Overview of Stewart.js library by Robert Eisele

For starters, a short description of the Stewart.js library will be provided, along with an explanation to the existing project's code.

1.1 Code overview:

The Stewart Platform code is written to work on web application, and exclusively relies on HTML5 and JavaScript. It was written by Robert Eisele.

The libraries used are the following:

- P5.js → Graphical representation of the Stewart Platform.
- quaternion.js → To perform rotations in the 3D space.
- bezier.js → Used to parse SVG path files to a comprehensible set of instructions for the animation of the logos.

Apart from those libraries, for the visual representation of the code, we have a file called default.html that takes care of the user interface on the webpage.

This html file links all the libraries described, apart from the main JavaScript file: stewart.js.

There is a small script inside the HTML file that calls all the main functions of the stewart.js script.

1.1.1 HTML Script overview (default.html)

Declared functions and methods' structure:

`setupPlatform()`

- `sketch(p)`
  - `p.setup()`
  - `p.draw()`
- `createSVGImage(id, root, d, box)`
  - `svg.onclick()`

Event functions

- `document.getElementById("canvas").onmousedown(ev)`
- `document.onmouseup()`

- `document.onmousemove(ev)`
- `document.onkeydown(e)`
- `window.onload`

Declared functions explained:

`setupPlatform()`: Function to set up the platform and initialize the animation, it contains all the other functions, and will execute when loading the window, with `window.onload = setupPlatform`

`sketch(p)`: p5.js sketch constructor function, sets up the p5.js canvas. The passed parameter references this sketch. In instances of this object, we will be referring to this sketch with the “p” keyword.

`p.setup()`: special p5.js function that, without having to be called, will execute once when creating the sketch object. It initializes the canvas and sets up the platform, as well as the animation.
Called by: constructor function `sketch`.

`p.draw()`: special p5.js function that, without having to be called, will execute continuously to update the canvas (by default 60 times per second), when creating the sketch object.
Called by: constructor function `sketch`.

`createSVGImage(id, root, d, box)`: generates an SVG image dynamically and appends it to a designated HTML container. Its parameters are: the image's identifier (id), the container to append it to (root), the SVG path data (d), and the bounding box (box) information.
Called by: `setupPlatform`.

Event functions:
`svg.onclick()`: when this SVG element is clicked, the specified function will be executed.

`document.getElementById(“canvas”).onmousedown(ev)`: when the mouse button is pressed (mousedown event) on the element with the id "`canvas`", it records

the initial mouse position relative to the canvas (`ev.pageX` and `ev.pageY`) and sets `rotation.active` to true.
`document.onmouseup()`: when the mouse button is released (mouseup event), it sets `rotation.active` to false.

`document.onmousemove(ev)`: while the mouse is being moved, if `rotation.active` is true, it updates the rotation's center (`rotation.x` and `rotation.y`) based on the mouse movement.

`document.onkeydown(e)`: When any key is pressed (keydown event), it triggers the `document.onkeydown` function. If the pressed key is the spacebar (key code 32), it toggles the visibility of a path using `animation.toggleVisiblePath()`. If any other key is pressed, it triggers an animation based on the pressed key using `animation.start(String.fromCharCode(e.keyCode | 32))`.

`window.onload`: the `setupPlatform` function is assigned to `window.onload`, so it will execute once the entire webpage, including external scripts and images, has finished loading.

1.1.2 Main script overview (stewart.js)

Declared functions and methods' structure:

Anonymous function(root)

- getHexPlate(r_i, r_o, rot)
- parseSVGPath(str)
- Animation(platform)
    - SVG(svg, box)
        - move(x, y, z)
    - Interpolate(data)
    - toggleVisiblePath()
    - drawPath(p)
    - start(t)
    - _start(play, next)
    - moveTo(nt, no, time, next)
    - update(p)
    - this.cur.fn(pct)
- Stewart()

- `init(opts)`
- `initCircular/initHexagonal(opts)`
  - `getLegs()`
  - `drawBasePlate(p)`
  - `drawPlatformPlate(p)`
- `draw()`
  - `drawCone(p, radius, h)`
  - `drawFrame(p)`
- `update(translation, orientation)`
- `getServoAngles()`

Declared functions explained:

`Anonymous function(root)`: An anonymous function that takes a parameter named "root" and executes itself: IIFE (immeditaely invoked function expressions). An IIFE creates a private scope for the function's code and any variables it declares. This prevents them from unintentionally affecting or conflicting with variables and functions in the global namespace (usually the window object in browsers).
Called by: itself.

`getHexPlate(r_i, r_o, rot)`: Get the vertices of the "hexagonal" plates, used for both the base and the platform. We have 3 arguments: inner radious, outer radious and the rotation of the plate.
Called by: `Stewart.initHexagonal`, `Stewart.initCircular`.

`parseSVGPath(str)`: Parse an SVG path string and extract its individual segments along with their parameters. The function processes the path string character by character, identifying commands (e.g., M, L, C, Q, A) and their associated parameters.
Called by: `Animation.SVG`.

`Animation(platform)`: Constructor function for the Animation object that has the platform object as argument. Constructors are used to create and initialize an object instance of a class.
Called by: `p.setup` function on main HTML script.

`Animation.SVG(svg, box)`: The purpose of this function is to take an SVG path string and convert it into a series of 3D animation steps. The animation steps are then

returned for further use. Check `parseSVGPath()` at line 44 for parsing SVG path to individual segments for animation.
Called by: clicking at an SVG image displayed on screen, through the onclick event located in the `createSVGImage` function in the main html script.

`Animation.SVG.move(x, y, z)`: This function calculates the relative position of the given coordinates within the bounding box and adds an animation step to the `ret` array. It takes as parameters the desired next position as xyz coordinates.
Called by: `Animation.SVG`.

`Animation.Interpolate(data)`: It creates the "normalized" animation type object that needs to be passed as an argument to the `_start` function. This takes as argument the array that stores the animation steps, created by the `Animation.SVG` function.
Called by: `Animation.SVG`

`Animation.toggleVisiblePath()`: Toggles visibility of the animation path.
Called by: pressing spacebar on the keyboard, through `document.onkeydown` function in main HTML script.

`Animation.drawPath(p)`: Draws a red line from the origin of the platform throughout the animation path if `pathVisible` variable is set to true both onscreen and on the actual animation. Takes the sketch `p` as argument, needed to perform drawing operations on that sketch.
Called by: `p.draw` function on main HTML script.

`Animation.start(t)`: It takes as parameter `t` the type of animation: could be 'wobble', 'tilt', etc. Its purpose is to initialize the animation, only for the predefined animations. This function executes to initialize all predefined animations. SVG's animations directly use the `_start` function.
Called by: instantiating `new Animation` object, `Animation.update` and when pressing a key (not spacebar) using the `document.onkeydown` function of the html file script.

`Animation._start(play, next)`: sets necessary parameters for the execution of the animation. It takes in two parameters: the object containing the info about the animation to start, and the name (as string) of the next animation.
Called by: `Animation.start` and when clicking an svg image, through the `svg.onclick` event.

`Animation.moveTo(nt, no, time, next)`

`Animation.update(p)`: This function updates the platform position of current animation. Calculating the elapsed time and applying necessary changes depending on it.
Called by: `p.draw` function on main HTML script.

`Animation.this.cur.fn(pct)`: retrieves the data of the translation and rotation movements needed to perform the animation. Its argument is `pct`, which stands for percentage of animation completion (0 to 1).
Called by: `Animation.update`, `Animation.drawpath`.

`Stewart()`: Constructor function for the stewart platform object.
Called by: `p.setup` function on main HTML script.

`Stewart.init(opts)`: This function responsible for initializing the Stewart platform with the provided options. It initializes parameters such as rod length, horn length, base and platform geometry, servo range, etc. It calculates initial offsets based on the provided options.
Called by: `Stewart.initCircular/initHexagonal`

`Stewart.initCircular/initHexagonal(opts)`: This function initializes the Stewart platform with a circular/hexagonal configuration (depending on which function is called), defining the base and platform geometry along with other parameters. It also uses the `init` function to set up the common configuration for the Stewart platform.
Called by: `p.setup()` function on main HTML script.

`Stewart.initCircular/initHexagonal.getLegs()`: it's responsible for computing the configuration of each leg based on provided geometric parameters. It iterates through each leg, determining midpoints and directions to calculate base and

platform joint positions. Additionally, it adjusts the platform index order if necessary for proper alignment. The function constructs leg objects encapsulating these configurations. Ultimately, it returns an array containing the configuration of all legs.
Called by: `Stewart.init`.

`Stewart.initCircular/initHexagonal.drawBasePlate(p)`: Called periodically, draw the plate of the base using the base vertices retrieved with `getHexPlate`.
Called by: `Stewart.draw`

`Stewart.initCircular/Hexagonal.drawPlatformPlate(p)`: Called periodically, draw the plate of the platform using the platform vertices retrieved with `getHexPlate`.
Called by: `Stewart.draw`

`Stewart.draw()`: Draw the coordinate system axes and also draw the base plate and the platform plate, as well as all other objects: legs, base joints, platform joints.
Called by: `p.draw()` function on main HTML script.

`Stewart.draw.drawCone(p, radius, h)`: designed to render a cone shape using a given rendering context for the sketch (p), radius, and height (h).
Called by: `Stewart.draw.drawFrame`

`Stewart.draw.drawFrame(p)`: draws the coordinate system axes, with its lines and respective cones as arrows.
Called by: `Stewart.draw`

`Stewart.update(translation, orientation)`: Updates the position of the elements in the system, based on the desired translation of the platform origin and orientation of the platform calculated by the animation object. All the calculations used in this function are the ones described in Robert Eisele's paper on inverse kinematics of a Stewart Platform.
Called by: `Animation.update`.

`Stewart.getServoAngles()`: While not necessary for the execution of the visual representation, this function is strictly essential if we want to work with a real prototype,

since the only variable we are going to enter to the servos is the value of their rotation angles. This calculates each of the angles the servos have to rotate to accomplish a specific position of the horn edge (`H`), using basic trigonometric functions described in the paper.
Called by: this function is not explicitly used in the code.

1.1.3 Code explanation:

General procedure

Initially, a comprehensive overview of the code will be provided, offering a general explanation. Subsequently, focus will be given to key functions, referenced with square brackets as follows[x.y], providing detailed explanations following the initial overview.

The respective line of code where the key functions are executed are also provided. Follow with files `stewart-commented.js` and `default-commented.html`.

1. Since all the code in the HTML script depends on the `setupPlatform` function, nothing will happen until the `window.load` event is triggered. When it is, `setupPlatform` will execute and start the rest of the code.
2. A new instance of the `Sketch` object is created and attached to `canvas` element. With the creation of this object, function `p.setup` will be executed.
    a. In the `p.setup` function, the 3D canvas will be initialized, as well as the camera, and new instances of `Stewart` and `Animation`[1] objects will be created, then the `initHexagonal`[2] method of the `Stewart` object (`platform`) is called to initially draw the platform in a hexagonal disposition with default values.
    b. The `p.draw` function will be executed continuously right after `p.setup`:
        i. Clears the background with `p.background`.
        ii. Applies transformations for user interaction using `p.push`, `p.translate`, `p.rotateX`, and `p.rotateY` (only when event functions trigger them).
        iii. Draws motion paths if active, with the `drawPath`[3] method from the Animation object.

iv. Updates animation with the `update`[4] method from the `Animation` object (re-calculates new platform position and orientation).
v. Draws the platform with the `draw`[5] method from the `Stewart` object (re-draws platform, legs and other components).

3. For the SVG's visualization, the following procedure takes place:
    a. A variable named `$images` is declared and set to the `images` element in the HTML boilerplate.
    b. The `SVGS` array is created, which will contain a list of objects, and each object will represent one SVG image, with its path and box dimensions.
    c. A for loop will loop through all the objects in the `SVGS` array, executing the function `createSVGImage`[6] (previously declared in our HTML script) for each of the SVG object.

This is what will take place when executing the HTML script. Nonetheless, there are event functions (described in previous section) that will execute when an action takes place, such as clicking an element with the mouse or pressing a certain key.

[1] When `Animation` object is created (line 54 HTML file):

1. Assign own object variables to parsed argument (`platform`) and initialize translation array.
2. Execute `start`[1.1] function from Animation object with argument 'wobble' to start the default animation.
3. The Animation prototype object is also initialized, declaring `Animation` object's internal functions, and setting object variables (`cur`, `next`, `startTime`, `platform`, `translation`, `orientation`, `pathVisible`)

[1.1] When `Animation.start` function is executed (line 345 JavaScript file):

1. Checks if the `t` parameter is within the defined animations with the `fn` object, that contains all the defined animations.
    a. If the passed parameter is not inside the `fn` object, then it console logs "Failed" and ends the function, since the passed parameter is wrong.
    b. If the `t` parameter is within the `fn` object (defined animation), then it executes the `_start`[1.2] function to start with the animation, passing as parameters the object related to the corresponding animation and the string that represents the name of the next animation.

[1.2] When `Animation._start` function is executed (line 600 JavaScript file and line 132 HTML file):

1. (Checks if the play object has a start method in it. if it does, it calls it.) – Not needed to run the code since I have not seen any play object with a start method.
2. Sets current animation (`this.cur`) as the passed parameter play object, as well as `this.next`, which is the string name of the next animation.
3. Sets animation start time (`this.startTime`) to current date and time.

[2] When `platform.initHexagonal` method is executed (line 55 HTML file):

Note: the `platform.initCircular` method is exactly the same as `initHexagonal`, except it initializes a circular configuration, in contrast to hexagonal.

1. Check if the `opts` argument with the platform options was passed. If it wasn't, set `opts` to an empty object.
2. Define all the platform variables: base and platform dimensions, horn length and rod length, shaft and anchor distances, and other Boolean variables. Get hardcoded values if on opts arguments wasn't passed when calling the function.
3. Generate the vertices for the hexagonal base plate and platform plate, and assign them to variables `baseInts` and `platformInts`, respectively. The vertices are found using the `getHexPlate`[2.1] function.
4. Execute the `init`[2.2] function defined in the `Stewart.prototype`, passing as argument an anonymous object for hexagonal configuration. This object contains:
    a. The following variables value: `rodLength`, `hornLength`, `hornDirection`, `servoRange` and `servoRangeVisible`.
    b. The following functions: `getLegs`[2.3], `drawBasePlate`[5.2], `drawPlatformPlate`[5.3].

[2.1] When `platform.getHexPlate` is executed (lines 1064, 1065 JavaScript file):

1. Initializes an empty array to store the vertices of the hexagon, `ret`, standing for return array.
2. Calculates the distance from the centre to the midpoint of each side, depending on `r_i` and `r_o` (inner radius and outer radius arguments) and stores it to variable `a_2` (apothem).
3. Loops six times to calculate the coordinates of each vertex, defining an angle `phi` and pushing resulting vertex as an object with x and y to the `ret` array.

4. Returns `ret` array with the coordinates of all the six vertices.

[2.2] When `platform.init` is executed (lines 999, 1072 JavaScript file):

1. Set this platform variables to passed object variables.
2. Initialize as empty arrays `B`, `P`, `sinBeta`, `cosBeta`, vectors `q`, `l` and points `H`. Each of these variables are a six-element array, storing values for each of the legs. These variables represent:
    a. `B`: base joints in base frame
    b. `P`: platform joints in platform frame
    c. `q`: vector from base origin to `P`
    d. `l`: vector from `B` to `P`
    e. `H`: servo horn end to mount the rod
    f. `sinBeta`: sin of pan angle of motors in base plate
    g. `cosBeta`: cos of pan angle of motors in base plate
3. Setup legs configuration:
    a. Declare legs variable and store the legs configuration by calling the function `getLegs`[2.3].
    b. Loop through all the elements on the legs array and push corresponding values to the previously initialized variables (`B`, `p`, `q`, `l`, `H`, `sinBeta`, `cosBeta`).
4. Set the initial offset `T0`, based on whether absolute height is used or not. This initial offset oversees starting all the animations with the legs (horn + rod) forming a 90-degree angle, so that they have the same range of motion upwards and downwards.

[2.3] When `platform.initHexagonal.getLegs` is executed (line 953 JavaScript file):

1. Declare necessary variables and initialize as empty arrays: `legs`, `basePoints`, `platPoints` and `motorAngle`.
2. Loop six times and perform necessary calculations to get the location of each leg through arrays.
3. Finally store objects containing calculated `basePoints`, `platPoints` and `motorAngle` (arrays of 6 elements), to the array legs.

[3] When `animation.drawPath` method is executed (line 70 HTML file):

1. Checks if path visibility is off. If so, do nothing and `return` (do not draw path).
2. Draw the shape:

a. Set `noFill`, indicating that the shape will have transparent fill, and stroke to red.
b. Declare variable `steps` for number of vertices of the shape. The more vertices, the higher the fidelity of the shape, but it will also take more processing power.
c. Loop through each step:
    i. Call the fn function inside the current animation object and pass as argument `i / steps`, which represents the progress ratio of the animation (0 to 1). This sets a value for `this.translation`.
    ii. Create a vertex at the position of `this.translation`.

[4] When `animation.update` method is executed (line 73 HTML file):

1. Declare variable `now`, and set to current date and time, as well as variable `elapsed`, and set to the percentage of completion of the animation, using `startTime` and `now` to calculate it: 0 to 1, being 1 100% completed.
2. If `elapsed` is greater than 1, set it back to 1.
3. Call `fn` function inside `animation` object to update `this.translation` and `this.orientation`, passing as argument the `elapsed` variable.
4. If the animation is completed and there is a next animation, then start the next animation.
5. Call `this.platform.update`[4.1] function to update the position of the platform.

[4.1] When `platform.update` method is executed (line 666 JavaScript file):

1. Simplify variables that are going to be used and contain this.
2. Set platform's translation and orientation to corresponding passed parameters.
3. For each of the six legs, apply the math described in Robert Eisele's paper and do the following:
    a. Perform a quaternion rotation based on orientation to vector pk and store it in declared variable `o`.
    b. Simplify variable names of `l`, `q`, `H` and `B` so that they can be referenced faster. These variables are arrays of size 6 that contain three-dimensional arrays as their elements.
    c. Calculate vector from base origin to Pk (platform joint), using formula in paper: qk = T + R x pk x R (conjugate). Store it in variable `q`.

d. Vector from B to P (from base anchor to platform anchor), using formula in paper: lk = qk - bk = T + R x pk x R(conjugate) – bk. Store it in variable `l`.
e. Calculate parameters from the trigonometric identity. In the paper, d = `rodLength` and h = `hornLength`, and apply last formulas.
f. Finally, calculate endpoint of servo horn that intersects with servo rod (`H`).

[5] When `platform.draw` method is executed (line 76 HTML file):

1. Draw the axis coordinate of the base frame, using `drawFrame`[5.1] function and the base plate, using the `drawBasePlate`[5.2] function.
2. Translate the coordinate origin to the platform origin position, and after that apply a rotation matrix to rotate the plane according to its rotation quaternion. With these transformations made to the coordinate system, draw the platform plate using the `drawPlatformPlate`[5.3] function, as well as the axis coordinate of the platform, using `drawFrame`[5.1].
3. Finally, draw the base joints, platform joints, rods and horns, accessing their corresponding locations with the previously calculated arrays `B`, `q`, `H` and `B`.

[5.1] When `platform.draw.drawFrame` is executed (lines 1239, 1253 JavaScript file):

1. Declares variables that store the size of the arrows.
2. Draw the corresponding lines in three different colours.
3. Using the `drawCone` function, draw the cones, next to each of the lines, that make up the arrows.

[5.2] When `platform.drawBasePlate` is executed (line 1243 JavaScript file):

1. Sets the stroke colour to black, the fill colour to a tonality of yellow and draws a shape from vertices, looping through the length of `baseInts` and drawing a vertex in each of them.

[5.3] When `platform.drawPlatformPlate` is executed (line 1250 JavaScript file):

1. Sets the stroke colour to black, the fill colour to cyan and draws a shape from vertices, looping through the length of `PlatformInts` and drawing a vertex in each of them.

[6] When `createSVGImage` method is executed (line 178 HTML file):

1. Declare `xmlns` variable, which is a string containing the SVG namespace (necessary to display svg images)
2. Declare `svg` variable, which creates an element of type “svg”, containing the `xmlns` namespace created. Set attributes to this element: viewBox, width and height.
3. Add an `onclick` event to the svg, such as when it’s clicked, it calls the `_start`[1.2] function. It passes in two arguments: (`play`, `next`). The play argument is the object that contains the info about the animation to play. It is created using the `Animation.SVG`[6.1] function, and its arguments are the current SVG path (as string) and the viewBox, as object. As there is no next animation, the second argument is null.
4. Create the `path` element with its respective attributes and append it to the svg element.

[6.1] When `Animation.SVG` is executed (line 132 HTML file):

1. Declare constants to control animation speed, lower and higher value of the z coordinate as well as drawing size (default 80x80) (`PERSEC`, `L`, `H`, `SCREEN_SIZE`).
2. Declare variable `cur`, that represents current position within the SVG path, and it’s initialized to the centre of the provided bounding box (`box`). Also declare variable `ret` and initializes as empty array, which is the array used to store the animation steps.
3. Assign the returned steps from `parseSVGPath`[6.2], passing svg path as argument, to variable `seg`.
4. Loop through all segments on `seg` and use the `move`[6.3] function in different ways (depending on command name, with a switch statement), passing x, y and z coordinates as arguments to calculate their relative positions within the bounding box, and to add an animation step to the ret array.
5. Once the loop went through all the segments and the `ret` array is completed, the function returns the interpolated `ret` array using `Animation.Interpolate(ret)` [6.4].

[6.2] When `parseSVGPath` function is executed (line 388 JavaScript file):

1. Convert passed string containing all the commands (svg path), into an array where every command or number becomes an element of the array, and sets it to variable `p`. For example: if `str` is "A, -3, B, c, 0.23", `p` will be [A, -3, B, c, 0.23].

2. Declare necessary constants to run the function: string containing all possible commands (`COMMANDS`), string with only uppercase commands (`UPPERCASE`), as well as the following variables, which are initialized:
    a. `segments`: array that will be returned in function, containing in the end all the steps of the animation.
    b. `cur`: object that is used to refer to the current position of the svg drawing.
    c. `start`: object that contains the starting position of the svg drawing.
    d. `cmd`: string that represents current command to parse.
    e. `prevCmd`: string that represents previously parsed command.
    f. `isRelative`: Boolean value that determines if a command is relative or not, based on if it's uppercase or lowercase.
3. While the length of `p` array is greater than zero, do the following:
    a. Check if the first element of the `p` array is found within the `COMMANDS` string.
        i. If it is, find previous command and assign it to `prevCmd`, remove first element of `p` array (new command) and assign it to current command `cmd`. Then, if the command is lowercase, it sets `isRelative` to `True`.
        ii. If it's not and the current command is inexistent, throw an error since the given command is invalid, else, find previous command and assign it to `prevCmd`.
    b. Now that we know the command and whether it's relative or not, introduce a switch statement that takes different actions depending on the type of command. The behaviour will depend on every command, but it will normally do the following:
        i. Declare necessary x and y variables and attach them to their respective values (retrieving them from the `p` array).
        ii. Check if the command is relative or not, and set `cur.x` and `cur.y` as relative or absolute value, respectively.
        iii. Make necessary transformations based on command and push `cur.x` and `cur.y` to the `segments` array.
4. Return the `segments` array.

[6.3] When `Animation.SVG.move` is executed (line 399 to 479 JavaScript file):

1. Declare `relX` and `relY` variables and assign them the desired position (with `x` and `y` arguments) relative to the bounding box. Then scale it to the screen size, and finally centre it.
2. Do the same but with the current position (using cur), instead of desired position, and with variable names `relCurX` and `relCurY`.
3. Push the desired position to `ret` array, as well as origin command and animation time (which is calculated subtracting desired position by current position, then dividing by speed. (distance / distance/time = time).
4. Finally set new current position (`cur`) to desired position passed arguments `x`, `y` and `z`.

[6.4] When `Animation.Interpolate` function is executed (line 485 JavaScript file):

1. Creates a `duration` variable and calculates is value, adding all the durations of the whole animation steps together.
2. Return an object containing all the necessary parameters to perform the animation:
    a. `duration`, `pathVisible` and `next`
    b. `fn` function:
        i. Doesn't modify orientation (sets it to `Quaternion.ONE`)
        ii. Declares `pctStart` and initializes to zero, to represent starting progress of the animation. (From 0 to 1).
        iii. Loop through every step of the animation (`data` array), and do the following:
            1. Declare variable `p`, which equals current step of the animation, as well as `pctEnd`, which equals the percentage of animation elapsed until current step's end.
            2. If statement that will only execute the code inside it if the elapsed animation percentage (`pct`) is within `pctStart` and `pctEnd`. It will calculate how far the current animation is in selected step with the `scale` variable, and set `this.translation` value for `x`, `y` and `z`, having in account previous step position and current step offset, multiplied by the `scale` coefficient.
            3. Set pctStart equal to pctEnd to continue with the loop.
        iv. Set `this.translation` to last element in `data` array, since this one isn't considered inside the loop or if statement.

Documented by: Albert Castellanos Roig

Last modified: 03/06/2024