's place them on the time graph
from top to bottom.
```

```
Job C:
    start = 0
    finish = 6
    weight = 2
    ------------------
    ------------------
    j = 3
    c[ j ] = 0
```

Figure 5.6: *Beginning of the WIS visualization*

The code snippet for that animation sequence is shown in **Figure 5.7**. First, an AnimSequence instance is created in line 85. Its description (which is printed to the console if debugging on the animTimeline is enabled) is set in line 86. The animations for the line and textbox are added to the sequence in lines 87—90. As we can see, when the animSequence is played forward, the line is set to wipe up from the bottom upon entering and point between the middle-bottom

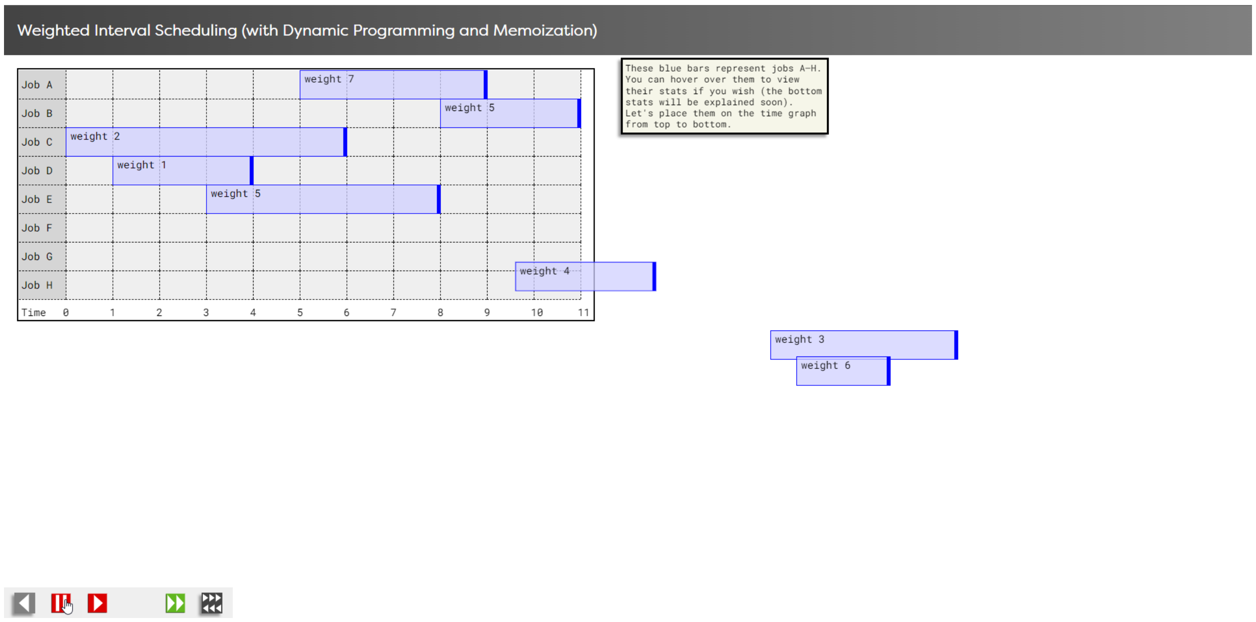
([0.5, 1]) of the textbox and the middle-top ([0.5, 0]) of the topmost job bar. The text box simply fades in. Notice that the text box animation has *blocksPrev* set to *true*; this means that stepping backward through the sequence will cause both the line and the text box to exit at the same time. After adding the animBlock and animBlockLine to the animSequence, we add the animSequence to the animTimeline. It is worth noting that the free-line was able to point to the textbox location before it was visible; this is because we set up the CSS so that adding the "hidden" class to the text boxes just sets *visibility* to *hidden* instead of setting *display* to *none*. Along with being absolutely positioned, the text box is then able to have its bounding box values available even while invisible.



```
/******************************************************* */
// DESCRIBE THAT WE'RE ABOUT TO MOVE BARS ONTO GRAPH
/******************************************************* */
{
  const animSequence = new AnimSequence();
  animSequence.setDescription(`Describe that we're about to move bars onto graph`);
  animSequence.addManyBlocks([
    [ 'line', freeLine_placeBars, 'enter-wipe-from-bottom', null, [0.5, 1], jobsUnsorted[0].getJobBar(), [0.5, 0] ],
    [ 'std', textbox_placeBars, 'fade-in', {blocksPrev: false} ],
  ]);
  animTimeline.addOneSequence(animSequence);
}
```

Figure 5.7: *Code snippet for first visualization animation*

See **Figure 5.8** for the next animation sequence. Here, we utilize translations to move the job bars onto the time graph in the order that the user defined them. On the bottom left, we see several buttons for controlling playback. Hovering over them describes what they do and what their keyboard shortcuts are. As shown, the colors (as well as the cursor) change depending on what buttons are activated and when. In **Figure 5.8**, we can tell that playback was paused in the middle of playing the sequence forward. Thus, the rewind button is grayed out, while the pause and play buttons are depressed and red; additionally, the rewind and play buttons are both disabled, which is indicated by changing the cursor to a red cross when hovering over them (this is not shown in the figure). The fast-forward button is also currently held down, indicated by the green coloring (the separate coloring was used because that button is only active as long as it is held down).


Weighted Interval Scheduling (with Dynamic Programming and Memoization)

These blue bars represent jobs A–H. You can hover over them to view their stats if you wish (the bottom stats will be explained soon). Let's place them on the time graph from top to bottom.

Figure 5.8: Moving the job bars onto the graph in unsorted order

The code snippet for that animation sequence is shown in **Figure 5.7**. This time, we make use of translations. Each job bar element is moved to its designated spot on the graph because of lines 106 and 107. Recall how much work is done behind the scenes in AnimBlock, AnimSequence, and AnimTimeline to make this small snippet of code work.

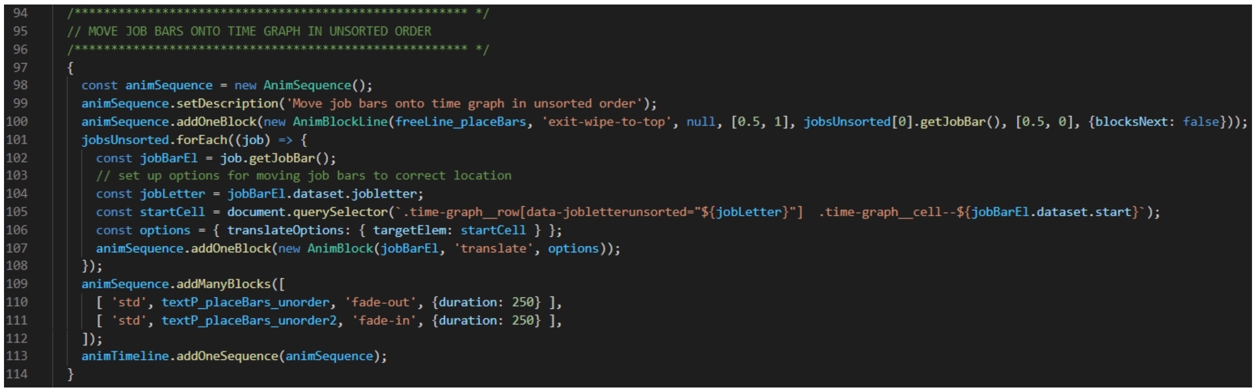
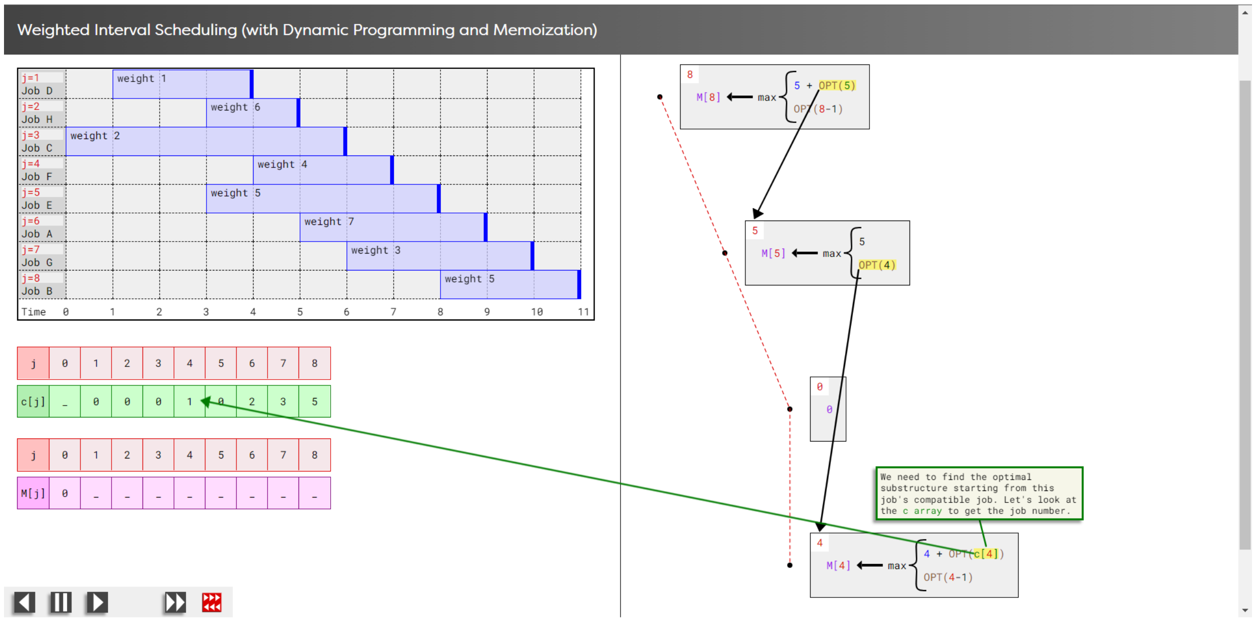

```
94   /****************************************************** */
95   // MOVE JOB BARS ONTO TIME GRAPH IN UNSORTED ORDER
96   /****************************************************** */
97   {
98     const animSequence = new AnimSequence();
99     animSequence.setDescription('Move job bars onto time graph in unsorted order');
100    animSequence.addOneBlock(new AnimBlockLine(freeLine_placeBars, 'exit-wipe-to-top', null, [0.5, 1], jobsUnsorted[0].getJobBar(), [0.5, 0], {blocksNext: false}));
101    jobsUnsorted.forEach((job) => {
102      const jobBarEl = job.getJobBar();
103      // set up options for moving job bars to correct location
104      const jobLetter = jobBarEl.dataset.jobletter;
105      const startCell = document.querySelector(`.time-graph__row[data-jobletterunsorted="${jobLetter}"]  .time-graph__cell--${jobBarEl.dataset.start}`);
106      const options = { translateOptions: { targetElem: startCell } };
107      animSequence.addOneBlock(new AnimBlock(jobBarEl, 'translate', options));
108    });
109    animSequence.addManyBlocks([
110      [ 'std', textP_placeBars_unorder, 'fade-out', {duration: 250} ],
111      [ 'std', textP_placeBars_unorder2, 'fade-in', {duration: 250} ],
112    ]);
113    animTimeline.addOneSequence(animSequence);
114  }
```

Figure 5.9: Code snippet for moving the job bars onto the graph

Skipping ahead in the visualization, we begin forming the tree that represents the recursive process of computing the optimal weight. **Figure 5.10** displays the tree near the beginning of its formation. This demonstrates the utility of continuously tracking endpoints—we can see that the endpoints of the green arrow pointing across the screen are at their targets; this is in despite the fact that the right side of the screen has evidently been scrolled downward as the left side remains fixed. When paired with our animation presets that can be used to draw the arrow extending (such as *enter-wipe-from-right*), we have a straightforward way of drawing the user's eyes to important details and ensuring that the visuals are responsive.



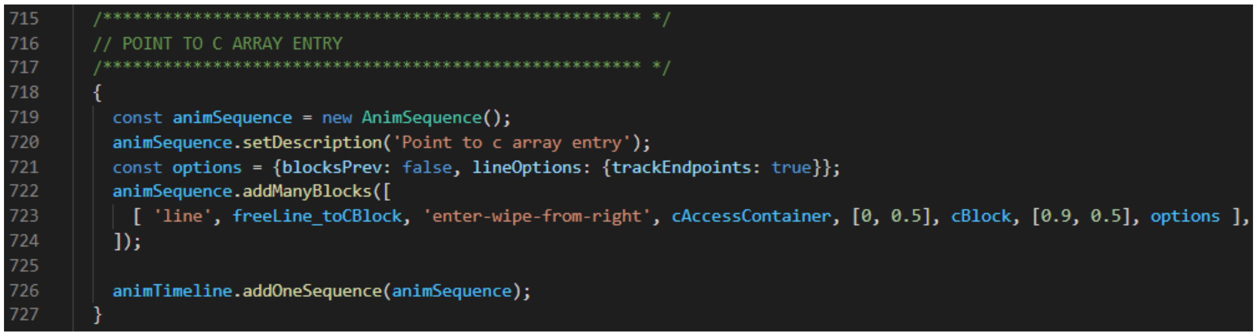
j	0	1	2	3	4	5	6	7	8
c[j]	_	0	0	0	1	0	2	3	5

j	0	1	2	3	4	5	6	7	8
M[j]	0	_	_	_	_	_	_	_	_

We need to find the optimal substructure starting from this job's compatible job. Let's look at the c array to get the job number.

Figure 5.10: *Partway into the tree for finding the optimal weight*

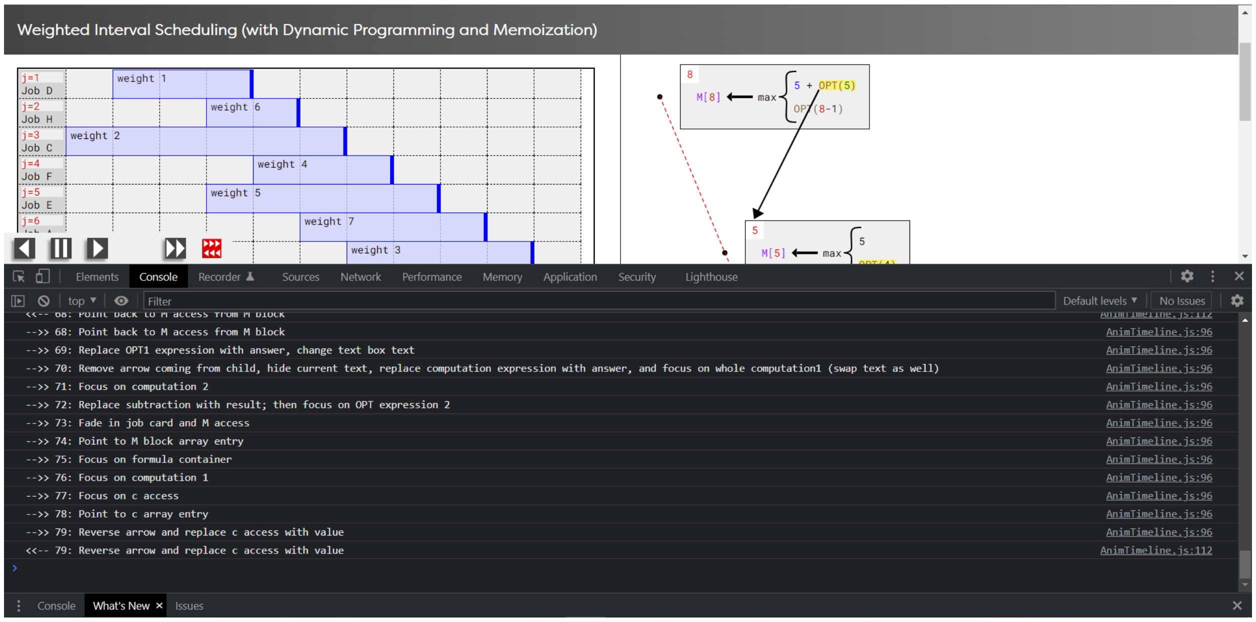
The code snippet for that animation sequence is shown in **Figure 5.11**. *trackEndpoints* is indeed set to *true* using the *options* object in line 721, which explains why the free-line is updating itself accordingly.



```
715  /******************************************************* */
716  // POINT TO C ARRAY ENTRY
717  /******************************************************* */
718  {
719    const animSequence = new AnimSequence();
720    animSequence.setDescription('Point to c array entry');
721    const options = {blocksPrev: false, lineOptions: {trackEndpoints: true}};
722    animSequence.addManyBlocks([
723      [ 'line', freeLine_toCBlock, 'enter-wipe-from-right', cAccessContainer, [0, 0.5], cBlock, [0.9, 0.5], options ],
724    ]);
725
726    animTimeline.addOneSequence(animSequence);
727  }
```

Figure 5.11: *Code snippet for pointing to the c array*

We have also seen throughout the past examples that the AnimSequence method *setDescription()* was being used. Recall that if debugging for an animTimeline is enabled, all animSequences print their descriptions to the console (see **subsection 4.4.6**). **Figure 5.12** displays the same view of **Figure 5.10** but with the console visible. Apparently, this is step 78 of the whole visualization because the last printout indicates that we stepped backwards from step 79, so if we experience a bug in development and see "–» 78: Point to c array entry" in the console, we know where to search for the problem first.



Weighted Interval Scheduling (with Dynamic Programming and Memoization)
j=1 Job D
j=2 Job H
j=3 Job C
j=4 Job F
j=5 Job E
j=6
weight 1
weight 6
weight 2
weight 4
weight 5
weight 7
weight 3
8
M[8] ⟵ max 5 + OPT(5) OPT(8-1)
5
M[5] ⟵ max 5
Elements Console Recorder Sources Network Performance Memory Application Security Lighthouse
top Filter Default levels No Issues
Console What's New Issues

```
  -->> 68: Point back to M access from M block                     AnimTimeline.js:96
  -->> 69: Replace OPT1 expression with answer, change text box text   AnimTimeline.js:96
  -->> 70: Remove arrow coming from child, hide current text, replace computation expression with answer, and focus on whole computation1 (swap text as well)   AnimTimeline.js:96
  -->> 71: Focus on computation 2                                  AnimTimeline.js:96
  -->> 72: Replace subtraction with result; then focus on OPT expression 2   AnimTimeline.js:96
  -->> 73: Fade in job card and M access                           AnimTimeline.js:96
  -->> 74: Point to M block array entry                            AnimTimeline.js:96
  -->> 75: Focus on formula container                              AnimTimeline.js:96
  -->> 76: Focus on computation 1                                  AnimTimeline.js:96
  -->> 77: Focus on c access                                       AnimTimeline.js:96
  -->> 78: Point to c array entry                                  AnimTimeline.js:96
  -->> 79: Reverse arrow and replace c access with value           AnimTimeline.js:96
  <<-- 79: Reverse arrow and replace c access with value          AnimTimeline.js:112
>
```

Figure 5.12: Descriptions printed to the console

This concludes the demonstration for the purposes of the paper; we highly advise watching the video demonstration at **https://www.youtube.com/playlist?list=PLQ9MSztsBAbDZUZQhdBzz4mGxKW5m1fsL**, which covers the visualization in more detail and serves as an extension of this paper.

6. CONCLUSION

This research addresses the challenge of generating new tools for helping students to advance their computational thinking and understand complex algorithms. Computer science education is an active field of research where significant progress has been made in topics such as introductory programming, auto-grading, and plagiarism detection. There has been some progress in leveraging the huge advances in computing power to build educational tools, but in complex topics covered in 400-level courses (such as analysis of algorithms), instruction remains the same as it was decades ago.

This thesis proposes a web animation framework for visualizing algorithms across timelines so that students can observe how data structures change as the algorithm execution progresses. The framework supports several playback features; namely, it allows users to step backward and forward through program timeline, effectively playing animations normally and in reverse. This alone gives students an opportunity to analyze the data changes and think about the process at their own pace, but several other features such as changing the playback rate, pausing in the middle of animations, skipping animation sequences, animated lines, and debugging tools make the framework a robust tool. Our approach allows for seamless integration into normal front-end development and UI/UX design, which lets developers integrate text (including mathematical formulas) and other graphics into the visualization. This aids students learning about sophisticated problem-solving techniques that may apply to several problems. The prototype implemented in this work achieved a level of responsiveness and efficiency well beyond the planned progress for this thesis.

Two important next steps in this research area are (1) to demonstrate the generality of the proposed framework by deploying it to build animated visualizations for additional classes of algorithms and (2) to assess the impact of the tool on student learning by carrying out user studies.

REFERENCES

[1] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002.

[2] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce, and S. H. Edwards, "Algorithm visualization: The state of the field," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 3, pp. 1–22, 2010.

[3] B. Shneiderman, "The eyes have it: a task by data type taxonomy for information visualizations," in *Proceedings 1996 IEEE Symposium on Visual Languages*, pp. 336–343, 1996.

[4] J. Kleinberg and E. Tardos, *Algorithm Design*. USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[5] J. A. Velázquez-Iturbide, A. Pérez-Carrasco, and J. Urquiza-Fuentes, "Srec: An animation system of recursion for algorithm courses," *SIGCSE Bull.*, vol. 40, pp. 268–277, June 2008.

[6] J. A. Velázquez-Iturbide and A. Pérez-Carrasco, "Systematic development of dynamic programming algorithms assisted by interactive visualization," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 71–76, Association for Computing Machinery, 2016.

[7] S. Simonák, "Using algorithm visualizations in computer science education," *Central European Journal of Computer Science*, vol. 4, pp. 183–190, 2014.

[8] S. Ssimonak, "Algorithm visualizations as a way of increasing the quality in computer science education," *2016 IEEE 14th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pp. 153–157, 2016.

[9] Mozilla, "Resources for developers by developers." **https://developer.mozilla.org/en-US/**. Last visited 4/3/22.