[ScrollMagic](http://docs.google.com/index.html)

* [Classes](http://docs.google.com/classes.list.html)
  + [Controller](http://docs.google.com/ScrollMagic.Controller.html)
  + [Scene](http://docs.google.com/ScrollMagic.Scene.html)
* [Events](http://docs.google.com/events.list.html)
  + [add](http://docs.google.com/ScrollMagic.Scene.html#event:add)
  + [change](http://docs.google.com/ScrollMagic.Scene.html#event:change)
  + [destroy](http://docs.google.com/ScrollMagic.Scene.html#event:destroy)
  + [end](http://docs.google.com/ScrollMagic.Scene.html#event:end)
  + [enter](http://docs.google.com/ScrollMagic.Scene.html#event:enter)
  + [leave](http://docs.google.com/ScrollMagic.Scene.html#event:leave)
  + [progress](http://docs.google.com/ScrollMagic.Scene.html#event:progress)
  + [remove](http://docs.google.com/ScrollMagic.Scene.html#event:remove)
  + [shift](http://docs.google.com/ScrollMagic.Scene.html#event:shift)
  + [start](http://docs.google.com/ScrollMagic.Scene.html#event:start)
  + [update](http://docs.google.com/ScrollMagic.Scene.html#event:update)
* [Plugins](http://docs.google.com/mixins.list.html)
  + [GSAP](http://docs.google.com/animation.GSAP.html)
  + [Velocity](http://docs.google.com/animation.Velocity.html)
  + [addIndicators](http://docs.google.com/debug.addIndicators.html)
  + [jQuery](http://docs.google.com/framework.jQuery.html)

Source: ScrollMagic/Controller.js

/\*\*  
 \* The main class that is needed once per scroll container.  
 \*  
 \* @class  
 \*  
 \* @example  
 \* // basic initialization  
 \* var controller = new ScrollMagic.Controller();  
 \*  
 \* // passing options  
 \* var controller = new ScrollMagic.Controller({container: "#myContainer", loglevel: 3});  
 \*  
 \* @param {object} [options] - An object containing one or more options for the controller.  
 \* @param {(string|object)} [options.container=window] - A selector, DOM object that references the main container for scrolling.  
 \* @param {boolean} [options.vertical=true] - Sets the scroll mode to vertical (`true`) or horizontal (`false`) scrolling.  
 \* @param {object} [options.globalSceneOptions={}] - These options will be passed to every Scene that is added to the controller using the addScene method. For more information on Scene options see {@link ScrollMagic.Scene}.  
 \* @param {number} [options.loglevel=2] Loglevel for debugging. Note that logging is disabled in the minified version of ScrollMagic.  
 \*\* `0` => silent  
 \*\* `1` => errors  
 \*\* `2` => errors, warnings  
 \*\* `3` => errors, warnings, debuginfo  
 \* @param {boolean} [options.refreshInterval=100] - Some changes don't call events by default, like changing the container size or moving a scene trigger element.   
 This interval polls these parameters to fire the necessary events.   
 If you don't use custom containers, trigger elements or have static layouts, where the positions of the trigger elements don't change, you can set this to 0 disable interval checking and improve performance.  
 \*  
 \*/  
ScrollMagic.Controller = function(options) {  
 /\*  
 \* ----------------------------------------------------------------  
 \* settings  
 \* ----------------------------------------------------------------  
 \*/  
 var  
 NAMESPACE = 'ScrollMagic.Controller',  
 SCROLL\_DIRECTION\_FORWARD = 'FORWARD',  
 SCROLL\_DIRECTION\_REVERSE = 'REVERSE',  
 SCROLL\_DIRECTION\_PAUSED = 'PAUSED',  
 DEFAULT\_OPTIONS = CONTROLLER\_OPTIONS.defaults;  
  
 /\*  
 \* ----------------------------------------------------------------  
 \* private vars  
 \* ----------------------------------------------------------------  
 \*/  
 var  
 Controller = this,  
 \_options = \_util.extend({}, DEFAULT\_OPTIONS, options),  
 \_sceneObjects = [],  
 \_updateScenesOnNextCycle = false, // can be boolean (true => all scenes) or an array of scenes to be updated  
 \_scrollPos = 0,  
 \_scrollDirection = SCROLL\_DIRECTION\_PAUSED,  
 \_isDocument = true,  
 \_viewPortSize = 0,  
 \_enabled = true,  
 \_updateTimeout,  
 \_refreshTimeout;  
  
 /\*  
 \* ----------------------------------------------------------------  
 \* private functions  
 \* ----------------------------------------------------------------  
 \*/  
  
 /\*\*  
 \* Internal constructor function of the ScrollMagic Controller  
 \* @private  
 \*/  
 var construct = function () {  
 for (var key in \_options) {  
 if (!DEFAULT\_OPTIONS.hasOwnProperty(key)) {  
 log(2, "WARNING: Unknown option \"" + key + "\"");  
 delete \_options[key];  
 }  
 }  
 \_options.container = \_util.get.elements(\_options.container)[0];  
 // check ScrollContainer  
 if (!\_options.container) {  
 log(1, "ERROR creating object " + NAMESPACE + ": No valid scroll container supplied");  
 throw NAMESPACE + " init failed."; // cancel  
 }  
 \_isDocument = \_options.container === window || \_options.container === document.body || !document.body.contains(\_options.container);  
 // normalize to window  
 if (\_isDocument) {  
 \_options.container = window;  
 }  
 // update container size immediately  
 \_viewPortSize = getViewportSize();  
 // set event handlers  
 \_options.container.addEventListener("resize", onChange);  
 \_options.container.addEventListener("scroll", onChange);  
  
 var ri = parseInt(\_options.refreshInterval, 10);  
 \_options.refreshInterval = \_util.type.Number(ri) ? ri : DEFAULT\_OPTIONS.refreshInterval;  
 scheduleRefresh();  
  
 log(3, "added new " + NAMESPACE + " controller (v" + ScrollMagic.version + ")");  
 };  
  
 /\*\*  
 \* Schedule the next execution of the refresh function  
 \* @private  
 \*/  
 var scheduleRefresh = function () {  
 if (\_options.refreshInterval > 0) {  
 \_refreshTimeout = window.setTimeout(refresh, \_options.refreshInterval);  
 }  
 };  
  
 /\*\*  
 \* Default function to get scroll pos - overwriteable using `Controller.scrollPos(newFunction)`  
 \* @private  
 \*/  
 var getScrollPos = function () {  
 return \_options.vertical ? \_util.get.scrollTop(\_options.container) : \_util.get.scrollLeft(\_options.container);  
 };  
  
 /\*\*  
 \* Returns the current viewport Size (width vor horizontal, height for vertical)  
 \* @private  
 \*/  
 var getViewportSize = function () {  
 return \_options.vertical ? \_util.get.height(\_options.container) : \_util.get.width(\_options.container);  
 };  
  
 /\*\*  
 \* Default function to set scroll pos - overwriteable using `Controller.scrollTo(newFunction)`  
 \* Make available publicly for pinned mousewheel workaround.  
 \* @private  
 \*/  
 var setScrollPos = this.\_setScrollPos = function (pos) {  
 if (\_options.vertical) {  
 if (\_isDocument) {  
 window.scrollTo(\_util.get.scrollLeft(), pos);  
 } else {  
 \_options.container.scrollTop = pos;  
 }  
 } else {  
 if (\_isDocument) {  
 window.scrollTo(pos, \_util.get.scrollTop());  
 } else {  
 \_options.container.scrollLeft = pos;  
 }  
 }  
 };  
  
 /\*\*  
 \* Handle updates in cycles instead of on scroll (performance)  
 \* @private  
 \*/  
 var updateScenes = function () {  
 if (\_enabled && \_updateScenesOnNextCycle) {  
 // determine scenes to update  
 var scenesToUpdate = \_util.type.Array(\_updateScenesOnNextCycle) ? \_updateScenesOnNextCycle : \_sceneObjects.slice(0);  
 // reset scenes  
 \_updateScenesOnNextCycle = false;  
 var oldScrollPos = \_scrollPos;  
 // update scroll pos now instead of onChange, as it might have changed since scheduling (i.e. in-browser smooth scroll)  
 \_scrollPos = Controller.scrollPos();  
 var deltaScroll = \_scrollPos - oldScrollPos;  
 if (deltaScroll !== 0) { // scroll position changed?  
 \_scrollDirection = (deltaScroll > 0) ? SCROLL\_DIRECTION\_FORWARD : SCROLL\_DIRECTION\_REVERSE;  
 }  
 // reverse order of scenes if scrolling reverse  
 if (\_scrollDirection === SCROLL\_DIRECTION\_REVERSE) {  
 scenesToUpdate.reverse();  
 }  
 // update scenes  
 scenesToUpdate.forEach(function (scene, index) {  
 log(3, "updating Scene " + (index + 1) + "/" + scenesToUpdate.length + " (" + \_sceneObjects.length + " total)");  
 scene.update(true);  
 });  
 // (BUILD) - REMOVE IN MINIFY - START  
 if (scenesToUpdate.length === 0 && \_options.loglevel >= 3) {  
 log(3, "updating 0 Scenes (nothing added to controller)");  
 }  
 // (BUILD) - REMOVE IN MINIFY - END  
 }  
 };  
   
 /\*\*  
 \* Initializes rAF callback  
 \* @private  
 \*/  
 var debounceUpdate = function () {  
 \_updateTimeout = \_util.rAF(updateScenes);  
 };  
   
 /\*\*  
 \* Handles Container changes  
 \* @private  
 \*/  
 var onChange = function (e) {  
 log(3, "event fired causing an update:", e.type);  
 if (e.type == "resize") {  
 // resize  
 \_viewPortSize = getViewportSize();  
 \_scrollDirection = SCROLL\_DIRECTION\_PAUSED;  
 }  
 // schedule update  
 if (\_updateScenesOnNextCycle !== true) {  
 \_updateScenesOnNextCycle = true;  
 debounceUpdate();  
 }  
 };  
  
 var refresh = function () {  
 if (!\_isDocument) {  
 // simulate resize event. Only works for viewport relevant param (performance)  
 if (\_viewPortSize != getViewportSize()) {  
 var resizeEvent;  
 try {  
 resizeEvent = new Event('resize', {bubbles: false, cancelable: false});  
 } catch (e) { // stupid IE  
 resizeEvent = document.createEvent("Event");  
 resizeEvent.initEvent("resize", false, false);  
 }  
 \_options.container.dispatchEvent(resizeEvent);  
 }  
 }  
 \_sceneObjects.forEach(function (scene, index) {// refresh all scenes  
 scene.refresh();  
 });  
 scheduleRefresh();  
 };  
  
 // (BUILD) - REMOVE IN MINIFY - START  
 /\*\*  
 \* Send a debug message to the console.  
 \* provided publicly with \_log for plugins  
 \* @private  
 \*  
 \* @param {number} loglevel - The loglevel required to initiate output for the message.  
 \* @param {...mixed} output - One or more variables that should be passed to the console.  
 \*/  
 var log = this.\_log = function (loglevel, output) {  
 if (\_options.loglevel >= loglevel) {  
 Array.prototype.splice.call(arguments, 1, 0, "(" + NAMESPACE + ") ->");  
 \_util.log.apply(window, arguments);  
 }  
 };  
 // (BUILD) - REMOVE IN MINIFY - END  
 // for scenes we have getters for each option, but for the controller we don't, so we need to make it available externally for plugins  
 this.\_options = \_options;  
  
 /\*\*  
 \* Sort scenes in ascending order of their start offset.  
 \* @private  
 \*  
 \* @param {array} ScenesArray - an array of ScrollMagic Scenes that should be sorted  
 \* @return {array} The sorted array of Scenes.  
 \*/  
 var sortScenes = function (ScenesArray) {  
 if (ScenesArray.length <= 1) {  
 return ScenesArray;  
 } else {  
 var scenes = ScenesArray.slice(0);  
 scenes.sort(function(a, b) {  
 return a.scrollOffset() > b.scrollOffset() ? 1 : -1;  
 });  
 return scenes;  
 }  
 };  
  
 /\*\*  
 \* ----------------------------------------------------------------  
 \* public functions  
 \* ----------------------------------------------------------------  
 \*/  
  
 /\*\*  
 \* Add one ore more scene(s) to the controller.   
 \* This is the equivalent to `Scene.addTo(controller)`.  
 \* @public  
 \* @example  
 \* // with a previously defined scene  
 \* controller.addScene(scene);  
 \*  
 \* // with a newly created scene.  
 \* controller.addScene(new ScrollMagic.Scene({duration : 0}));  
 \*  
 \* // adding multiple scenes  
 \* controller.addScene([scene, scene2, new ScrollMagic.Scene({duration : 0})]);  
 \*  
 \* @param {(ScrollMagic.Scene|array)} newScene - ScrollMagic Scene or Array of Scenes to be added to the controller.  
 \* @return {Controller} Parent object for chaining.  
 \*/  
 this.addScene = function (newScene) {  
 if (\_util.type.Array(newScene)) {  
 newScene.forEach(function (scene, index) {  
 Controller.addScene(scene);  
 });  
 } else if (newScene instanceof ScrollMagic.Scene) {  
 if (newScene.controller() !== Controller) {  
 newScene.addTo(Controller);  
 } else if (\_sceneObjects.indexOf(newScene) < 0){  
 // new scene  
 \_sceneObjects.push(newScene); // add to array  
 \_sceneObjects = sortScenes(\_sceneObjects); // sort  
 newScene.on("shift.controller\_sort", function() { // resort whenever scene moves  
 \_sceneObjects = sortScenes(\_sceneObjects);  
 });  
 // insert Global defaults.  
 for (var key in \_options.globalSceneOptions) {  
 if (newScene[key]) {  
 newScene[key].call(newScene, \_options.globalSceneOptions[key]);  
 }  
 }  
 log(3, "adding Scene (now " + \_sceneObjects.length + " total)");  
 }  
 } else {  
 log(1, "ERROR: invalid argument supplied for '.addScene()'");  
 }  
 return Controller;  
 };  
  
 /\*\*  
 \* Remove one ore more scene(s) from the controller.   
 \* This is the equivalent to `Scene.remove()`.  
 \* @public  
 \* @example  
 \* // remove a scene from the controller  
 \* controller.removeScene(scene);  
 \*  
 \* // remove multiple scenes from the controller  
 \* controller.removeScene([scene, scene2, scene3]);  
 \*  
 \* @param {(ScrollMagic.Scene|array)} Scene - ScrollMagic Scene or Array of Scenes to be removed from the controller.  
 \* @returns {Controller} Parent object for chaining.  
 \*/  
 this.removeScene = function (Scene) {  
 if (\_util.type.Array(Scene)) {  
 Scene.forEach(function (scene, index) {  
 Controller.removeScene(scene);  
 });  
 } else {  
 var index = \_sceneObjects.indexOf(Scene);  
 if (index > -1) {  
 Scene.off("shift.controller\_sort");  
 \_sceneObjects.splice(index, 1);  
 log(3, "removing Scene (now " + \_sceneObjects.length + " left)");  
 Scene.remove();  
 }  
 }  
 return Controller;  
 };  
  
 /\*\*  
 \* Update one ore more scene(s) according to the scroll position of the container.   
 \* This is the equivalent to `Scene.update()`.   
 \* The update method calculates the scene's start and end position (based on the trigger element, trigger hook, duration and offset) and checks it against the current scroll position of the container.   
 \* It then updates the current scene state accordingly (or does nothing, if the state is already correct) – Pins will be set to their correct position and tweens will be updated to their correct progress.   
 \* \_\*\*Note:\*\* This method gets called constantly whenever Controller detects a change. The only application for you is if you change something outside of the realm of ScrollMagic, like moving the trigger or changing tween parameters.\_  
 \* @public  
 \* @example  
 \* // update a specific scene on next cycle  
 \* controller.updateScene(scene);  
 \*  
 \* // update a specific scene immediately  
 \* controller.updateScene(scene, true);  
 \*  
 \* // update multiple scenes scene on next cycle  
 \* controller.updateScene([scene1, scene2, scene3]);  
 \*  
 \* @param {ScrollMagic.Scene} Scene - ScrollMagic Scene or Array of Scenes that is/are supposed to be updated.  
 \* @param {boolean} [immediately=false] - If `true` the update will be instant, if `false` it will wait until next update cycle.   
 This is useful when changing multiple properties of the scene - this way it will only be updated once all new properties are set (updateScenes).  
 \* @return {Controller} Parent object for chaining.  
 \*/  
 this.updateScene = function (Scene, immediately) {  
 if (\_util.type.Array(Scene)) {  
 Scene.forEach(function (scene, index) {  
 Controller.updateScene(scene, immediately);  
 });  
 } else {  
 if (immediately) {  
 Scene.update(true);  
 } else if (\_updateScenesOnNextCycle !== true && Scene instanceof ScrollMagic.Scene) { // if \_updateScenesOnNextCycle is true, all connected scenes are already scheduled for update  
 // prep array for next update cycle  
 \_updateScenesOnNextCycle = \_updateScenesOnNextCycle || [];  
 if (\_updateScenesOnNextCycle.indexOf(Scene) == -1) {  
 \_updateScenesOnNextCycle.push(Scene);   
 }  
 \_updateScenesOnNextCycle = sortScenes(\_updateScenesOnNextCycle); // sort  
 debounceUpdate();  
 }  
 }  
 return Controller;  
 };  
  
 /\*\*  
 \* Updates the controller params and calls updateScene on every scene, that is attached to the controller.   
 \* See `Controller.updateScene()` for more information about what this means.   
 \* In most cases you will not need this function, as it is called constantly, whenever ScrollMagic detects a state change event, like resize or scroll.   
 \* The only application for this method is when ScrollMagic fails to detect these events.   
 \* One application is with some external scroll libraries (like iScroll) that move an internal container to a negative offset instead of actually scrolling. In this case the update on the controller needs to be called whenever the child container's position changes.  
 \* For this case there will also be the need to provide a custom function to calculate the correct scroll position. See `Controller.scrollPos()` for details.  
 \* @public  
 \* @example  
 \* // update the controller on next cycle (saves performance due to elimination of redundant updates)  
 \* controller.update();  
 \*  
 \* // update the controller immediately  
 \* controller.update(true);  
 \*  
 \* @param {boolean} [immediately=false] - If `true` the update will be instant, if `false` it will wait until next update cycle (better performance)  
 \* @return {Controller} Parent object for chaining.  
 \*/  
 this.update = function (immediately) {  
 onChange({type: "resize"}); // will update size and set \_updateScenesOnNextCycle to true  
 if (immediately) {  
 updateScenes();  
 }  
 return Controller;  
 };  
  
 /\*\*  
 \* Scroll to a numeric scroll offset, a DOM element, the start of a scene or provide an alternate method for scrolling.   
 \* For vertical controllers it will change the top scroll offset and for horizontal applications it will change the left offset.  
 \* @public  
 \*  
 \* @since 1.1.0  
 \* @example  
 \* // scroll to an offset of 100  
 \* controller.scrollTo(100);  
 \*  
 \* // scroll to a DOM element  
 \* controller.scrollTo("#anchor");  
 \*  
 \* // scroll to the beginning of a scene  
 \* var scene = new ScrollMagic.Scene({offset: 200});  
 \* controller.scrollTo(scene);  
 \*  
 \* // define a new scroll position modification function (jQuery animate instead of jump)  
 \* controller.scrollTo(function (newScrollPos) {  
 \* $("html, body").animate({scrollTop: newScrollPos});  
 \* });  
 \* controller.scrollTo(100); // call as usual, but the new function will be used instead  
 \*  
 \* // define a new scroll function with an additional parameter  
 \* controller.scrollTo(function (newScrollPos, message) {  
 \* console.log(message);  
 \* $(this).animate({scrollTop: newScrollPos});  
 \* });  
 \* // call as usual, but supply an extra parameter to the defined custom function  
 \* controller.scrollTo(100, "my message");  
 \*  
 \* // define a new scroll function with an additional parameter containing multiple variables  
 \* controller.scrollTo(function (newScrollPos, options) {  
 \* someGlobalVar = options.a + options.b;  
 \* $(this).animate({scrollTop: newScrollPos});  
 \* });  
 \* // call as usual, but supply an extra parameter containing multiple options  
 \* controller.scrollTo(100, {a: 1, b: 2});  
 \*  
 \* // define a new scroll function with a callback supplied as an additional parameter  
 \* controller.scrollTo(function (newScrollPos, callback) {  
 \* $(this).animate({scrollTop: newScrollPos}, 400, "swing", callback);  
 \* });  
 \* // call as usual, but supply an extra parameter, which is used as a callback in the previously defined custom scroll function  
 \* controller.scrollTo(100, function() {  
 \* console.log("scroll has finished.");  
 \* });  
 \*  
 \* @param {mixed} scrollTarget - The supplied argument can be one of these types:  
 \* 1. `number` -> The container will scroll to this new scroll offset.  
 \* 2. `string` or `object` -> Can be a selector or a DOM object.   
 \* The container will scroll to the position of this element.  
 \* 3. `ScrollMagic Scene` -> The container will scroll to the start of this scene.  
 \* 4. `function` -> This function will be used for future scroll position modifications.   
 \* This provides a way for you to change the behaviour of scrolling and adding new behaviour like animation. The function receives the new scroll position as a parameter and a reference to the container element using `this`.   
 \* It may also optionally receive an optional additional parameter (see below)   
 \* \_\*\*NOTE:\*\*   
 \* All other options will still work as expected, using the new function to scroll.\_  
 \* @param {mixed} [additionalParameter] - If a custom scroll function was defined (see above 4.), you may want to supply additional parameters to it, when calling it. You can do this using this parameter – see examples for details. Please note, that this parameter will have no effect, if you use the default scrolling function.  
 \* @returns {Controller} Parent object for chaining.  
 \*/  
 this.scrollTo = function (scrollTarget, additionalParameter) {  
 if (\_util.type.Number(scrollTarget)) { // excecute  
 setScrollPos.call(\_options.container, scrollTarget, additionalParameter);  
 } else if (scrollTarget instanceof ScrollMagic.Scene) { // scroll to scene  
 if (scrollTarget.controller() === Controller) { // check if the controller is associated with this scene  
 Controller.scrollTo(scrollTarget.scrollOffset(), additionalParameter);  
 } else {  
 log (2, "scrollTo(): The supplied scene does not belong to this controller. Scroll cancelled.", scrollTarget);  
 }  
 } else if (\_util.type.Function(scrollTarget)) { // assign new scroll function  
 setScrollPos = scrollTarget;  
 } else { // scroll to element  
 var elem = \_util.get.elements(scrollTarget)[0];  
 if (elem) {  
 // if parent is pin spacer, use spacer position instead so correct start position is returned for pinned elements.  
 while (elem.parentNode.hasAttribute(PIN\_SPACER\_ATTRIBUTE)) {  
 elem = elem.parentNode;  
 }  
  
 var  
 param = \_options.vertical ? "top" : "left", // which param is of interest ?  
 containerOffset = \_util.get.offset(\_options.container), // container position is needed because element offset is returned in relation to document, not in relation to container.  
 elementOffset = \_util.get.offset(elem);  
  
 if (!\_isDocument) { // container is not the document root, so substract scroll Position to get correct trigger element position relative to scrollcontent  
 containerOffset[param] -= Controller.scrollPos();  
 }  
  
 Controller.scrollTo(elementOffset[param] - containerOffset[param], additionalParameter);  
 } else {  
 log (2, "scrollTo(): The supplied argument is invalid. Scroll cancelled.", scrollTarget);  
 }  
 }  
 return Controller;  
 };  
  
 /\*\*  
 \* \*\*Get\*\* the current scrollPosition or \*\*Set\*\* a new method to calculate it.   
 \* -> \*\*GET\*\*:  
 \* When used as a getter this function will return the current scroll position.   
 \* To get a cached value use Controller.info("scrollPos"), which will be updated in the update cycle.   
 \* For vertical controllers it will return the top scroll offset and for horizontal applications it will return the left offset.  
 \*  
 \* -> \*\*SET\*\*:  
 \* When used as a setter this method prodes a way to permanently overwrite the controller's scroll position calculation.   
 \* A typical usecase is when the scroll position is not reflected by the containers scrollTop or scrollLeft values, but for example by the inner offset of a child container.   
 \* Moving a child container inside a parent is a commonly used method for several scrolling frameworks, including iScroll.   
 \* By providing an alternate calculation function you can make sure ScrollMagic receives the correct scroll position.   
 \* Please also bear in mind that your function should return y values for vertical scrolls an x for horizontals.  
 \*  
 \* To change the current scroll position please use `Controller.scrollTo()`.  
 \* @public  
 \*  
 \* @example  
 \* // get the current scroll Position  
 \* var scrollPos = controller.scrollPos();  
 \*  
 \* // set a new scroll position calculation method  
 \* controller.scrollPos(function () {  
 \* return this.info("vertical") ? -mychildcontainer.y : -mychildcontainer.x  
 \* });  
 \*  
 \* @param {function} [scrollPosMethod] - The function to be used for the scroll position calculation of the container.  
 \* @returns {(number|Controller)} Current scroll position or parent object for chaining.  
 \*/  
 this.scrollPos = function (scrollPosMethod) {  
 if (!arguments.length) { // get  
 return getScrollPos.call(Controller);  
 } else { // set  
 if (\_util.type.Function(scrollPosMethod)) {  
 getScrollPos = scrollPosMethod;  
 } else {  
 log(2, "Provided value for method 'scrollPos' is not a function. To change the current scroll position use 'scrollTo()'.");  
 }  
 }  
 return Controller;  
 };  
  
 /\*\*  
 \* \*\*Get\*\* all infos or one in particular about the controller.  
 \* @public  
 \* @example  
 \* // returns the current scroll position (number)  
 \* var scrollPos = controller.info("scrollPos");  
 \*  
 \* // returns all infos as an object  
 \* var infos = controller.info();  
 \*  
 \* @param {string} [about] - If passed only this info will be returned instead of an object containing all.   
 Valid options are:  
 \*\* `"size"` => the current viewport size of the container  
 \*\* `"vertical"` => true if vertical scrolling, otherwise false  
 \*\* `"scrollPos"` => the current scroll position  
 \*\* `"scrollDirection"` => the last known direction of the scroll  
 \*\* `"container"` => the container element  
 \*\* `"isDocument"` => true if container element is the document.  
 \* @returns {(mixed|object)} The requested info(s).  
 \*/  
 this.info = function (about) {  
 var values = {  
 size: \_viewPortSize, // contains height or width (in regard to orientation);  
 vertical: \_options.vertical,  
 scrollPos: \_scrollPos,  
 scrollDirection: \_scrollDirection,  
 container: \_options.container,  
 isDocument: \_isDocument  
 };  
 if (!arguments.length) { // get all as an object  
 return values;  
 } else if (values[about] !== undefined) {  
 return values[about];  
 } else {  
 log(1, "ERROR: option \"" + about + "\" is not available");  
 return;  
 }  
 };  
  
 /\*\*  
 \* \*\*Get\*\* or \*\*Set\*\* the current loglevel option value.  
 \* @public  
 \*  
 \* @example  
 \* // get the current value  
 \* var loglevel = controller.loglevel();  
 \*  
 \* // set a new value  
 \* controller.loglevel(3);  
 \*  
 \* @param {number} [newLoglevel] - The new loglevel setting of the Controller. `[0-3]`  
 \* @returns {(number|Controller)} Current loglevel or parent object for chaining.  
 \*/  
 this.loglevel = function (newLoglevel) {  
 // (BUILD) - REMOVE IN MINIFY - START  
 if (!arguments.length) { // get  
 return \_options.loglevel;  
 } else if (\_options.loglevel != newLoglevel) { // set  
 \_options.loglevel = newLoglevel;  
 }  
 // (BUILD) - REMOVE IN MINIFY - END  
 return Controller;  
 };  
  
 /\*\*  
 \* \*\*Get\*\* or \*\*Set\*\* the current enabled state of the controller.   
 \* This can be used to disable all Scenes connected to the controller without destroying or removing them.  
 \* @public  
 \*  
 \* @example  
 \* // get the current value  
 \* var enabled = controller.enabled();  
 \*  
 \* // disable the controller  
 \* controller.enabled(false);  
 \*  
 \* @param {boolean} [newState] - The new enabled state of the controller `true` or `false`.  
 \* @returns {(boolean|Controller)} Current enabled state or parent object for chaining.  
 \*/  
 this.enabled = function (newState) {  
 if (!arguments.length) { // get  
 return \_enabled;  
 } else if (\_enabled != newState) { // set  
 \_enabled = !!newState;  
 Controller.updateScene(\_sceneObjects, true);  
 }  
 return Controller;  
 };  
   
 /\*\*  
 \* Destroy the Controller, all Scenes and everything.  
 \* @public  
 \*  
 \* @example  
 \* // without resetting the scenes  
 \* controller = controller.destroy();  
 \*  
 \* // with scene reset  
 \* controller = controller.destroy(true);  
 \*  
 \* @param {boolean} [resetScenes=false] - If `true` the pins and tweens (if existent) of all scenes will be reset.  
 \* @returns {null} Null to unset handler variables.  
 \*/  
 this.destroy = function (resetScenes) {  
 window.clearTimeout(\_refreshTimeout);  
 var i = \_sceneObjects.length;  
 while (i--) {  
 \_sceneObjects[i].destroy(resetScenes);  
 }  
 \_options.container.removeEventListener("resize", onChange);  
 \_options.container.removeEventListener("scroll", onChange);  
 \_util.cAF(\_updateTimeout);  
 log(3, "destroyed " + NAMESPACE + " (reset: " + (resetScenes ? "true" : "false") + ")");  
 return null;  
 };  
  
 // INIT  
 construct();  
 return Controller;  
};  
  
// @include('Controller/\_static.js')

© Jan Paepke 2015

Documentation generated by [JSDoc 3.5.5](https://github.com/jsdoc3/jsdoc) using a customized version of the [DocStrap template](https://github.com/terryweiss/docstrap).