DESCARTES

matemáticas interactivas

Descartes JS User Manual

Alejandro Radillo Díaz

José Luis Abreu León

Joel Espinosa Longi

November 21, 2023

Contents

1

About this document

This user manual is destined both for those who have not used Descartes to those who have some experience and wish to improve it.

The manual deals with the *DescartesJS* editor and its general functionality. The functionality is however, the main object of this document. The functionality here described is the most common in Descartes. Son particular details are not not dealt with since they are not commonly used.

The user manual includes exercises intended to make some concepts mor familiar to the reader. These exercises are made available once the reader has gathered some knowledge about Descartes and is considered ready to put it in practice. The solved exercises are provided in the form of Descartes interactive scenes, so that the reader can compare them with his own scenes. These provided solution exercises can be found in the *Exercises* folder included in the *DescartesDocumentation.zip* compressed file in https://descartes.matem.unam.mx/doc/DescartesJS-EN/DescartesJSDocumentation.zip. These interactive scenes are also included in the web. It is, however, important to know they can be downloaded from this compressed file should the reader wish to use them locally. It is also worth noticing that the links to the interactive scenes provided in this document redirect to the solved exercise in the web, which allows to view the finished version of the exercise, as well as a series of instructions to complete it. The interactive scenes may contain some commands not present in the instructions or set of steps to complete the interactive. These are there only so that the interactive scene is correctly displayed within a container in which the user can switch between the complete interactive and the document containing the instructions.

The exercises are typically made up of a series of steps designed to guide the user to the creation of an interactive scene. However, the guiding steps are intentionally not explicit. This will hopefully make the reader practice, try different approaches, and check, if necessary, the user guide to achieve a final scene. This may result in slight differences between the user's interactive scene and that provided as an exercise solution (it is possible for two different approaches to yield one interactive scene). Observations to each step in the creation of a scene are also provided, mentioning the expected result after each step. These observations may sometimes include suggestions or information to help the used achieve the desired result for each step.

The exercises in this document tend to increase in difficulty as the reader progresses. Later exercises involve more functionality and of mor complexity. That is why it is a sound idea to address them in the order in which they appear, even though the reader may skip

to different functionality via the hyperlinks provided in the document.

The reader will hopefully be able to generate his/her own *DescartesJS* interactive scenes after reading this document. It is also possible the reader may end up creating interactive scenes of greater complexity than those provided in the exercises, due to the versatility of the programming tool.

It is important to keep in mind that this document is not static. It is updated almost as soon as changes are implemented in the programming tool, and corrected when errors are spotted. It is therefore a good idea to periodically download it so as to have the latest version. This particular version of the document covers the functionality of *DescartesJS* up to version 1.3

As this is the english version of the user manual, the menus and interaction with *DescartesJS* will be set to English, even though it is set to Spanish by default.

2

What is DescartesJS?

2.1 The beginnings of Descartes

Descartes was originally developed in Java at the end of the twentieth century as an interactive scene generating tool. By then it consisted of a Java program which was able to generate *.html* browser files, so that it was possible to view the interactive scenes as web pages.

html files made in Descartes typically house interactive content and are, also typically, developed to improve teaching certain aspects of physics and mathematics across a wide variety of difficulty levels.

Even though there is quite a menu of interactive programs from which to choose, such as GeoGebra, Cabri and others, Descartes is a very versatile tool which, in turn, can produce from very simple interactive programs to much more complicated ones such as geometry editing interactive scenes. Addititonally, it allows for specific interactions defined by the teacher/programmer, such as exercises which involve randomness (for instance, generating problems and also when building optional answers to a given exercise) and that can be custom-made to suit the teachers' needs (for instance, by defining the number of decimales, etc.). These and many other features, which will be addressed further along on this document, make of *Descartes* a very useful tool.

In its original Java version, *Descartes* was developed to be used solely in computers. However, to new technologies made it necessary to change its functionality.

- The advent of mobile devices
- The HTML5 canvas element.

2.2 Recent changes

It eventually became necessary for the Descartes generated interactive scenes to be able to run on mobile devices (on JavaScript). *Descartes 5* is born out of this need. Even though the editor was not drastically changed, a new library had to be included.

Descartes 5 is the penultimate main version of Descartes. It was also developed in Java. However, the JavaScript version of Descartes (named *DescartesJS*) is the most current main version. Its editor is built on JavaScript and its functionality is very similar to

that of *Descartes 5*. *DescartesJS* nonetheless has many new features and advantages over *Descartes 5*. All this will be explained later on the document.

It is also worth mentioning that, unlike *Descartes 5*, *DescartesJS* has the advantage that whatever is viewed on the editor will be viewed exactly once the saved file is opened in a browser. This is an improvement on *Descartes 5*, in which text fonts and sizes were sometimes displayed differently in the editor and in a browser. Even though a scene's edition is performed on the editor, to be later opened in the user's browser of choice, the functionality and aspect remains the same in both cases since both use the same Descartes intepreter (the interpreter will also be addressed shortly).

3

Introduction to *DescartesJS* components

Descartes JS has two main elements:

1. The editor
2. The intrpreter

3.1 The editor

The editor is a JavaScript program (*DescartesJS.exe*). Its installer can be downloaded from `https://descartes.matem.unam.mx`, and maintenance is periodically performed to include improvements and bug fixes. So, it is good practice to reinstall it every now and then (in my experience, at least once a month).

Once installed, a shortcut is created in Windows on the desktop.

The editor program is a graphic user interface by which the user can save *html* files with various types of interactive scenes. Figure 3.1.0.1 shows the editor. Its functionality will be addressed shortly.

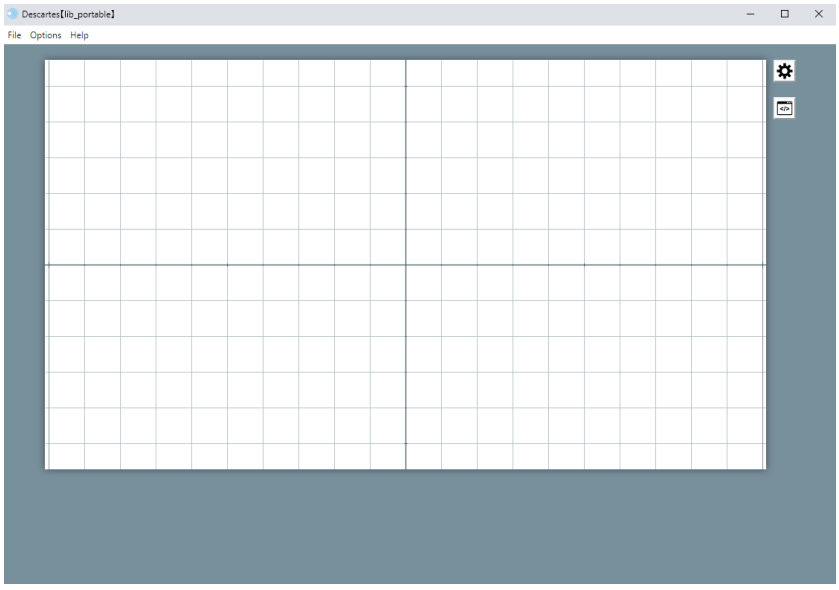


Figure 3.1.0.1: The *Descartes JS* editor upon launching.

Suggestion: *DescartesJS* can be installed in a folder of choice. It is good practice to install it near the system root. For instance, in Windows it could be installed in a *c:/DescartesJS* folder.

3.2 The *interpreter*

The *interpreter* is a *js* file (*descartes-min.js*) that can be downloaded from `https://arquimedes.matem.unam.mx/Descartes5/lib/descartes-min.js`.
However, every time the editor is opened, it checks online to verify if the user's interpreter is the latest version. If not, it will prompt to download the latest version, as is shown in Figure 3.2.0.1. The interpreter's update dialog appears upon launching *DescartesJS* right after it has been installed, and will subsequently be displayed only if the local interpreter is older than the online one. Additionally, if the server housing the interpreter is down, it can be automatically downloaded from an alternative site so that it is always up to date.

The updates may ocasionally include some editor's funcionality changes. Which is another reason to accept updates whenever they are available.

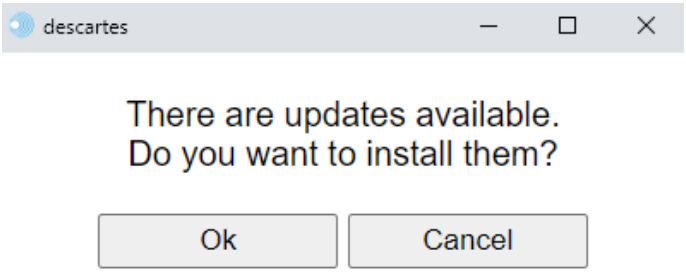


Figure 3.2.0.1: Interpreter update dialog.

This interpreter is given that name because it is a file which enables browsers to correctly *intepret* the code in the *html* file generated by the *DescartesJS* editor, and to therefore display it correctly.

It is also known as the *library* (*DescartesJS library*), though that term is not preferred since it may be confused with the library functionality of *DescartesJS* itself.

4

Learning to use the editor

4.1 First time setup

A couple of changes have to be implemented for language purposes. Quick instructions are provided in this section, even though the parameters used to make the changes will be addressed more thoroughly later in the document.

4.1.1 Changing the *DescartesJS* editor and configuration editor language

DescartesJS' default language is Spanish. So, it is necessary to change a couple of things following the original installation in order for the displayed language to be English.

Once the *DescartesJS* editor is opened, first click on the *Opciones* menu, then expand the *Lenguaje* option and choose *Inglés*. The menu bar, along with all their options, should now be displayed in English, as in Figure 3.1.0.1. The internal parameters of *DescartesJS*' *configuration editor*, as well as their tooltip information, will now be displayed in English.

This change remains even if *DescartesJS* is closed and reopened, so it is only necessary to perform it once.

4.1.2 Changing the scene configuration language

In order to change the language in which the *configuration* of a particular scene is saved, click on the tool button at the top left of *DescartesJS* (the one with the gear icon). A new window will be displayed. In it, inside the *Scene* tab, search for the *language* parameter and set its menu option to *english*. Finally, click on *Ok*. This changes the *DescartesJS* scene configuration language. Within the interactive scene's *html* file, there is a block of code flanked by an `<ajs>...</ajs>` label, in which all the *DescartesJS* related information is stored. It consists of various parameters and values for each, which are in Spanish by default. However, by changing the configuration language, it is possible to save them in English.

This change is only recommended for users who plan on editing the *html* directly from a text editor and not only via *DescartesJS*. However, users who only plan on using *DescartesJS* can ignore this. It is necessary to do this change every time a new *html* scene file is saved from scratch, since the default language for each is Spanish. However, it is not necessary to change it for loaded *html* files.

4.2 The Editor's menu bar

As with any program, the Editor has a menu bar. Al igual que cualquier programa, el Editor tiene una barra de menús. These consist of:

4.2.1 The *File* menu

- **New**: Creates a new *DescartesJS* file.
- **New window**: Launches a new *DescartesJS* window when one is already open. When a user is editing a scene, and wishes to edit a second one simultaneously, it is necessary to use this menu option, rather than trying to open the scene via the Start Menu.
- **Open**: Launchese a system dialog through which a *DescartesJS html* file can be selected for opening.
- **Recently opened**: When hovered with the mouse, displays a list of the paths of the recently edited files, so that the user can locate and open them with greater ease by simply selecting the file.
- **Reload**: Reloads the interactive scene from the most recently saved *html* file.
- **Save**: If the interactive scene has not yet been saved, this options desplays a dialog to choose the *html* scene's path and save name. Otherwise, using this option simply results in overwriting the file with the recent changes. It is always wise to save periodically so as to reduce teh chance of losing progress due to a program crash.
- **Save as**: Launches a dialog to choose a path and an *html* file name in which to save the current progress.
- **Show container folder**: Opens the folder in which the currently edited file is stored.
- **Edit scene**: Launches the *scene's configuration editor* (*SCE*). This editor can also be launched by clicking the tool button (the one with the gear icon displayed at the top right corner of the canvas).
- **Export**: Has several options, each of which launches a dialog to save the file under edition.
  - **Descartes macro**: Launches a dialog to save part of the scene's content as a *macro*. A *macro* is typically a text file (usually with a *txt* file extension) with information from the Definitions, Programs, and Graphics tabs (present in the *SCE*). This information can be later imported into other *DescartesJS* scenes. More information on this functionality can be found under the Macro graphic section.
  - **Descartes library**: Stores the scene's content as a new library (also a text file with its *txt* file extension). This library can be later imported via the *Definitions* tab. It is useful when there is a large number of definitions which the user may

wanto to group into a library. Its functionality is similar to that of the *macro*, except that a *library* stores only information from the Definitions tab. More information on the Descartes library can be found under the Library section.

  - **png**: Allows for the *DescartesJS* file displayed to be saved as a *png* image file.
  - **jpg**: Allows for the *DescartesJS* file displayed to be saved as a *jpg* image file.
  - **svg**: Allows for the *DescartesJS* file displayed to be saved as an *svg* image file. This functionality is still in an experimental phase.
  - **pdf**: Allows for the *DescartesJS* file displayed to be saved as a *pdf* file.

- **Close scene**: Closes the currently edited *DescartesJS* scene file. If it has undergone unsaved changes and this option is selected, a confirmation dialog is displayed indicating that information will potentially be lost should the user proceed.
- **Exit**: Closes *DescartesJS* altogether. As with the *Close scene* option, a confirmation dialog is displayed when there are unsaved changes.
  If an attempt is made to close *DescartesJS* via the system's window closing button, the confirmation dialog is also displayed. If the user clicks on *Cancel*, the program will not be closed and no changes are lost.

Some *File* menu elements include keyboard shortcuts for easier access. These shortcuts are shown at the right of each of the menu's options. They are not addressed in this document since they depend on the operating system being used. The user can, however, display the menu to check them.

It is also possible to open an *DescartesJS html* file by dragging and dropping it on the *DescartesJS* Windows desktop icon created upon installation. Yet another way to open an existing file in Windows is to right click it and expand the *Open with* menu option. A *DescartesJS* menu element with the Descartes icon is visible, and clicking it will open the file in *DescartesJS*.

4.2.2 The *Options* menu

The options in this menu are:

- ***descartes-min.js***: When selected, a submenu is displayed at its right with the following options related to the *DescartesJS* interpreter:

  - **internet**: Whenever a file save action is performed, the *descartes-min.js* interpreter file is read from its location in the Internet (`https://arquimedes.matem.unam.mx/Descartes5/lib/descartes-min.js`). This option has the advantage that the used interpreter file version is always the most recent available. However, it does have the disadvantage that the interactive scene files generated while saving under this option will not work if disconnected from the Internet.

  - **portable**: This is the option selected by default. Every time a file is saved, a */lib* folder containing the interpreter (*descartes-min.js*) is created in the same folder where the scene is being saved. If the folder with the interpreter file already exists, the file will be overwritten with the latest version obtained at the time the program checks if there are updates available. The advantage of using this option (as well as any other different from *internet*) is that, when a user loads the scenes saved under this scheme in a browser, it is not necessary to have internet connection, as the interpreter used is the local one.
  - **project**: When saving with this option selected, the */lib* folder containing the interpreter is generated one level *above* that where the *html* scene files are being stored. For example, if the *html* files are stored in *c:/Project/scenes/*, the location of the */lib* folder will be *c:/Project/*. This option is useful when there are a lot of *html* scene files all grouped at the same location and pertaining to a same project. Only one copy of the interpreter is enough to service all these files, which in turn saves disk space.
  - **custom**: The user enters, via a pop-up dialog, the location where the interpreter is to be saved. This option is useful for projects with complex file structures where there are various levels of scene files, but which are all to be services by a single interpreter file.

  As already stated, the last three options allow the user to use a local version of the interpreter. This enables scenes to work even in the absence of internet connectivity. It is useful to understand how the interpreter file is stored and when using the local options. When opening *DescartesJS*, it checks for the latest interpreter version stored in the web and keeps it if it is newer than the one locally stored. Then, upon saving a scene, it will copy it to the local folder (depending on the local option chosen by the user) if it does not already exist. If it already exists, it will be overwritten by the latest version.

- **Add to HTML**: When selected, the following submenu options are displayed, all of which are selected by default:

  - **library**: If selected, the content of any Descartes libraries imported in the scene is stored as a copy in the scene's *html* code upon saving the scene. It is stored as text inside a *script* label. That is, after the *html* file's body, there is a `<script> </script>` text block including the name of the library.
    Suppose the scene being edited imports a library named *prm.txt* inside an *lbr* folder. The scene will be able to use any definitions in that library. And if the *library* option in the *Add to HTML* menu is marked, the text content of that library will be saved in the scene's *html* file under the script flanked by the labels `<script type=''descartes/library'' id=''lbr/prm.txt''>...</script>`. Even if the original library file (*prm.txt*) were to be deleted, if the copy of such library is still inside the scene's *html* code in the form of a script, the scene will then import any necessary definition from the script.

  - **macro**: If this option is marked, the text of an imported *macro* is included as a script (between the `<script></script>` labels) in the scene's *html* code upon saving the scene. It works in basically the same way as with the libraries.
  - **array**: Does pretty much the same as the two options above, but for arrays. *Arrays* are a type of definition, and will be addressed further along the document under the array section. When the *array* option is marked, the array's information imported from a text file is included as a script in the scene's *html* code upon saving. And, in the same way as in the previous cases, its information can be retrieved from this script in the event that the text file containing the information goes missing.

- **Language**: It is a menu to select the language used both in menus as inside the *scene's configuration editor* (the one launched by pressing the tool button, the one with the gear icon on the top right corner of the canvas). This is actually one of the ones changed in section 4.1 in order to get *DescartesJS* working in English. The user may set it to *Spanish*, *English*, *German*, *Catalan*, *Basque*, *Galician*, *Valencian* and *Portuguese*.
- **Zoom**: A menu with three options that control the zoom to *DescartesJS*' work area or canvas. It is particularly useful when the size of a scene does not fit in a certain screen, and it is therefore necessary to zoom out. Any changes in zoom are only for the user's comfort, and will not have any impact whatsoever on the scene being developed. The three options inside *Zoom* are:
  - **Increase**: slightly increases size.
  - **Decrease**: slightly decreases size.
  - **Initial**: restores the original size of the canvas before implementing any zoom change.
- **Color theme**: A menu to select the background color behind *DescartesJS*' work area or canvas. The options are *Classic* (a gray-green color typical of *DescartesJS*), *Dark*, *Light* and *Blue*.
- **Show console**: When this option is used, a window known as the *DescartesJS console* is launched. Its function is to list possible parsing errors *DescartesJS* finds. It can also be used to print values which can themselves be used for debugging purposes. Printing is achieved via the `_Print_()` or `_Trace_()` commands (both are equivalent). These functions' argument is typically a variable whose value the user wishes to know. The argument can also be an expression. For instance, in a loop where the value of a certain variable *i* increases by 1 in each step, the instruction `_Print_(i+',’+(i+1))` within the loop will result in printing the value of *i* in the console, followed by a comma, and followed by the value of *i* plus one unit.

  Note that the console also allows for the visualization of internal programming errors (for example, divisions by zero, square roots with negative arguments, etc.). It

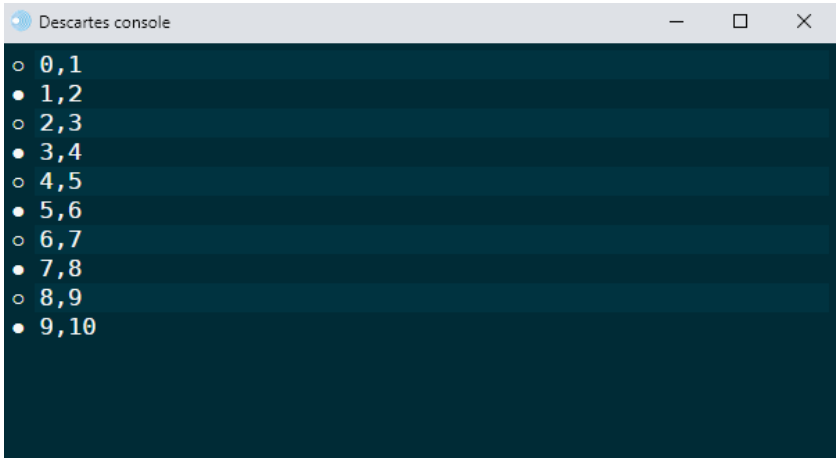


Figure 4.2.2.1: The DescartesJS console, printing the `i` and `i+1` example mentioned.

is suggested to always have it open when developing a scene involving complicated programming, so as to be able to be notified of any errors as soon as they are entered.

When the amount of lines in the console excedes its size, a vertical scroll bar appears which can be used to navigate the screen. When any new lines added to the console, these are displayed at the end of its contents. And whenever this happens, the scrollbar is placed at the bottom, so as to be able to see the last lines first.

It is always possible to clear the console's contents by closing it and reopening it again.

The errors printed in the console are in Spanish.

4.2.3 The *Help* menu

This menu includes options related to *DescartesJS* miscellaneous information.

- **Documentation**: Launches a browser window displaying the pdf of the current documentation (check if it's possible to set the link to the English documentation).
- **Release notes**: Launches a window which indicates the latest version number and which includes a list of this version's changes / improvements. It also has a *Versiones anteriores* (*previous versions*) hyperlink at the end of the list, which can be used to obtain the history list of the previous versions along with the change / improvement lists.
- **About Descartes**: Launches a window which contains the *DescartesJS* logo, the editor and interpreter's version, and information about the program's author and license.

5

The scene configuration editor (SCE)

The scene configuration editor is a new window where the real programming takes place. This window is launched by pressing the tool button (the one with the gear icon near the top right corner of the canvas). Figure 5.0.0.1 shows this button's location.

Figure 5.0.0.1: Tool button, which launches the scene configuration editor.

The button below, with a code screen icon, launches a screen in which the code of the scene can be edited as text. Figure 5.0.0.2 shows this screen.

The elements of the scene are enumerated in this screen. They are also separated via alternating background colors, so as to be able to locate a particular code line por easily. This code can be manually edited and any changes will be implemented upon pressing the *Ok* button at the bottom of the screen. However, the user should be extra careful when using this screen for edition, since a mere typo can be enough to wreck the code. Even though the code may seem strange at the beginning, as we go forward it will be easier to understand.

Figure 5.0.0.3 shows the configuration editor window. The superior part of this window contains various tabs essential for its use.

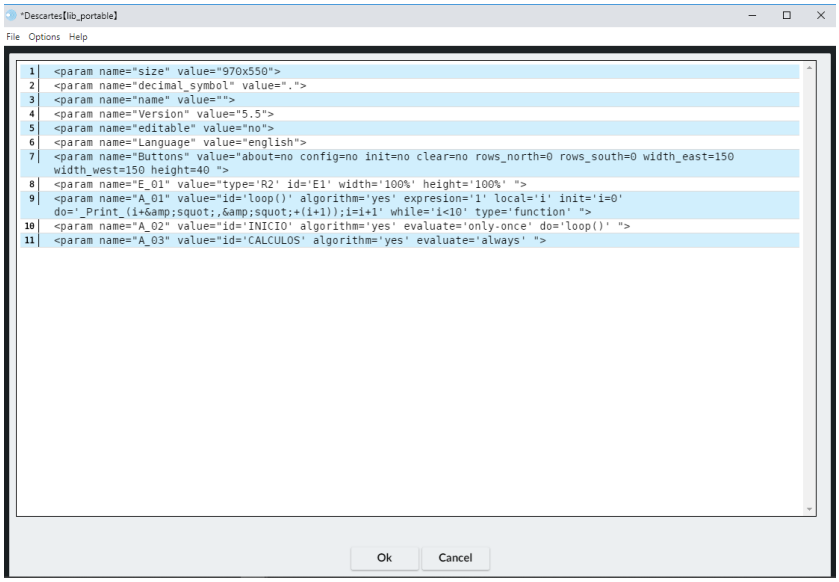


Figure 5.0.0.2: Scene edition screen.

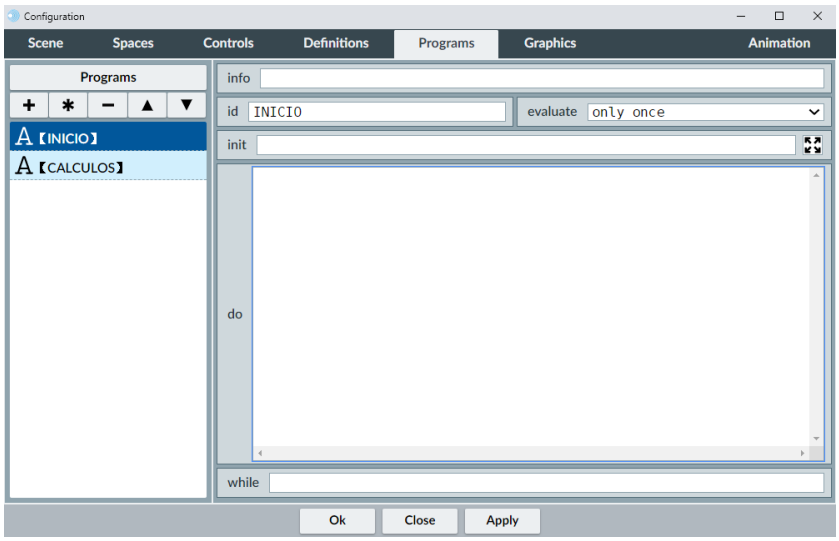


Figure 5.0.0.3: Configuration editor window.

5.1 Tabs

The scene configuration editor consists of seven tabs used to edit different parts of the interactive scene. A brief description of each is provided in this section, even though later sections include more in-depth information on each.

The *Scene* tab is used to make changes in the scene's general interface.

The *Spaces* tab is used to add, duplicate, remove or edit the properties of the existing

spaces in an interactive. Spaces are areas which house graphic elements as well as controls with which the user can interact.

The *Controls* tab is used to add, duplicate, remove or edit the properties of the interactive scene's controls. These controls are the means of interaction with the user.

The *Definitions* tab is used to add, duplicate, remove or edit the properties of elements making up the scene's actual programming. This part contains elements inside which the hard code that makes the scene work is housed.

The *Programs* tab contains algorithms and events. Just as the Definitions, these also house the actual code behind the scene's functionality. In particular, they have to do with the scene's initial preparation, so that it is ready to be used. But they also have to do with instruction which are repeated in every step or only when certain conditions are met.

The *Graphics* tab is used to add, duplicate, remove or edit the graphics displayed as part of certain Spaces.

The *Animation* tab is used to edit the instruction and conditions of an animation, if one is to be present.

As already mentioned, all these tabs will be addressed with more depth in their respective sections.

A very useful editing tool are the pop-up information tooltips that appear when hovering the name of a parameter with the mouse. For example, the *do* in Figure 5.0.0.3 will display its tooltip information if the mouse is held over the word for a moment. The information appears as a panel containing the explanatory text about the item being hovered. The panel disappears when the mouse is moved. Figure 5.1.0.1 shows an example of this type of help for the *do* parameter of the *INICIO* algorithm inside the *Programs* tab. The current version of *DescartesJS* offers the option to choose from among 10 different languages: English, German, Catalan, Euskara, French, Galician, Italian, Valencian and Portuguese.

5.2 Buttons in the scene configuration editor

The configuration editor also has three buttons at the bottom.

- *Ok*: When a change is made in the configuration editor, it is not immediately implemented on the scene. When the *Ok* button is pressed, the configuration editor window closes and focus is set on the editor window, where the scene is displayed in its canvas with the latest changes implemented.
- *Close*: This button closes the configuration editor, but does not apply any changes. Any new change will not be present in the interactive scene after *closing* the configuration editor.
- *Apply*: This button applies all changes. The interactive scene will refresh itself with the latest changes. This button basically does the same as the *Ok* one, except that

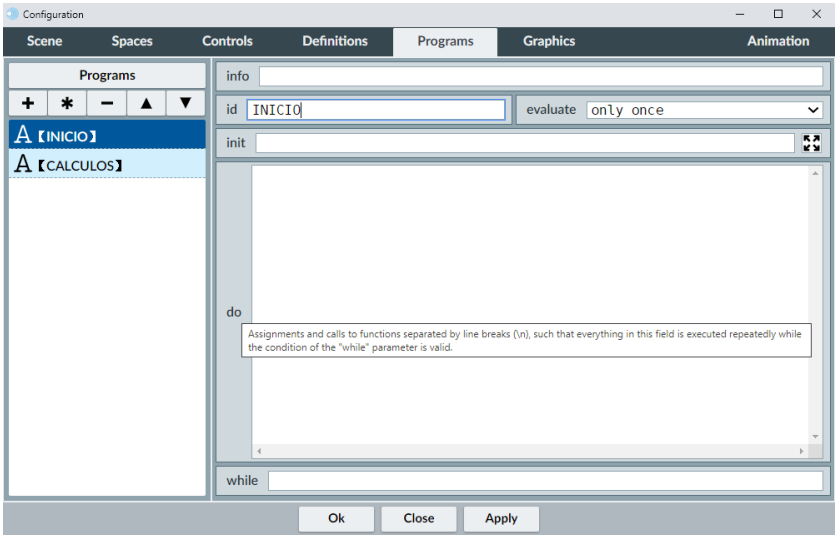


Figure 5.1.0.1: Emergent tooltip information for elements in the configuration editor.

it does not close the configuration editor: focus is set on the *DescartesJS* editor, and the configuration editor remains open in the background.

Whenever any of these three buttons is used, the *DescartesJS* main editor (the one with the canvas) associated to that configuration editor window is brought up as the top window. This enables the user to quickly focus on the interactive scene in question, which is usefull when multiple scenes are being edited in multiple instances of *DescartesJS*. In this way, the user knows which scene was the one in which changes were recently applied.

When using automatic animations (animations that launch themselves as soon as the interactive scene containing them is loaded), it is better to use the *Ok* button instead of the *Apply* one when implementing changes, since only by using the *Ok* button will the animation begin automatically. When loading any such scene in a browser, the animation will always start automatically if it is so set in the Animation tab.

It is also worth remembering the clicking on *Ok* or *Apply* does not necessarily mean that the changes are saved. Changes can be made via the configuration editor, for example, by clicking the *Ok* button, after which the interactive scene will implement the changes. However, the *html* scene file will not have such changes until it is saved via the *Save* or *Save as* options in the *File* menu.

6

The *Scene* tab

Scene allows the user to control the most general aspects of the scene's interface. Figure 6.0.0.1 shows the tab and its elements.

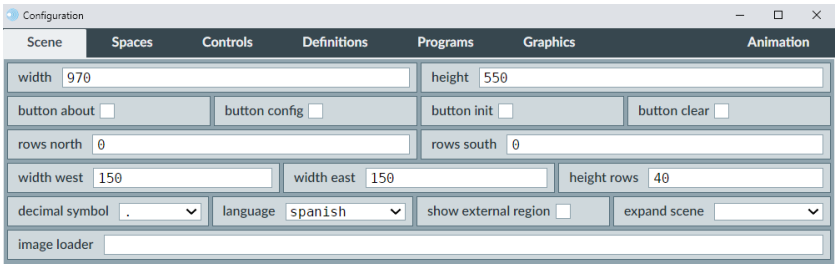


Figure 6.0.0.1: *Scene* tab parameters.

The parameters in this tab are the following.

- **width**: A text field where the width of the scene (measured in pixels, or *px*) is set.
- **height**: A text field where the height of the scene (also in *px*) is set.
- **button about**: A checkbox that, when marked, includes in the interactive scene a button labeled *créditos*. When this button is clicked, a pop-up window is displayed with information about the authorship and license of *DescartesJS*.
  This functionality of launching the information window is designed specifically for scenes that are viewed in a browser, not in the *DescartesJS* main editor. Pressing this button in the editor will launch a system window only.
- **button config**: A checkbox that, when marked, includes in the interactive a button labeled *config*. When clicked, a pop-up window is displayed which contains the *DescartesJS* code of the scene. This code can be embedded in a web page if, for instance, the user wishes a web page to include such information.
  As with the *about* button, this button also only works when viewing the interactive scene in a browser, and not in the *DescartesJS* editor.
- **button init**: A checkbox that, when marked, includes in the interactive scene a button labeled *inicio*. When this button is clicked, it reloads the scene from the beginning, as if it had just been loaded from the last saved version, thus removing any changes the user may have introduced since it was last saved.
- **button clear**: A checkbox that, when marked, includes a button labeled *limpiar* in the interactive scene. When this button is clicked, any graphic traces that could have

been introduced by the user are erased. More information on Traces can be found later in the document.

- **rows north**: A text field in which the user enters an integer that corresponds to the number of rows that make up the north (topmost) panel of the scene.
  This panel has a gray background, and typically houses controls. When the *about* and/or *config* buttons are to be displayed, they will be displayed in a north panel row regardless of whether rows are set to be displayed in the north panel via the *rows north* parameter or not.
- **rows south**: A text field in which the user enters the number of rows that make up the south (bottom) panel of the scene.
  This panel also has a gray background and can also house controls. When the *init* and/or *clear* buttons are to be displayed, they will be displayed in a row in the south panel regardless of whether rows are set to be displayed via the *rows south* parameter or not.
- **width west**: A text field in which the user enters the number of *px* of the width of the west (left) panel.
  Again, this panel has a gray background and can also house controls.
- **width east**: Basically the same as *width west* but related to the east (right) panel.
- **height rows**: A text field where the height of the rows (the same rows indicated in the *rows north* and *rows south* parameters) are entered in *px*.
- **decimal symbol**: A menu to choose the type of symbol used to divide the integer part from the decimal part in a number. The possible options are a period and a comma.
- **language**: A menu with language options in which to display the saved scene settings.
  As a reminder, this sets the language in which the parameters in the *DescartesJS* block of the scene's saved *html* file are displayed, but has nothing to do with the language in which the *DescartesJS* interface is shown. This is set in the language option of the *Options* menu in the *DescartesJS* main editor, whereas the saved scene settings are set via the *language* menu in the *Scene* tab.
- **show external region**: A checkbox that, when marked, enables the user to lauch the *external region* by right clicking the mouse on a scenes canvas.
  The external region is a pop-up window that can be used to house controls as well. This window **always** houses the *about*, *config*, *init* and *clear* buttons, even when they are not to be displayed as part of a scene (when their respective checkboxes are unmarked). Any control with its region set to *external* will be found in this window. This region is typically reserved for the programmer developing the scene, who may use controls for debugging purposes that are not to be visible for the end user. In these cases, the show external region is unmarked prior to releasing the scene to the end user.

- **expand scene**: A menu via which a scene that does not exactly fit in the container can be altered to make it fit. This behavior can only be seen correctly in a browser, since the editor's size is defined by the *width* and *height* parameters in the *Scene* tab.

  When set to cover, if the browser's area exceeds that of the scene, a larger space will be considered to cover the browser's area completely. No rescaling or stretching takes place; it only considers more (or less) of the space being shown in the editor, so as to cover the browser's area. The spaces in the scene keep their scale.

  When set to *fit*, if the browser's size is different than the original scene, the spaces in the scene are rescaled until they fit in the browser. The horizontal and vertical scales are changed until either the scene's width fits the horizontal span of the browser, or the scene's height fits the vertical span of the browser (whichever happens first). Thus, the relative horizontal to vertical scale ratio is maintained, and the scene will not be deformed.

- **image loader**: A text field where a path and file, relative to the folder that houses the scene's file, is introduced. If left empty, the *DescartesJS* default logo is displayed upon loading a scene. Otherwise, the image set in the path will be displayed. This behavior can be seen correctly only in a browser, as it may not work in the editor.

The *config*, *init* and *clear* buttons may be helpful during the development of a scene. They are, however, seldom useful once the scene is finished. Additionally, since these buttons are found two in the north region and two in the south one, their presence results in the reduction of the area available for the scene.

7

The *Spaces* tab

Spaces has the parameters necessary to control all the spaces (initially shown as cartesian planes) present in an interactive scene. All the graphic objects and texts are drawn inside spaces.

The default space shown is a 2D space displayed right after launching the *DescartesJS* main editor. Within the *Spaces* tab, it is labeled *E1* in the left panel that contains the list of the scene's spaces. Figure 7.0.0.1 shows an example of this tab.

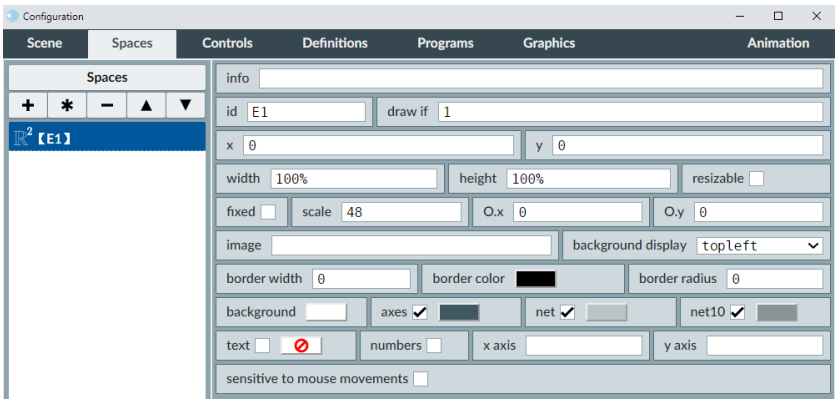


Figure 7.0.0.1: The *Spaces* tab.

The space shown in the Figure has the identifier *E1* by default, as can be seen in the *id* text field in the *Spaces* tab. Notice that the *width* and *height* of this space are set by default to 100%, thus indicating that this space is to span all the scene's area defined in the *width* and *height* parameters of the *Scene* tab.

7.1 *R2* or two dimensional space

We now address the parameters found in 2D spaces in the *Spaces* tab.

- **info**: Text field where the user may enter a reminder of information about this space (i. e., what it does, or what it stores). The purpose of this brief description is only as a reminder to the programmer, or to make it easier to identify a space. It will not be shown to the end user. If text is entered in the *info* text field, part of it will be displayed in the list panel at the left, so that the programmer may easier find a particular space when there are many spaces involved.

The *info* text field is present in **all** other elements (not just spaces) of the *DescartesJS* configuration editor. It is there as a means to better identify certain elements when there are many involved. Even though it is present in future elements of the configuration editor, its description will not be repeated for each element since its function is the same as stated here.

- **id**: A text field that contains the space's identifier. It is the name with which the program identifies the space in question. Identifiers start with letters. Identifiers are only important in the program itself; they will not be known to the scene's end user.
- **draw if**: A text field in which a boolean condition is entered. This condition is evaluated to determine if the space will be displayed or not.
  Many elements in *DescartesJS* have this *draw if* attribute, which easily allows the programmer to build objects that are shown or hidden under specific conditions. If, for example, `2==1` is entered, the program compares 2 with 1. As they are not equal, the expression is considered false, a 0 is internally returned, which in turn results in the space **not** being displayed. If, however, `1==1` is entered, since this expression is true, a 1 value is internally returned and the space **is** displayed. It is also possible to directly enter 1 or 0 in the text field in order to respectively display the space or not. More information on boolean conditions can be reviewed in the boolean operators and conditions section.
- **x**: A text field in which the horizontal coordinate of the top left corner of the space is entered in *px*. Measurements start at the top left corner of the scene's area. The space's top left corner will therefore be *x* pixels to the right of the top left corner of the canvas.
- **y**: A text field in which the vertical coordinate of the top left corner of the space is entered in *px*. Measurements start at the top left corner of the canvas, and positive amounts result in moving the space down. The space top left corner will therefore be *y* pixels below the top left corner of the canvas.
- **width**: A text field which contains the width of the space in *px*. If the amount entered has a % suffix, the amount is taken to be the percentage of the scene's width (the width entered in the *width* parameter of the *Scene* tab) instead of as a number of pixels.
- **height**: A text field which contains the height of the space in *px*. Again, a % suffix indicates the amount entered is to be considered as the percentage of the scene's total height.
- **resizable**: A checkbox that, if marked, allows for a more flexible behavior of the space in question. When activated, the height and width of the space can be defined via variables.
  If a variable is entered for the height or width of the space, the value of that variable will determine the height and/or width, making that space a more dynamic object. It is therefore possible to assign the space a height or width greater than that of the entire scene.

- **fixed**: A checkbox that, when marked, fixes the axes of the space.
  If the axes are not fixed, the end user can move the origin of the space with the mouse (by pressing, dragging and dropping). The scale of the space (the number of *px* that make up a unit length) can also be changed by pressing the right mouse button and dragging vertically (up to increase the scale, down to reduce it). If the *fixed* checkbox is activated, these actions are no longer possible. The spaces scale and offset can still be changed, though not using the mouse, and rather changing the value of certain space-the specific variables. More information regarding this functionality can be found under the space variables section.

- **scale**: A text field which contains an integer value corresponding to the scale of the space (how many pixels make up a length unit in any of the axes. The default value is 48 (the unit of length in the cartesian plane of the space is made up of 48 px). Reducing the value of the scale results in zooming out, while increasing it results in zooming in.

- **O.x**: (from *x offset*). A text field in which an integer value is entered, corresponding to the number of px by which the space's origin is moved horizontally from the center of the space.
  By default, the origin of a cartesian plane is placed at the center of the space. Entering a positive value for *O.x* results in shifing the origin to the right by that amount of px. Entering a negative value results in a shift towards the left of the space.

- **O.y**: (from *y offset*). A text field which indicates the vertical offset in px of the origin of the cartesian plane. It works in a similar manner to *O.x*, only vertically. A positive value corresponds to a downward displacement, while a negative one to an upward displacement.

- **image**: A text field in which a path and name of an image file (e. g., *png*, *jpg*, *gif* or *svg*) is entered. The image will then be displayed as the space's background. The path is relative to the folder where the *html* scene file is stored. For instance, `images/cover.png` would be an appropriate path and name if the *images* folder were contained in the same folder as the *html* file.
  In order for the changes to be evident, it is recomended to first save the *html* scene file, and then set the path and name of the image.

- **background display**: A menu with options related to the position where the background image is to be placed.

  -**topleft**: the image's top left corner is placed at the top left corner of the space.

  -**stretch**: the image is stretched both horizontally and vertically so that the whole area of the space is covered by it.

  -**patch**: the image repeats itself as many times as necessary so as to cover all the space's area.

  -**center**: the image preserves its size, but its center is placed at the space's center.

- **border width**: A text field in which the user sets the number of px that correspond to the width of a border around the space's area. If left with its default 0 value, no border is drawn.
- **border color**: A button that launches the color editor. Upon accepting a color, it will be set as that of the border surrounding the space.
- **border radius**: When the border corners are to be set as curved, this text field contains the number of px that comprise the radius of an arc that is to be the aforementioned curved corner. When using its default 0 value, the corners are not curved.
- **background**: A button which launches the color editor, with which the space's background color can be set.
- Checkboxes to edit the axes, net, text and numbers. Each checkbox comes with a button to launch the color editor, so as to easily set the color of each of the elements. The checkboxes are:
  - **axes**: the cartesian axes.
  - **net**: grid lines parallel to the coordinate axes used as a frame of reference.
  - **net10**: color of the grid lines drawn every 10 units in the previous grid.
  - **text**: displays the coordinates of the point where the user left clicks. muestra las coordenadas del punto en que se hace clic en el espacio. Este checkbox no tiene control de color. The checkbox is not marked by default, meaning the coordinates are not to be displayed when the user left clicks.
  - **numbers**: A checkbox that, when marked, results in drawing reference numbers at the axes.
    The color of the numbers is the same as that of the axes. If the axes are not drawn, the numbers are not drawn either, even if the current checkbox is marked. Its default value is unmarked.
- **x axis**: A text field in which the title name of the x axis is set.
- **y axis**: A text field in which the title name of the y axis is set.
- **sensitive to mouse movements**: A checkbox that, when marked, implies the program will recalculate all the mouse related variables when the mouse is hovered over the space in question, even if the user does not actively click the mouse.
  This functionality can be powerful, but its abuse is not suggested since it involves significantly increasing the number of internal calculations, which could result in a less than optimal performance.
  As it will eventually be seen, this functionality can also be used with 3D spaces.

7.2 R3 o tridimensional space

Up until now we have only dealt with 2D spaces. 3D spaces can also be used in *DescartesJS*. These are a type of space in which 3D objects can be handled. Figure 7.2.0.1 shows the configuration editor displaying a new tab (*3D Graphics*). This tab is only displayed when a 3D space has been added to the list of spaces in the Spaces tab, and the changes have been applied.

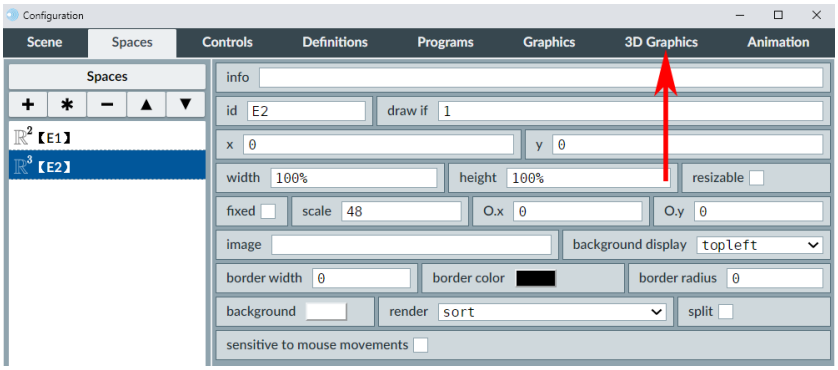


Figure 7.2.0.1: *3D graphics* location.

Most of the parameters of these spaces are the same ones present in 2D spaces, with only a few exceptions, namely:

- **render**: A menu which determines how the objects present in 3D are displayed. There are two different ways in which objects are rendered in 3D:
  - **sort**: draws 3D objects from back to front. This is the fastest method to draw objects, but has the drawback that objects composed of large faces may not be drawn correctly.
  - **painter**: draws first the objects that are covered by others in a given perspective. The rendering is slower when using this option, but is more reliable when objects intersect each other.
- **split**: A checkbox that, when marked, improves the way objects that intersect each other are displayed. If the user is sure objects in this space are not to intersect, this checkbox can be left unmarked, which is its default state.

3D spaces can be handled by pressing and dragging. The user can, via these interactions, perform rotations of the space so as to be able to view it from different perspectives.

And, by right clicking and dragging, the user can zoom in towards the origin or zoom out. Right clicking and dragging upwards will zoom in, while dragging downwards will zoom out, in very much the same way as in 2D spaces.

7.3 HTMLIFrame space

These spaces contain web pages. They have a *file* text field in which the path (including the file name) to a file is entered, relative to the html scene's file folder. It is also possible to enter a web address. As usual, to view the implemented changes, the user has to first save the html file in its containing folder and then enter the *HTMLIFrame* file. After clicking *Apply* the associated file will be correctly displayed.

7.4 Spaces' panel in the *Spaces* tab

Besides the control parameters present in the *Spaces* tab, there is a panel to the left, which can be used to create new spaces, duplicate existing ones, remove them, select them for edition purposes, and alter the order in which they are listed. This can be seen in Figure 7.4.0.1

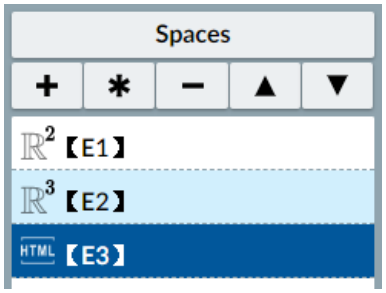


Figure 7.4.0.1: Left panel of the *Spaces* tab.

- *Spaces*: A gray button labeled *Spaces*. When clicked, a text edition window is launched in which the user can manually edit the properties of all the listed spaces. This is useful in cases in which, for example, a single same change has to be done in various spaces. Doing this change might require navigating from space to space in the configuration editor, while in the manual edition window it would be more immediate. The used must, however, be very careful while performing edition this way, since the accidental addition or elimination of certain characters could render the code unreadable. The use of this functionality is therefore only suggested for experienced *DescartesJS* users.
  It is important to point out that this functionality is not only available for spaces, but for all the other configuration editor tabs as well (with the exception of Scene and Animation). All the configuration editor tabs which handle a list of items (e. g., list of definitions, graphic objects, etc.) have this left panel and the top button which launches a text edition window that can be used to perform manual changes to the code.

- +: This button launches a dialog. In it, the user can provide, via a menu, the type of space to add (*R2*, *R3* or *HTMLIFrame*) and a name, via a text field, for a new space to be added. The dialog has the *Add* and *Cancel* buttons to implement the action or not.
- *: This button launches a dialog in which the user can, via a text field, provide a name to a new space that is to be a clone of the last selected space in the list. Upon pressing *Clone* in this dialog, a new space will appear at the botton of the list. This space is of the same type and has the same parameters' values as the one selected for cloning.
- -: This button launches a confirmation prompt. The user is asked to confirm the deletion of the selected space.
- order arrows: An upward pointing triangle and a downward one. These two buttons can be respectively used to move the selected space in the list upward or downward. The order in which spaces are listed is the order in which they are drawn on the canvas. For example, in Figure 7.4.0.1, the *E1* space is drawn first, then the *E2*, and then *E3*. If they are set at the same coordinates, then the last one in the list will end up covering the other two.

In order to get a better grasp of the concepts, we can perform an exercise to practice. This exercise's interactive scene, along with the instructions to successfully do it, can be found at Spaces. The standalone interactive scene's document is found at this link. All these files are also found in the *DescartesJSUserManual.zip* file. It is good practice to apply the changes after each step of the instructions, as well as to save periodically.

8

The *Graphics* tab

We skip ahead to the last of the tabs in the scene configuration editor: the *Graphics* tab. We do this because it is the most natural step after *Spaces* when learning to use *DescartesJS*. Graphics consist of a wide variety of figures and texts that are used to enrich the interactive scene.

Many of these graphics share the same functionality. So, the general or shared functionality of all the graphic objects is described at the end of the topic and is not present in each of them: only the parameters unique to a particular graphic object will be explained inside the description of that object.

Just as with *Spaces*, there is a panel to the left of the *Graphics* tab containing a list of the graphic objects. And just as with *Spaces* the + button adds a new graphic, the - button can be used to remove, there are two buttons to shift the selected graphic upwards or downwards in the list, and there is another button reading *Graphics* with which a code list of all the graphics is made available for edition. Figure 8.0.0.1 shows the *Graphics* tab.

Figure 8.0.0.1: The *Graphics* tab.

Graphic objects, or graphics, have to be housed in a specific space. There is a new menu at the top of the left panel of the *Graphics* tab that display's the existing spaces. This menu is only visible when there are graphic objects present (if there are none, the menu is not available). And it works as a filter: when a given space is selected, the list of graphic objects in the panel below will only contain graphics **belonging** to that space. If the menu is set to *, all graphic objects are listed, regardless the space to which they belong. This functionality allows for an easier identification of the graphics present, especially when there are many, and / or many spaces as well.

When multiple graphics present in a single space intersect each other, they are drawn in the order in which they appear in the list of graphics of their containing space. So, the first one in the list will be drawn first, and the last one last. If there are intersections, the last one covers the preceding ones in the list.

Also as with spaces, there is a *Graphics* labeled button at the top of the panel. This one launches a window with the code related only to the *Graphics* tab. The code here displayed contains **all** the graphic objects, not only the ones filtered for a given space.

8.1 2D Graphics

These include the most commonly used graphics, and are drawn only on 2D spaces.

8.1.1 *Equation* graphic

This graphic is used to draw the curve of a given equation. Figure 8.1.1.1 shows and *equation* graphic, and Figure 8.1.1.2 shows the configuration required to obtain that graphic.

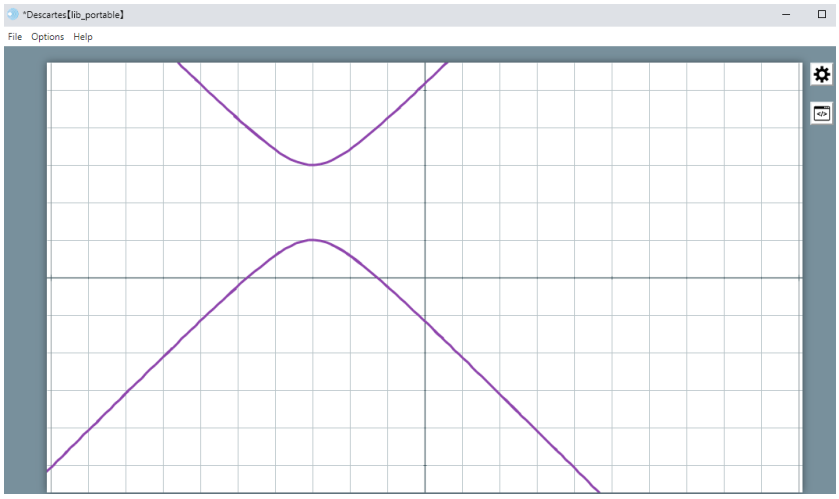


Figure 8.1.1.1: *Equation* graphic object example.

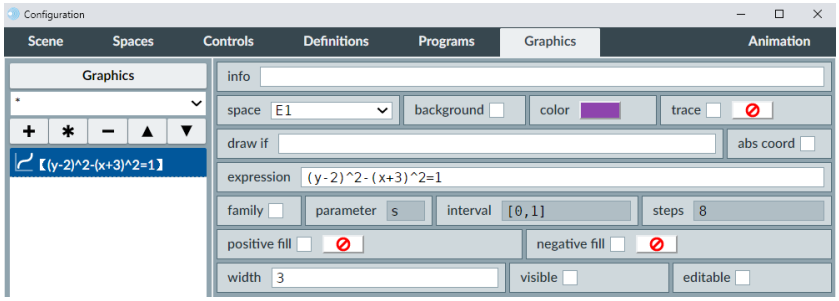


Figure 8.1.1.2: Configuration of the *equation* graphic. The hex color used is 8E44AD.

- **expression**: A text field where the equation to be graphed is entered. It uses the *y* and *x* variables to refer to the ordinate and abscissa variables in the plane, respectively.

For instance, the $y = x$ that appears by default draws the identity line. But it is also possible to enter $x^2 + \frac{y^2}{2} = 1$ (entered as $x \wedge 2 + (y \wedge 2)/2 = 1$). The *expression* can therefore be implicit, and not necessarily an explicit function y of x.

- **positive fill**: A checkbox that, when marked, enables the coloring of areas between the graph and the x axis, only for domain intervals for which the graph lies **above** that axis. It also has a color button (that is active only when the checkbox is marked) that launches the color editor to select the fill color. For example, for a $y = x^3$ function, the fill will be present right of the y axis between the graph and the x axis.

- **negative fill**: A checkbox that, when marked, enables the coloring of areas between the graph and the x axis, only for domain intervals for which the graph lies **below** the x axis. It also has its color button to select the negative fill color.

- **visible**: A checkbox that, when marked, enables a text field at the bottom of the interactive which has the entered equation in it. This is useful in cases where the user is supposed to know what is being graphed.

- **editable**: A checkbox that, when marked, allows the equation shown by marking the *visible* checkbox, to be edited by the user, allowing the graph to change as well. This checkbox only makes sense if the *visible* checkbox is marked.

- The **background**, **draw if**, **abs coord**, **trace**, **family**, **parameter**, **interval**, **steps**, **width**, **info** and **line style** are discussed at the end of the current topic. The color buttons included for the *color*, *positive fill color* and *negative fill color* are generic and explained under the color editor section.

It is important to bear in mind how the power operator works in the expressions of equations. Functions entered as `y=x∧p`, with p not an integer, are only defined for $x>0$. It would be necessary to enter, for example `y=(x∧3)∧(1/5)` if the graph to be drawn includes all the negative values of the x argument. If only `y=x∧(3/5)` is entered, the graph will only be drawn for positive domain values ($x>0$). This is related to the way in which roots are interpreted when using the $\wedge$ operator, and is explained with more detail under the power operator topic.

8.1.2 *Curve* graphic

This graphic consists of various segments that join given points determined by the graphic's parameters. The parameter is a variable that adopts values within an interval divided by a number of steps. A curve can be seen as a series of segments joined by their extreme points. Figure 8.1.2.1 shows an example of such a graphic. Figure 8.1.2.2 shows the configuration for this example.

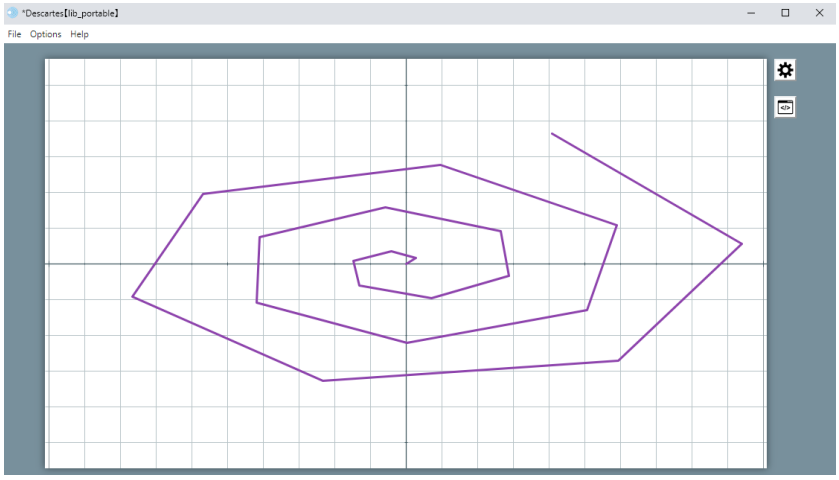


Figure 8.1.2.1: *Curve* graphic example.

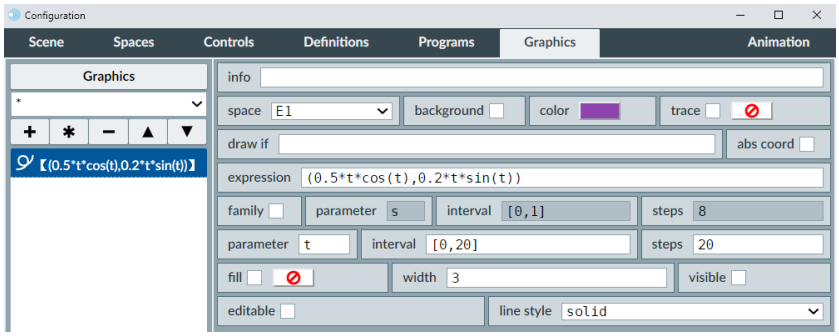


Figure 8.1.2.2: *Curve* graphic example configuration. The hex color used for the example is 8E44AD.

- **expression**: A text field in which the positions of the vertices defining the curve are entered via a couple of parametric equations (one for the horizontal coordinate, one for the vertical) using a certain parameter. The default value is (t, t), so the default curve lies on the identity line.
- **parameter**: A text field where the parameter to be used in the expression is specified. By default, the parameter is t. It is important to stress the difference between the curve's parameter and the family parameter (s by default).
- **visible**: A checkbox that, when marked, enables the curve's expression to be displayed in the interactive near the bottom.
- **editable**: A checkbox that makes sense only when the *visible* checkbox is marked. When marked, it makes the displayed curve's expression also editable, so that the user can also modify it, and thus modifying the graph.

Now that we have seen two different graphic objects, we can do a brief exercise. We

will see how a many sided polygon inscribed in a circle can be made to resemble the circle itself.

This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Curve. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

8.1.3 *Point* graphic

This graphic consists of a single point with its coordinates explicitly provided. Figure 8.1.3.1 shows an example of this type of graphic object. Figure 8.1.3.2 shows the configuration required to get such a graphic. Note that the cartesian plane can be clicked and dragged so that its position is shifted.

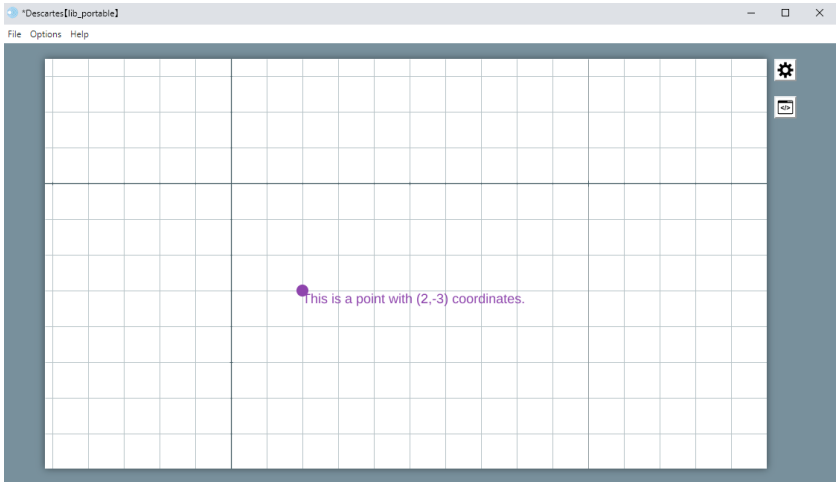


Figure 8.1.3.1: *Point* graphic object example.

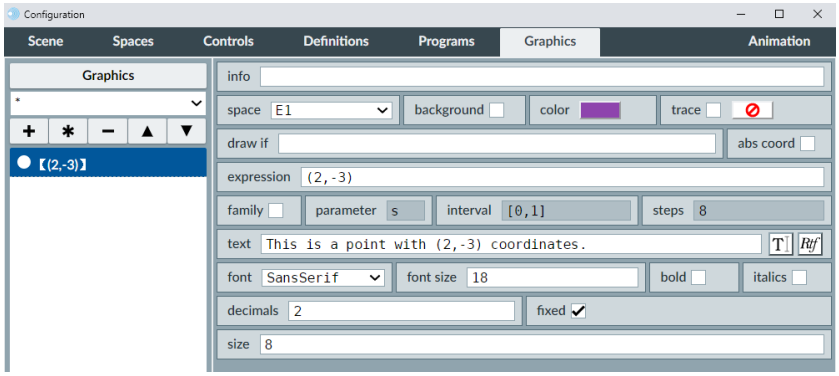


Figure 8.1.3.2: *Point* graphic example configuration. The hex color used for the example is 8E44AD.

- **expression**: A text field where the coordinates of the single point are entered.
- **decimals**: A text field in which an integer is entered, corresponding to the number of decimals that are to be shown should the text include any variable value.
- **fixed**: A checkbox which, when marked, forces the text to display the number of decimales chosen via the *decimals* paramter, even if they are not significant (here we take a significant decimal as that which is either non-zero, or is zero but has non-zero trailing digits to the right). When this checkbox is marked, the specified number of decimals will always be displayed, and no more. If not, only that number of significant decimals will be displayed.

A brief exercise might help clarify some of these aspects. In this exercise, we will also review families of graphic objects (in this case, family of points), a bit of texts and relative vs absolute coordinates. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Point. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

8.1.4 *Segment* graphic

This graphic object is, as its name says, a segment flanked by two points. Figure 8.1.4.1 shows an example of this type of graphic object. Figure 8.1.4.2 shows the configuration required to obtain this display.

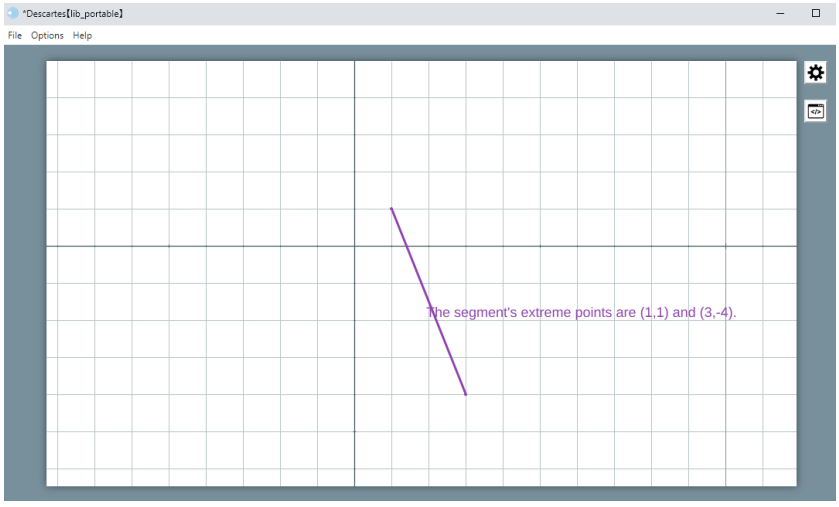


Figure 8.1.4.1: *Segment* graphic object example.

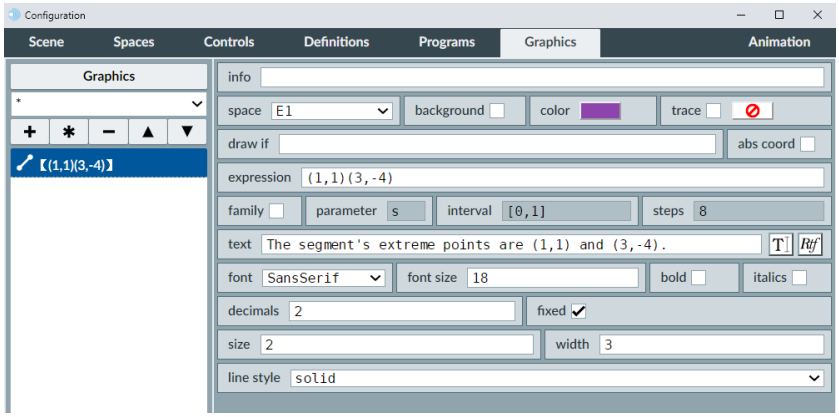


Figure 8.1.4.2: *Segment* graphic example configuration. The hex color used for the example is 8E44AD.

- **expression**: A text field in which the extreme points' coordinates are entered.
- **width**: A text field in which the segment's width in px is entered.

We now do a brief exercise to practice the use of segments. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Segment. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

Notice that, in this exercise, the segments used are set in absolute coordinates. That is the reason why they remain static even if the cartesian plane is moved around or zoomed.

8.1.5 *Polygon* graphic

This graphic object is drawn from the sequential coordinates of its individual vertices. Figure 8.1.5.1 shows an example of this type of graphic object. Figure 8.1.5.2 shows the configuration required to obtain the example.

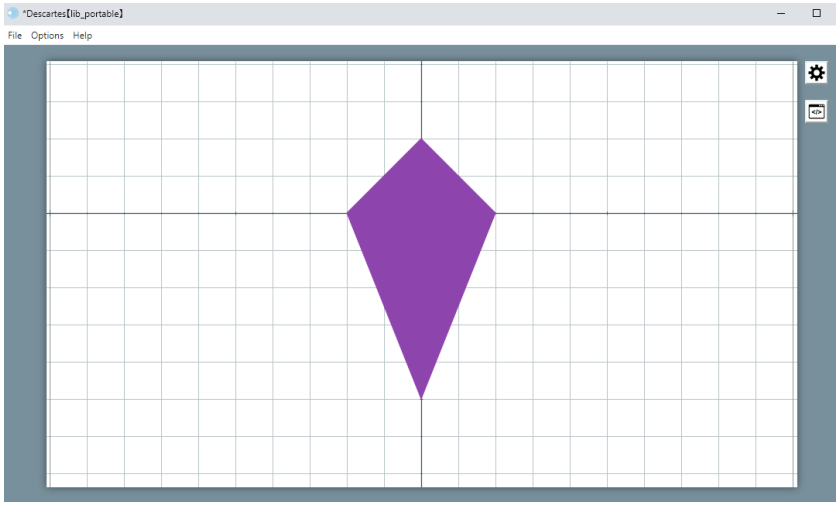


Figure 8.1.5.1: *Polygon* graphic object example.

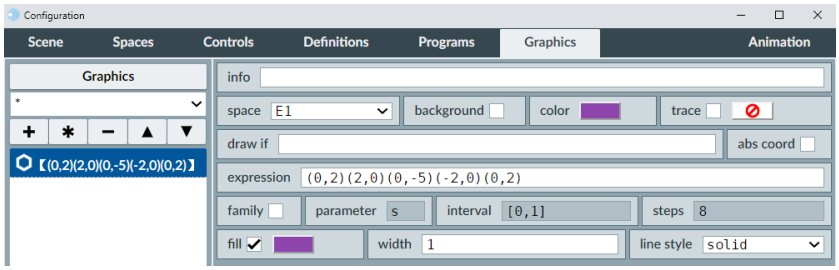


Figure 8.1.5.2: *Polygon* graphic example configuration. The hex color used for the example is 8E44AD.

- **expression**: A text field in which the coordinates of the polygon's vertices are entered sequentially. The sides of the polygon will be drawn as segments between the sequential vertices: from the first to the second, from the second to the third, and so on. Regardless of the implications of the *polygon* name, it can also be an open figure if the first and last vertices are not the same.
- **fill**: A checkbox that, when marked, allow the interior of the polygon to be filled with the color selected via the color editor button right of the checkbox. Particularly useful for closed polygons.

Let us do an exercise to practice the use of polygons. In this exercise, we will also include a second space which could be used as a means to provide the scene's user with feedback or instructions. This space is to be framed, and the frame will itself be the polygon. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Polygon. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

In this exercise, we also had a peek into some of *DescartesJS*' internal variables, as those associated with the width and height of spaces. As it was seen, knowing how to use them can come in handy. A more in-depth explanation of these variables can be found under the space variables topic.

8.1.6 *Rectangle* graphic

This graphic object allows the user to draw a rectangle by providing the coordinate of its top left corner, as well as the width and height of the rectangle. Figure 8.1.6.1 presents an example of this type of graphic object. Figure 8.1.6.2 shows the configuration required to obtain the example.

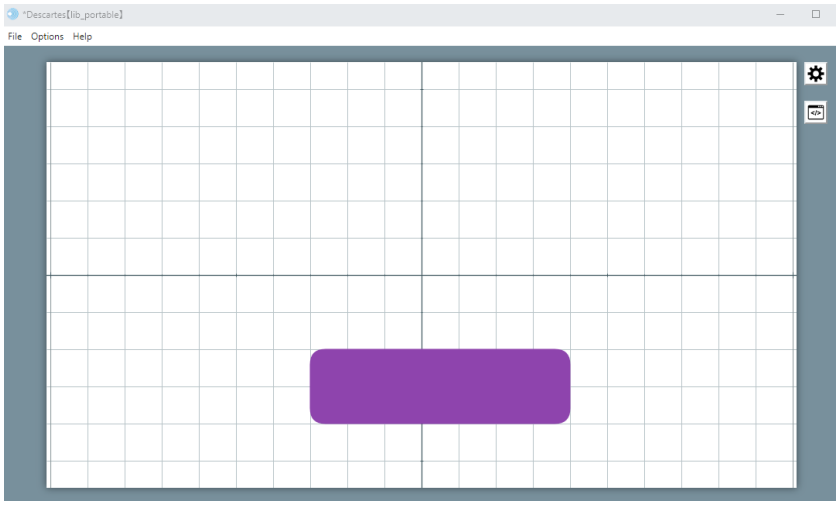


Figure 8.1.6.1: *Rectangle* graphic object example.

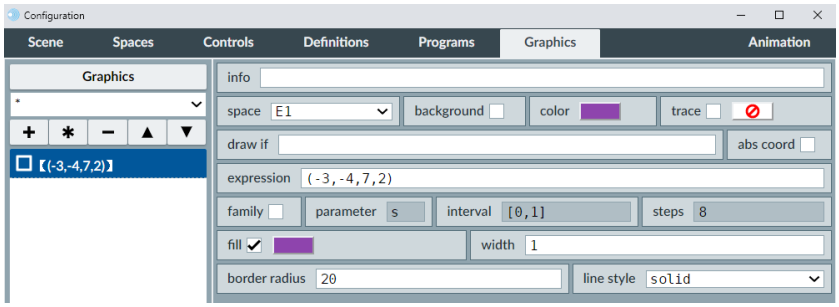


Figure 8.1.6.2: *Rectangle* graphic example configuration. The hex color used for the example is 8E44AD.

- **expression**: A text field in which the four coordinates are entered between a signle pair of parentheses and separated by commas: the x coordinate of the top left corner, its y coordinate, the width, and the length of the rectangle.

- **border radius**: When the corners are to be round, this text field indicates the number of pixels of the radius of an arc that will be the new corner.

Let us do an exercise to practice the use of rectangles. As with the polygon graphic's exercise, the rectangle here is to be used as a border for a space that could work as a panel to provide feedback or instructions to the scene's user. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Rectangle. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

We can see from this exercise that the rectangle has a more direct approach to drawing these objects than when using polygons.

8.1.7 *Arrow* graphic

This graphic object is very similar to the segment. It is also defined by a couple of coordinates: the beginning and the end (or spear) of the arrow, in that order. Figure 8.1.7.1 shows an example of this type of graphic object. Figure 8.1.7.2 presents the configuration required to obtain such an example.

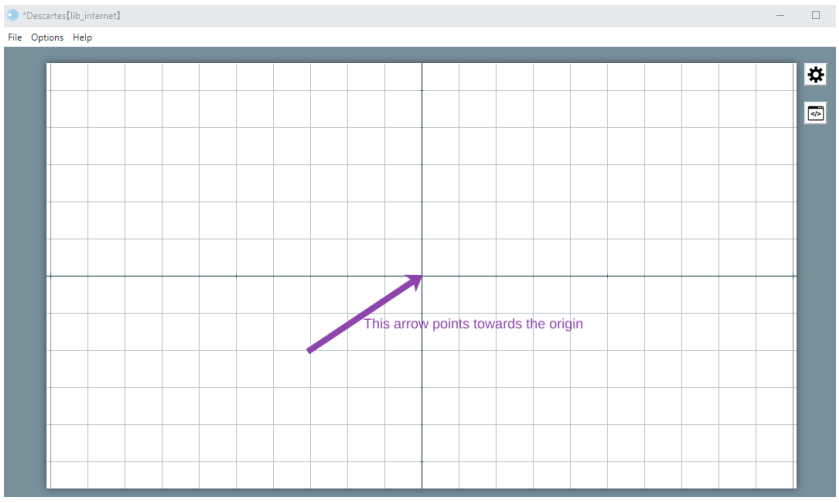


Figure 8.1.7.1: *Arrow* graphic object example.

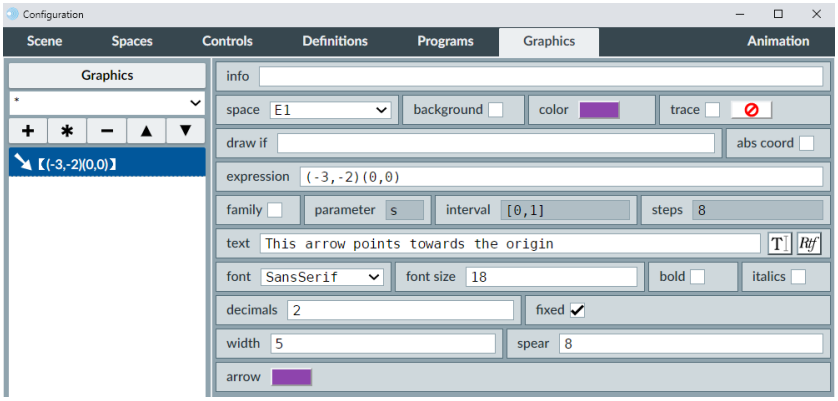


Figure 8.1.7.2: *Arrow* graphic example configuration. The hex color used for the example is 8E44AD.

- **expression**: A text field where the two coordinates are entered. As mentioned before, the last coordinate corresponds to the arrow's spear.
- **width**: A text field in which the width of the arrow's body is entered in px.
- **spear**: A text field in which the width of the arrow's spear is entered in px.
- **arrow**: A button that launches the color editor, so that the user can select the inner color of the arrow, since the *color* button at the top is related to its outer border.

Since the arrow graphic is very similar to the segment, no exercise is included for this type of graphic object.

8.1.8 *Arc* graphic

This graphic consists of a part of a circle's edge. The user only needs to provide the center, radius, starting angle, and ending angle of the arc to be drawn. Figure 8.1.8.1 presents an example of this type of graphic object. Figure 8.1.8.2 shows the configuration to get the example.

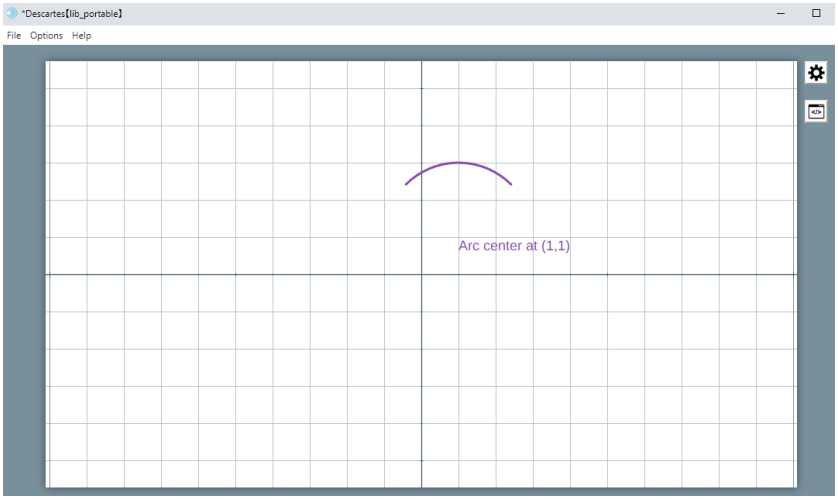


Figure 8.1.8.1: *Arc* graphic object example.

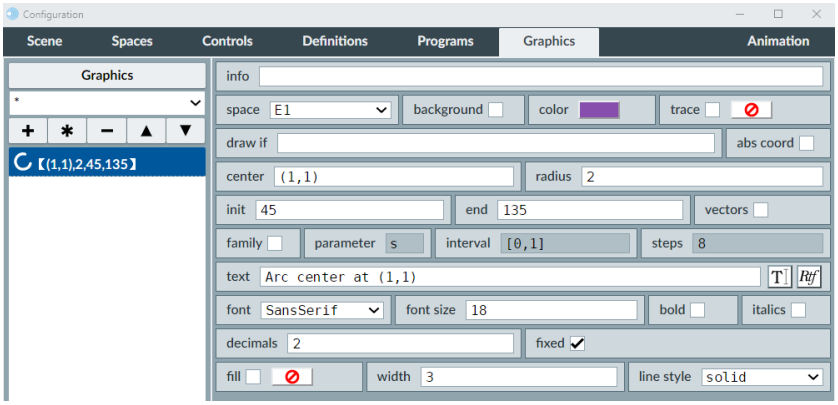


Figure 8.1.8.2: *Arc* graphic example configuration. The hex color used for the example is 8E44AD.

- **center**: A text field in which the coordinates of the center of the arc are entered. For example, (3,2) would imply the center is 3 units length to the right of the origin and 2 above it if the arc is set in coordinates relative to the cartesian plane.
- **radius**: A text field in which the arc's radius is entered.
- **init**: A text field in which to enter the initial angle in degrees. The initial or starting angle is the angle at which the arc starts being drawn.
- **end**: A text field in which the ending angle is entered in degrees. This angle is where the arc stops being drawn.
- **vectors**: A checkbox that, when marked, alters the interpretation of the *init* and *end* parameters. If it is marked, both these parameters should contain coordinates. Consider these vectors as drawn starting at the point defined by the *center* parameter,

and consider where they intersect the circle there centered an with a radius according to the *radius* parameter. The arc will be drawn as that part of the circumference subtended between those intersections. The vectors as such are not drawn, only the arc.

- **fill**: A checkbox that, when marked, allows the arc to be filled with a color. The color is selected by clicking the button at the right of the checkbox, which launches the color tool, via which the user can choose the fill color. Note that, if the arc's *vector* functionality is used (defining the arc via vectors instead of angles), the area filled is that contained between the vectors and the arc.

We do now an exercise to practice this functionality. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Arc. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJS-Documentation.zip* file.

A few things are worth mentioning regarding this exercise. An arc is a more immediate way to draw a part of a circumference, even if the same can be done using equation graphic object. The arc's *vector* functionality also allows to create an angle mark between lines if the vectors defining the lines are known. Additionally, if a circumference is to be drawn and its center and radius are known, it is better to draw it via the arc graphic object (a 360 degree arc) rather than via the equation one. It is actually easier for *DescartesJS* to draw it using an arc.

8.1.9 *Text* graphic

This graphic object prints text as well as variables' values. It can, for instance, display instructions for a given exercise, explanations, questions, etc. It can even be used by a scene's developer in order to print values for debugging purposes.

In previous *Descartes* versions, this graphic object could only be handled using absolute variables. Recent changes have endowed it with the possibility of using also relative ones. Most of text's functionality is explained in depth under the text editing tool topic. Figure 8.1.9.1 presents an example of a printed text. Figure 8.1.9.2 shows the configuration required to get this example. Figure 8.1.9.3 shows the *Rich text* edition window.

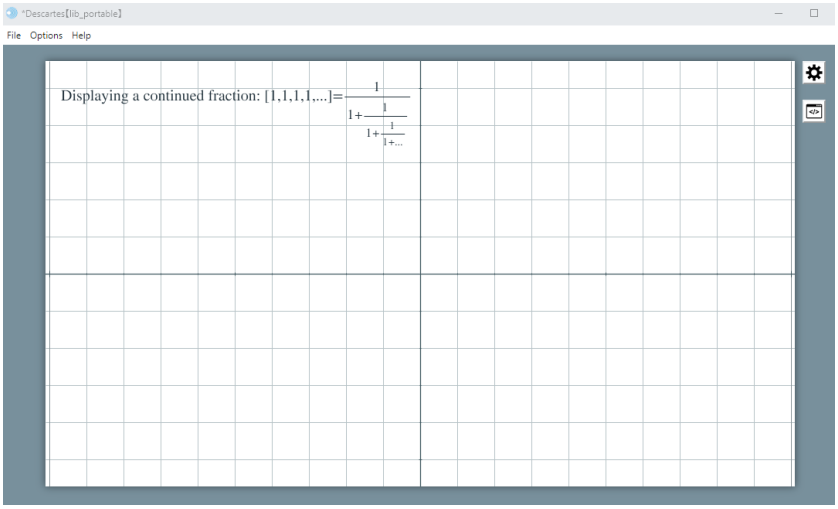


Figure 8.1.9.1: *Text* graphic object example.

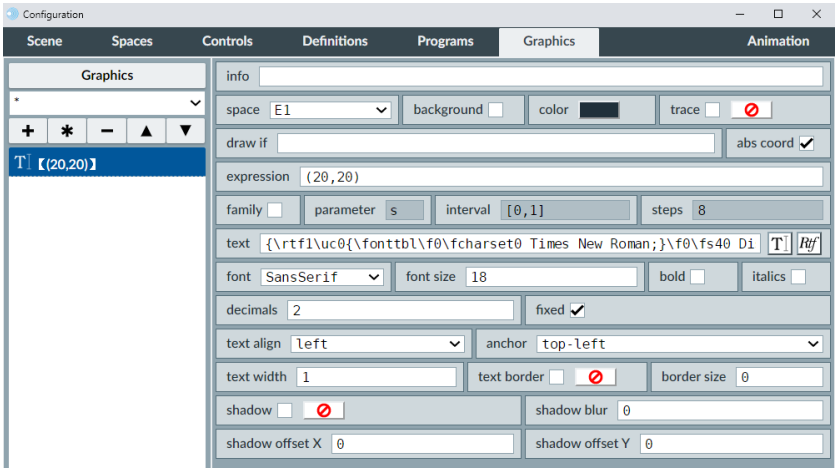


Figure 8.1.9.2: *Text* graphic example configuration. Note that *Rich text* is being implemented.

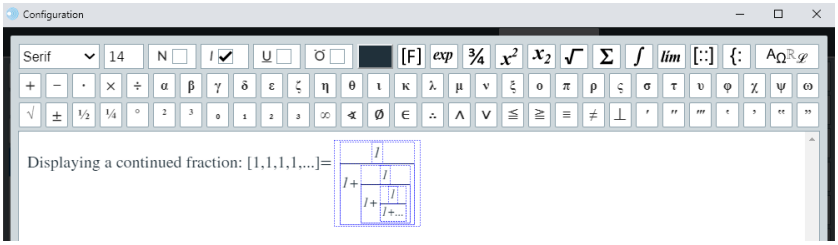


Figure 8.1.9.3: *Rich text* edition window that corresponds to Figure 8.1.9.2.

- **expression**: A text field where the coordinates of the text's anchor point are entered.

This anchor point can be placed in a position relative to the text via the *anchor* parameter described below.

- **text align**: A menu in which the the alignment of the text is selected for texts whose width exceeds a specified length (specified via the *text width* parameter described below) of px. If this happens, the text is wrapped around to another line, and the manner in which the lines are aligned is defined by the menu. The options are *left*, *center*, *right* and *justify*. This menu only makes sense when using *plain text*.
- **anchor**: A menu by which the text's anchor point position is set relative to the text itself. The text can be thought of as contained in an invisible rectangle. This menu determines which point of this rectangle is to be the anchor in the *expression* parameter (whether it is the *top left* corner of the rectangle, or the *top center* point at the middle of the rectangle's upper side, etc.).
- **text width**: A text field in which the allowed width of the text in px is entered. It only works when using *plain text*. This width, given in px, sets the maximum width the plain text may have before wrapping it around to a new line. When using *Rich text*, the line ends have to be determined by the user and manually entered. If the *text width* parameter value is less than 20, it is ignored as if no width limit is set. When using *Rich text*, this parameter should be set to 1.
- **text border**: When this checkbox is marked, a border is drawn around the text. The border's color can be selected via the color editor button right of the checkbox.
- **border size**: A text field where the text border's width (in px) is set. If the *text border* checkbox is marked and the the *border size* is left at its default 0 value, the text border's size will be automatically determined by *DescartesJS*.
- **shadow**: When this checkbox is marked, a shadow is drawn as if behind the text. The color of the shadow can be set using the color editor button right of the checkbox. When this checkbox is marked, the following parameters can be used to further edit the shadow's properties:
  - **shadow blur**: A text field in which the degree of blur is entered. A zero value is associated with a shadow with very sharp edges. Increasing the value results in more blurry shadows.
  - **shadow offset X**: A text field in which an integer is entered, representing the number of px the shadow will be horizontally placed relative to the text's position.
  - **shadow offset Y**: A text field in which an integer is entered, representing the number of px the shadow will be vertically placed relative to the text's position.

  Note that, when both offsets are at their default zero value, the shadow is places as if it were right behind the text.

Let us do an exercise to practice the *text* graphic object. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Text. The inter-

active scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

8.1.10 *Image* graphic

This graphic object can be a *jpg, png, gif, svg* (scalable vector graph), or *webp* image. *webp* images work ok in all browsers; and in *Safari* they are only correctly displayed in the *macOS Big Sur* version and later. Figure 8.1.10.1 presents an example of an image. Figure 8.1.10.2 shows the configuration required to get this example. Note that, for these examples to be reproduced, the scene's *html* file has to be already saved, and the image used in this example should be stored in an folder labeled *images* placed at the same level where the interactive scene's *html* file is saved; and the image's name should be *1.png*. The image file is the one included in the exercise at the end of this topic.

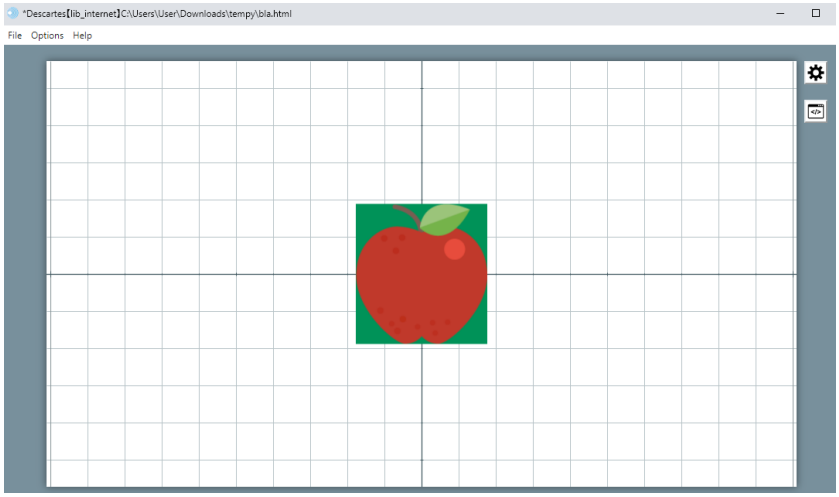


Figure 8.1.10.1: *Image* graphic object example.

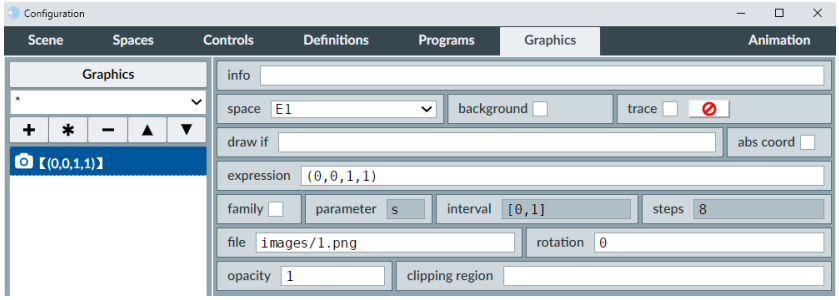


Figure 8.1.10.2: *Image* graphic example configuration.

- **expression**: A text field that contains the coordinates where the image is to be anchored. By default, the expression consists of two numbers separated by commas

and flanked by parentheses. These numbers correspond to the horizontal and vertical coordinates of the image's **top left corner**. However, four numbers can be also used (separated by commas as well). In this case, the first two numbers correspond to the horizontal and vertical coordinates of the image's **center**, and the last two numbers correspond the horizontal and vertical scale factors. If the scale factors are both 1, the size of image will be preserved. However, horizontal and scale factor values may differ, resulting in images that not necessarily preserve the image's original aspect ratio. Scale factors can also admit negative values. For instance, a horizontal scale factor of -2 will produce an image that horizontally is a mirror image of the original one, and has a width twice the size of the image's original width.

- **file**: A text field in which the image's path and name are entered. The path should be entered relative to the interactive scene's *html* file. the / character is used to separate folders. Folders above the scene's file (folders containing the folder in which the *html* file is stored) can be accessed using a double dot (..). For instance, *../image.png*.
- **rotation**: A text field in which the rotation angle (in degrees) is entered.
- **opacity**: A text field accepting a value between 0 and 1. A 0 value corresponds to minimum opacity (maximum transparency), while a 1 value (the default value) corresponds to a completely opaque image (null transparency).
- **clipping region**: A text field where an expression of the form `(x,y,width,height)` is entered. The user can use such an expression to allow only the clipped area of the image to be shown. x and y correspond to the clipped area's top left corner, while *width* and *height* correspond to the clipped area's width and height relative to the top left corner.
  For example, consider a 300px×150px image. If, both horizontally and vertically, only the central third of the image were to be shown (the central area of the image if it were in a 3×3 grid), the *clipping region* parameter would have to be set to `(100,50,100,50)`. This expression takes, horizontally, from 100px to 200 px (the middle third) and, vertically, from 50px to 100px (the middle third).

We are now ready for an exercise to practice the use of this graphic object. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Image. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise shows how to include images inside interactive scenes. The size and aspect ratio of these images can also be modified. Bear in mind that, though all these aspects are controlled by numbers in the exercise, variables can be used in their stead, so that the image can effectively be controlled dynamically.

Again, it is important to remember that, for an image to be displayed in an interactive, the interactive scene's *html* file has to be first saved with the path to the image set. It may be necessary to then reload the interactive scene, so that the path is refreshed and the image shown.

The image of a *DescartesJS* space: An image can be generated from whatever is viewed of a particular *DescartesJS* space. This is achieved by considering, as the image's name, the space's identifier with a *.image* suffix. Take, as an example, a space with an *E1* identifier. Over this space, consider an *E2* space covering it. *E1* has some graphic objects in it (curves, texts, etc.). If an *image* type graphic is created in *E2*, and its *file* parameter has `E1.image` in it, an image of what is viewed in *E1* is shown in *E2*. It follows all the rules laid out before (scaling, rotating, etc.). And, if the *E1* graphics change (for instance, if they are controlled via variables that change value), the image projected in *E2* will change as well. All this is particularly useful when, for instance, one needs a dynamic image made up of various graphic objects. And, if this image needs to repeat itself, the user can also use the *family* functionality of the image to create multiple instances of the *E1* space image.
In order to project a space as an image, this space should be below the space **in which** it is projected as an image (the projected space should appear above the space in which it is projected in the Spaces list panel at the left).
Additionally, even though graphics in the space being projected are displayed in the projected image, the space's background, border, etc. are not.

8.1.11 *Macro* graphic

A macro is a text file that includes parts of a *DescartesJS* scene, specifically from the *Definitions*, *Programs* and *Graphics* tabs. In order to generate such a file, one must first have a *DescartesJS* scene saved which includes the elements to be used in the macro.

Figure 8.1.11.1 presents an example of a macro in use. Figure 8.1.11.2 shows the configuration necessary to get this example. The macro used in this example can be found in a *macros* folder, which is stored at the same level as the scene's *html* file. The name of the text file is *macr.txt*. The arrow shown in the cartesian plane is **not** present in the *Graphics* tab of the scene; it is really being read from the macro.

Cabe notar dos cosas importantes. Aunque el macro viene incluido como un gráfico, realmente va más allá de solamente manipular gráficos. Recordemos que involucra también los elementos de *Definiciones* y *Programa*. Por otra parte, existe un macro en el selector *Gráficos* y otro en el selector *Gráficos 3D*. Los dos funcionan de manera muy similar, aunque hay diferencias leves en los parámetros de cada uno. Sólo se abordarán los macros en dos dimensiones en la presente documentación, dado que las diferencias entre ambos se explican solas.

- **file**: A text field which includes the path and name of the macro text file. The path is relative to the folder where the scene's *html* file is saved. Remember the / symbol is used to access folders below the scene's file folder (i.e., folder contained in that folder). For a macro to work, its text file has to be stored inside the folder containing the interactive scene's *html* file, or in a subfolder therein.
- **rotation**: A text field where a number or variable representing a rotation angle in degrees is entered. The macro will be rotated by such angle.

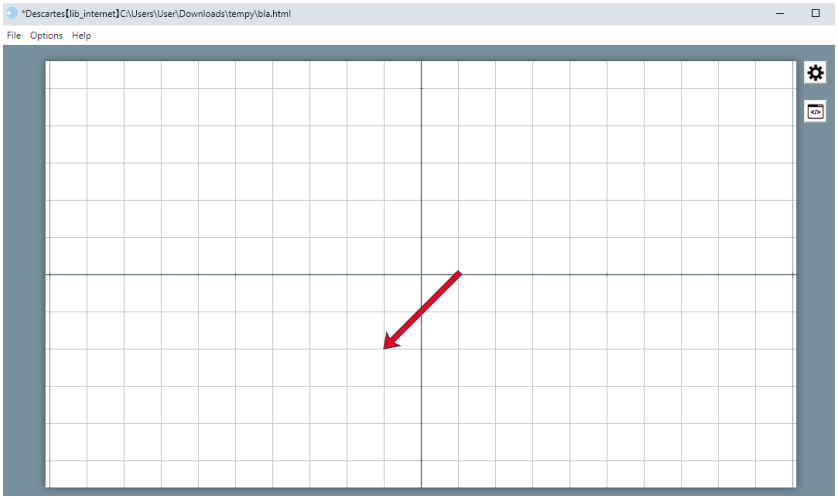


Figure 8.1.11.1: *Macro* graphic object example.

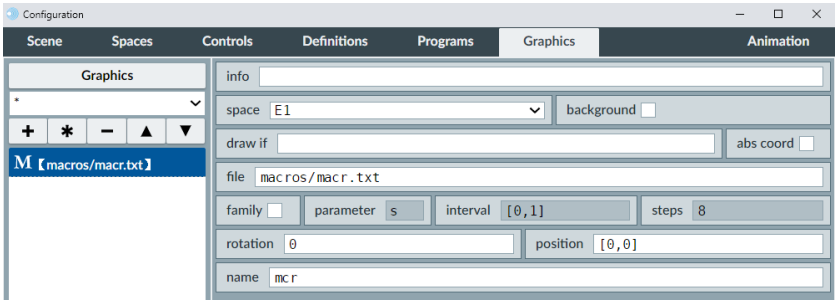


Figure 8.1.11.2: *Macro* graphic example configuration.

- **position**: A text field containing a coordinate. This coordinate will be where the macro's top left corner is to be placed. The pair of numbers is flanked by square brackets instead of parentheses.
- **name**: A text field for the name assigned to the macro. When a non graphic element (for instance, an element from the *Definitions* or *Programs* tabs) of the macro is to be called, its prefix should include the name of the macro followed by a period, and then the name of the definition on program element. For example, say you habe a macro named *mcr* which has a *distance()* function in it. If such a function is to be called in the main scene that contains the macro, it should be called as *mcr.distance()*.

The *File* menu in the main *DescartesJS* editor includes an *Export* option. It has a *Descartes macro* option in the submenu. When this option is selected, a pop-up window is launched in which the user can select the folder where the macro is to be saved and the name for it.

A macro can be used to simplify the design of subsequent scene's that all share certain features. For instance, say a same border design (comprised of various graphic objects) is

to be implemented in very many different scenes, each being saved as a different *html* file. The border design can then be exported as a macro. This macro can then be imported by each individual scene file. This means it is not necessary to repeat the code in each scene; only one copy of the code is stored in the macro, and all the scene's take it from there.

Additionally, it is possible to pass information from the main scene containing the macro down to variables included within the macro. This is achieved via assignments done in the main scene. If, for instance, the user wants to set a 2 value for a *wdth* variable in a macro named *macr* via the main scene, the assignment should be made as `macr.wdth=2`. The *wdth* value **within** the macro will then be 2.

We can now do an exercise to practice the use of the macro graphic object. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Macro. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise allows us to see the use of a macro as a sort of graphic template that can be used across of a multitude of different scenes, without needing to include the template's specific code in each. In this brief example, the template consists of only four triangles. But in a more professional template, it could rise up to 20 different graphic elements. It is certainly better to have only one copy of these elements in one place, instead of repeating this code resulting in files of larger size than needs be.

It is important to bear in mind that, in order for **graphic** macro elements to be displayed correctly in an importing scene, this last scene should have a space with the same name as that that housing the graphic elements inside the macro.

Note that, in the exercise, it was possible to control the value of a variable in the macro directly from the main scene. That is particularly useful to tweak aspects of the graphic elements' behavior without having to edit the macro itself.

It is sometimes necessary to manually edit a macro's text file. For example, when the elements exported were more than are really needed in the macro, the user can manually remove the surplus ones. It should be stressed, however, that such editions should be carried out using a text editor which allows to save in *UTF-8* coding, preferably without *BOM*. This will ensure that the macro will not have any hidden characters that would render it unreadable by *DescartesJS*. All this is addressed more in depth under the library definition topic.

8.1.12 *Sequence* graphic

This graphic object consists of a series of points that represent a mathematical sequence. A *DescartesJS* mathematical sequence requires a parameter that takes its values from a domain specified by the user. This parameter is used in the expression for the horizontal and vertical coordinates of the points. These coordinates are handled as ordered pairs.

Figure 8.1.12.1 presents an example of a sequence. Figure 8.1.12.2 shows the configuration required to set this example. Note that, in order to view all the point printed for the sequence, the cartesian plane has to be panned by clicking and dragging.

Figure 8.1.12.1: *Sequence* graphic object example.

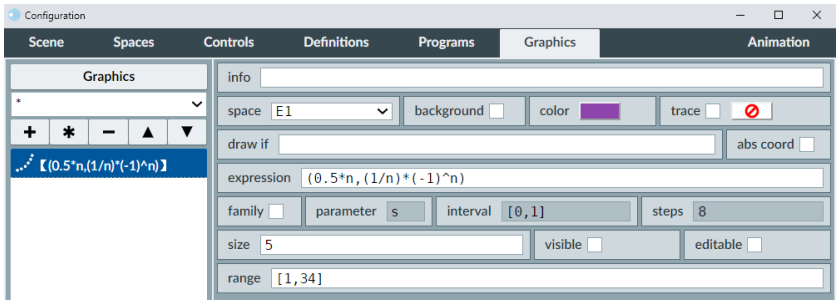


Figure 8.1.12.2: *Sequence* graphic example configuration. The hex color used for the example is 8E44AD.

- **expression**: A text field with the sequence's expression as an ordered pair. Its default value is $(n, 1/n)$. This means the first point (for which n=1) will be at $(1, 1/1)$, or $(1, 1)$. Its second point (for which n=2) will be at $(2, 1/2)$, or $(2, 0.5)$. And so on. Note that the values for n are taken from the *domain* parameter of the sequence, described below.
- **visible**: A checkbox which, when marked, makes the scene print out the sequence's correspondence rule (found in the sequence's *expression* parameter) inside the interactive. It is printed in the lower part of the space where the sequence is housed.
- **editable**: A checkbox which, when marked, allows the user to also modify the correspondence rule printed out when the *visible* checkbox is marked, thus enabling the sequence to change dynamically when the printed expression is modified.

- **domain**: A text field where the sequence's domain is entered. Its default value is [1,100], meaning that the values n will adopt are the integers from 1 to 100.

This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Sequence. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

8.1.13 *Fill* graphic

This graphic is a color that fills an area defined by bordering graphic objects.

Many graphic objects have their own *fill* parameter. However, sometimes it is necessary to color areas flanked by graphic objects, but that not necessarily belong to a specific graphic object fill. That is, areas *between* objects. Figure 8.1.13.1 presents an example of this graphic. Figure 8.1.13.2 shows the configuration required for this example. Note that the area to be filled is limited by a hyperbole with the equation $y^2 - x^2 = 1$. The reference point for the fill is the origin $(0,0)$, as stated in its *expression* parameter, so the search for the area to fill starts there.

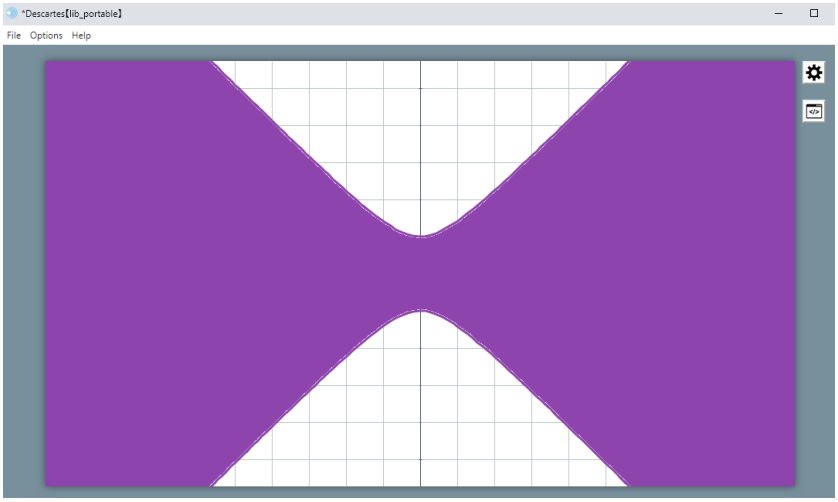


Figure 8.1.13.1: *Fill* graphic object example.

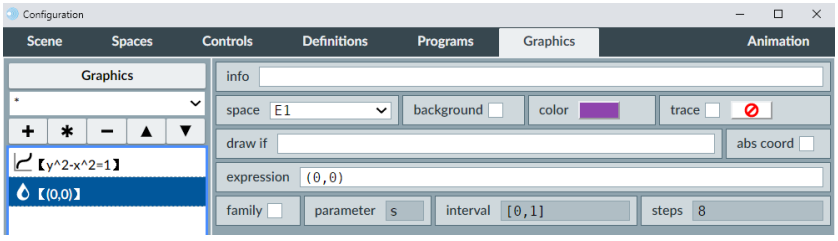


Figure 8.1.13.2: *Fill* graphic example configuration. The hex color used for the example is 8E44AD.

- **expresión**: A text field in which the reference point coordinates of the fill is set. This point lies inside the area to be filled.

Let us do an exercise involving various geometric shapes that cross each other, and how the *fill* graphic object can be used to color an area flanked by these shapes. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics Fill. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise shows us the usefulness of this type of graphic object to color areas made up of various shapes that may or not cross each other. You can think of the fill as paint being dropped on the point given by the fill's *expression* parameter, which then spreads until it finds a border it cannot cross. And it is particularly important to bear in mind that, for an object to effectively serve as a fill limit, it needs to be defined **before** the fill in the list of graphic objects, and has to also be drawn (the value of the object's *draw if* parameter has to be 1).

8.2 3D graphics

As expected, these graphics can only be drawn in 3D spaces. Since a default *DescartesJS* scene opens with only one 2D space, the user will need to add a 3D space in the *Spaces* to be able to draw the graphic objects described below. Additionally, given that many graphic objects are generic, or their configuration can be completely understood by means of the figures presented for them, only one exercise is included.

8.2.1 *Segment* graphic

It is a segment in three dimensions. Which means each extreme is a three dimensional coordinate.

Figure 8.2.1.1 presents an example of a three-dimensional segment. Figure 8.2.1.2 shows the configuration necessary for the example. In the configuration editor, only the last segment of three is seen (a segment along the z axis). However, the other two segments go along the other remaining coordinate axes (x and y). Each segment is 2 units length long. So, drawing these segments is a means to both locate the origin and know the space's orientation. We therefore include all these axis-representing segments in all the other 3D graphic objects' examples.

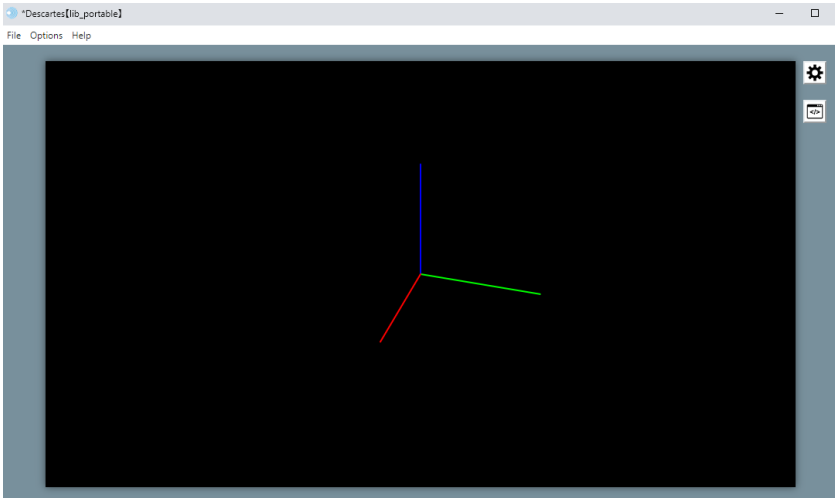


Figure 8.2.1.1: *3D Segment* graphic object example.

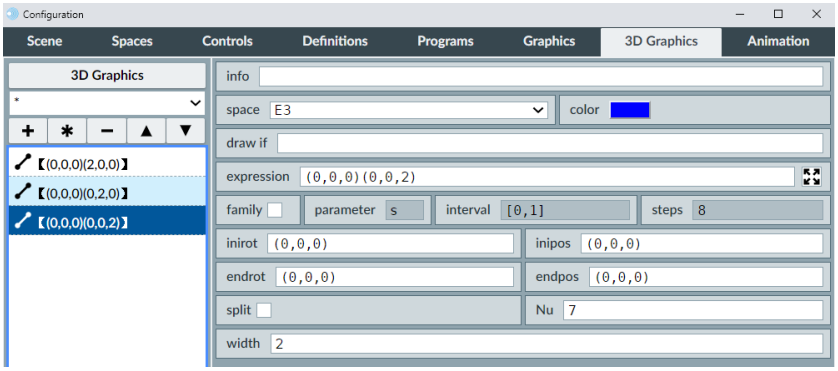


Figure 8.2.1.2: *3D Segment* graphic example configuration.

8.2.2 *Point* graphic

Three dimensional version of a point. Its *expression* parameter therefore admits a three dimensional coordinate.

Figure 8.2.2.1 presents an example of a three dimensional point. Figure 8.2.2.2 shows the configuration required for the example.

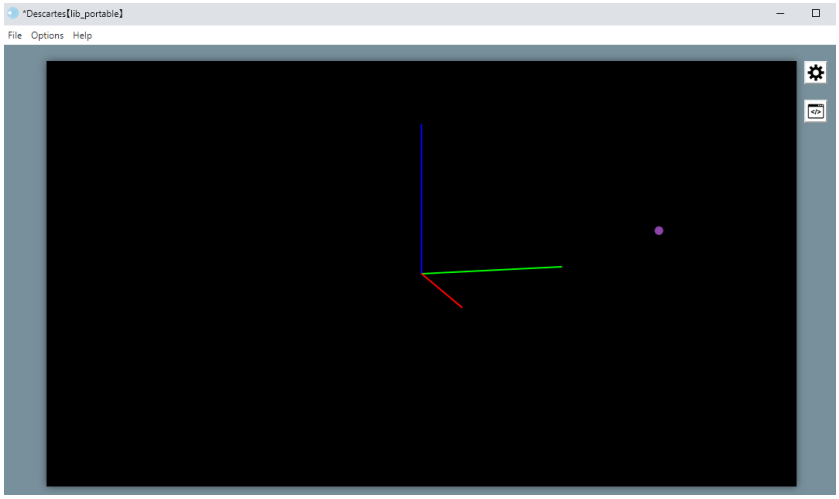


Figure 8.2.2.1: *3D Point* graphic object example.

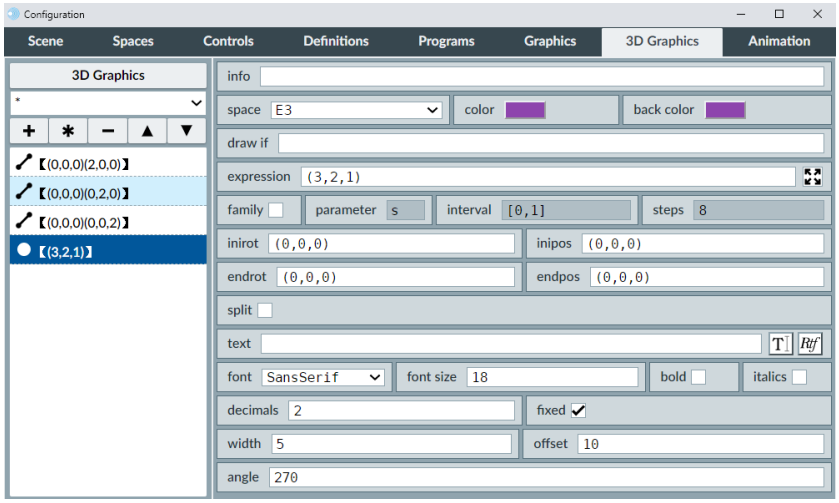


Figure 8.2.2.2: *3D Point* graphic example configuration. The hex color used for the example is 8E44AD.

- **offset**: A text field where the offset of a text accompanying a point is entered. This offset is the number of px of separation between the text and the point. The direction of this separation is entered in the *angle* parameter below.
- **angle**: A text field where the direction of the separation entered in the *offset* parameter is entered as an angle in degrees.

8.2.3 *Polygon* graphic

It is very similar to its two dimensional counterpart, except that the vertex coordinates (those provided in the *expression* parameter of the graphic) are, in this case, three dimen-

sional coordinates.

Figure 8.2.3.1 presents an example of a 3D polygon. Figure 8.2.3.2 shows the configuration necessary for the example. Note that, for polygons with more than 3 vertices, these need not lie in a same plane.

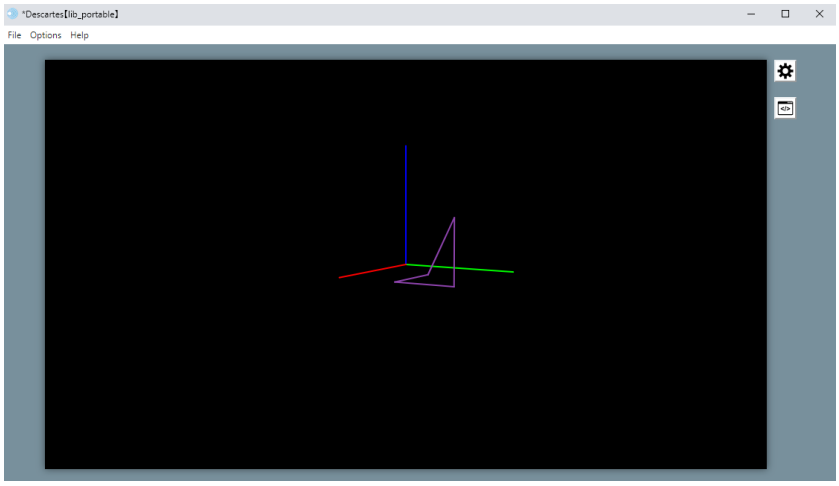


Figure 8.2.3.1: *3D Polygon* graphic object example.

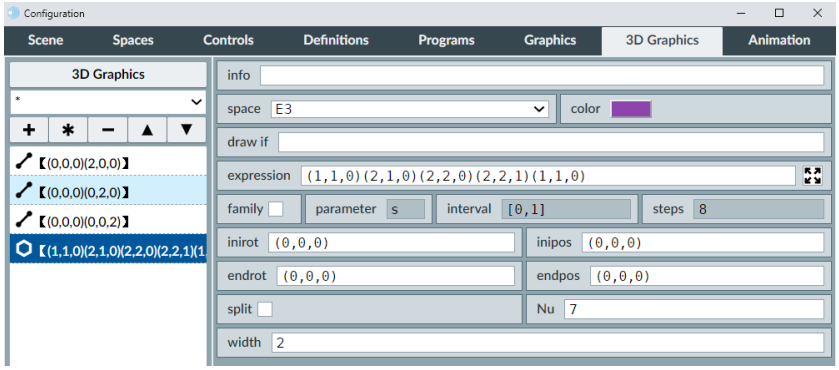


Figure 8.2.3.2: *3D Polygon* graphic example configuration. The hex color used is 8E44AD.

8.2.4 *Curve* graphic

This graphic is also very similar to its 2D counterpart. However, the *expression* parameter includes three assignments: one for x, another for y and another for z, each separated by the next by a space. The parameter of the curve is u, with a value range in $[0,1]$.

Figure 8.2.4.1 presents an example of the 3D curve. Figure 8.2.4.2 shows the configuration required for the example.

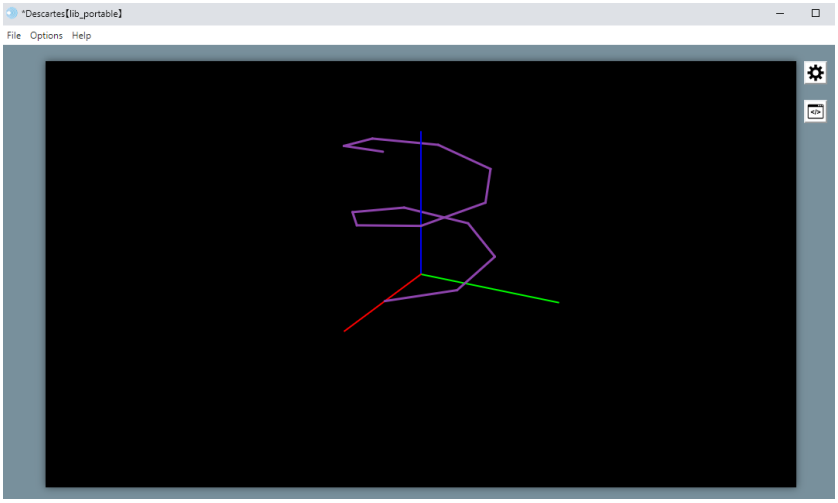


Figure 8.2.4.1: *3D Curve* graphic object example.

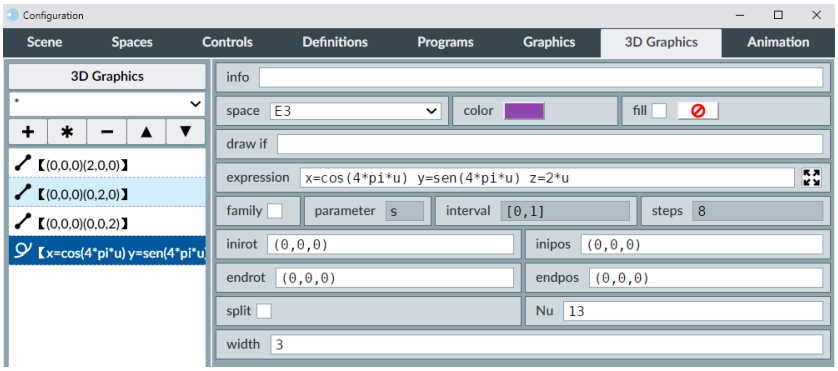


Figure 8.2.4.2: *3D Curve* example configuration. The hex color used is 8E44AD.

The configuration of a curve requires a value for a *Nu* parameter: the number of smaller intervals in which the $[0,1]$ interval for the u parameter is broken into. A small value will result in curves made up of segments. A larger value will result in a better resolution for the curve. However, excessively large values may result in a slow rendering of the interactive scene.

8.2.5 *Triangle* graphic

This graphic object is basically a three sided polygon. Its *expression* parameter contains three coordinates, one for each vertex. Each coordinate is, as expected, a three dimensional coordinate.

Figure 8.2.5.1 presents an example of a 3D triangle. Figure 8.2.5.2 shows the configuration required for this example.

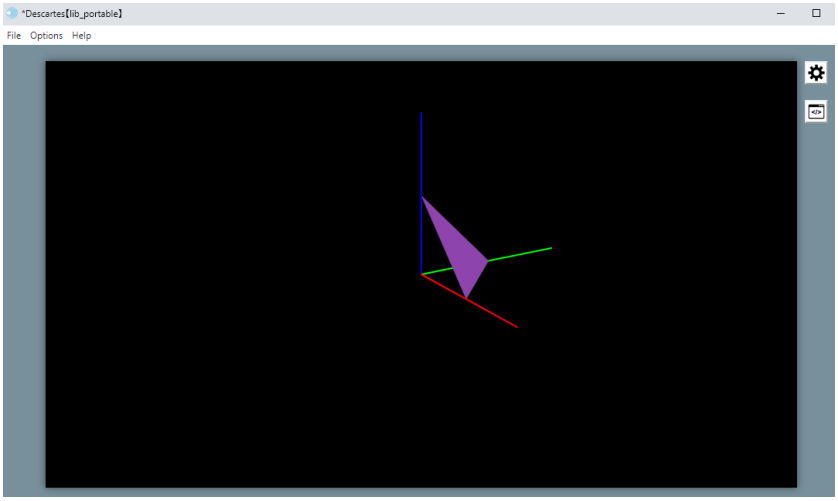


Figure 8.2.5.1: *3D Triangle* graphic object example.

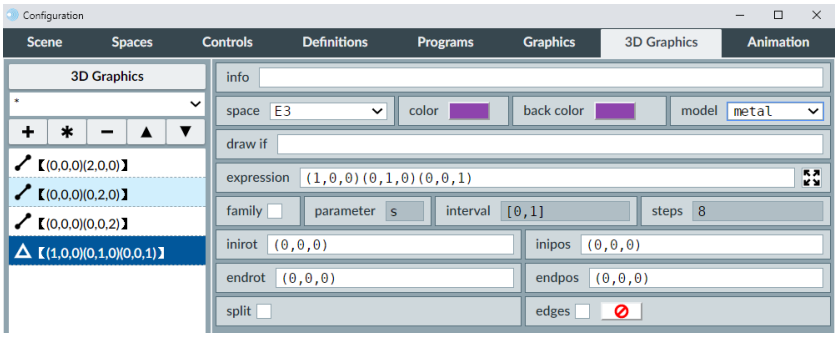


Figure 8.2.5.2: *3D Triangle* graphic example configuration. The hex color used is 8E44AD.

8.2.6 *Face* graphic

This graphic is basically a 2D polygon that can then be rotated in a 3D space. Since it is a 2D polygon, its *expression* parameter consists of a sequence of two dimensional coordinates (ordered pairs), as many as the number of vertices the face has. After its shape has been defined, the *inirot*, *inipos*, *endrot*, and *endpos* parameters can be used to rotate it and shift its position. If all these parameters keep their 0 default value, the face lies in the *xy* plane, as expected. The other rotation and shift parameters are dealt with in depth in the common 3D graphic parameters topic.

It is also possible to enter three dimensional coordinates for the vertices in the *expression* parameter. The face can then be drawn ignoring the shift and rotation parameters. However, these vertices must then be coplanar (they should all lie in the same plane). If this condition is not met, the face will not be drawn as expected, since this graphic object is not really three dimensional (it is a two dimensional object that lives in a 3D space).

Figure 8.2.6.1 presents a face graphic example. Figure 8.2.6.2 shows the configuration necessary for this example. In this example, the first approach was taken to draw the face (i. e., the two dimensional coordinates for the vertices). Note that there is non zero rotation present for the graphic in the *inirot* parameter.

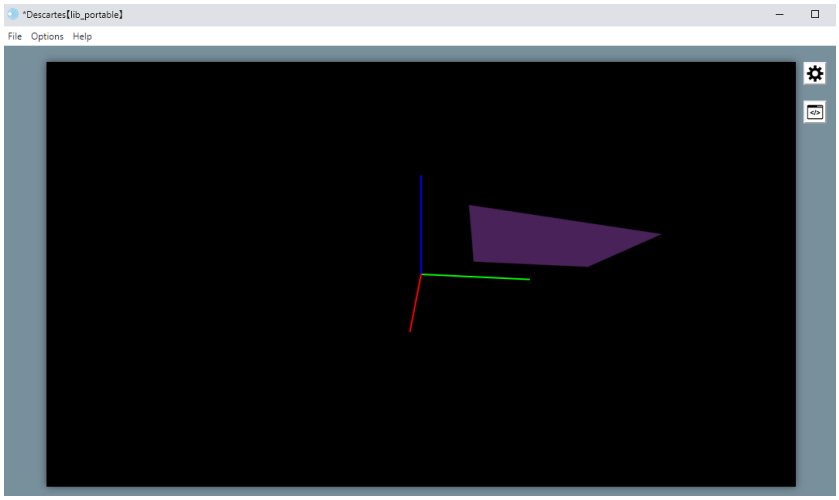


Figure 8.2.6.1: *3D Face* graphic object example.

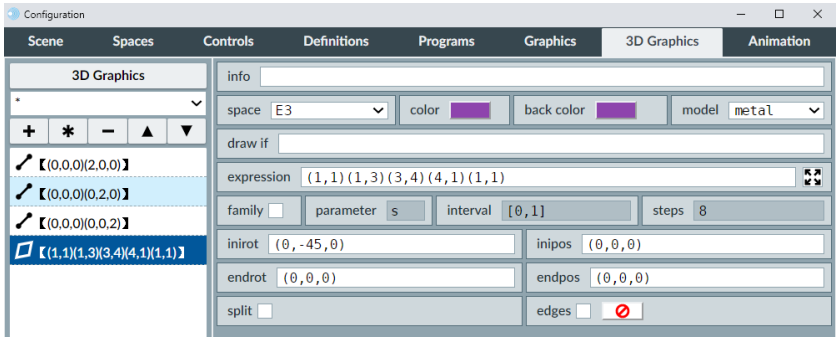


Figure 8.2.6.2: *3D Face* graphic example configuration. The hex color used is 8E44AD.

8.2.7 *Regular Polygon* graphic

This graphic object defines a regular polygon with a given number of sides. It is drawn by default in the *xy* plane, centered in its origin, and its very first vertex on the *x* axis. It can nevertheless then be shifted and rotated in 3D. The parameters used to shift and rotate it are discussed in depth in the common 3D graphic parameters topic.

Figure 8.2.7.1 presents an example of a 3D regular polygon. Figure 8.2.7.2 shows the example's required configuration. Note that the number of sides of the polygon is determined by the *Nu* parameter. Also note that the *inirot* and *inipos* are used to rotate and shift the polygon, so it leaves its default *xy* plane.

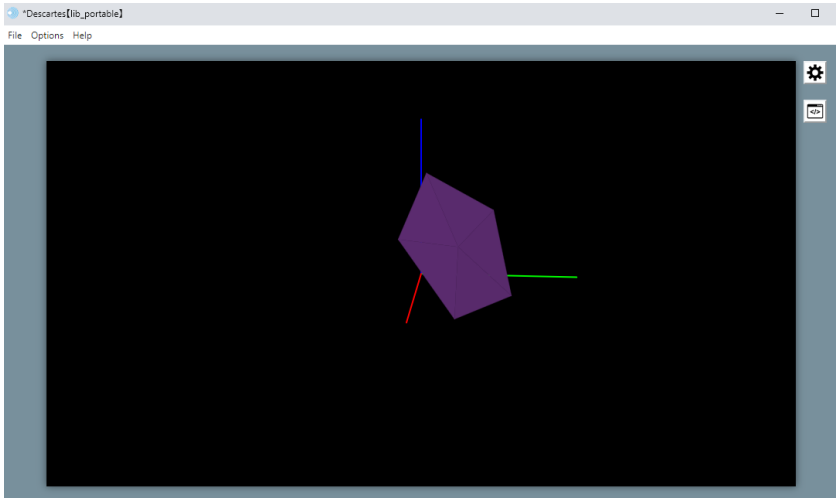


Figure 8.2.7.1: *3D Regular Polygon* graphic object example.

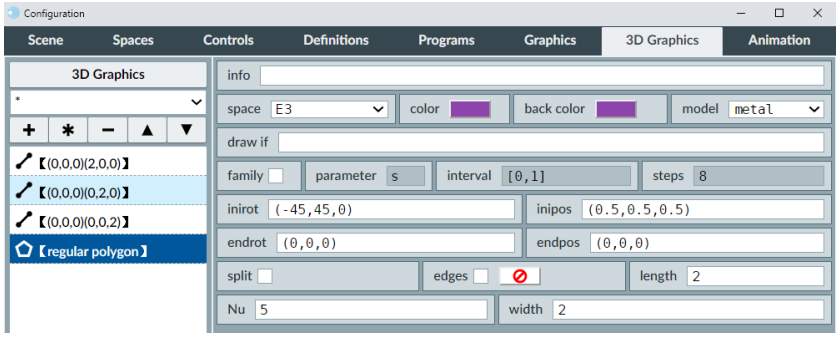


Figure 8.2.7.2: *3D Regular Polygon* graphic example configuration. The hex color used is 8E44AD.

8.2.8 *Surface* graphic

This graphic object consists of a surface drawn using two parameters: u and v, each of which draws its values from the $[0,1]$ interval. The *Nu* parameter is used to set the number of partitions of the $[0,1]$ interval for the u parameter, whereas *Nv* is used for the v parameter.

The *expression* parameter of a surface is a text field with an assignment for x, one for y, and one for z; each separated by a space. These may use the u and v variables that parametrize the surface.

Figure 8.2.8.1 presents an example of a surface. Figure 8.2.8.2 shows the examples required configuration.

Note that the assignments in this example's *expression* parameter use the u and v parameters for x and y. z is then defined as a function of x and y. $x = 3\cos(2\pi u)\sin(\frac{\pi}{2}v)$,

$y = 3\sin(2\pi u)\sin(\frac{\pi}{2}v)$, y $z = \cos(3\sqrt{x^2+y^2})$. x and y are defined by means of cylindrical coordinates. This endows the surface with a round edge. The square root of the sum of the square of the x and y values in the assignment for z involves a distance from from the z axis. And, since it is the argument of a cosine function, this configuration altogether gives the surface a wavelike nature.

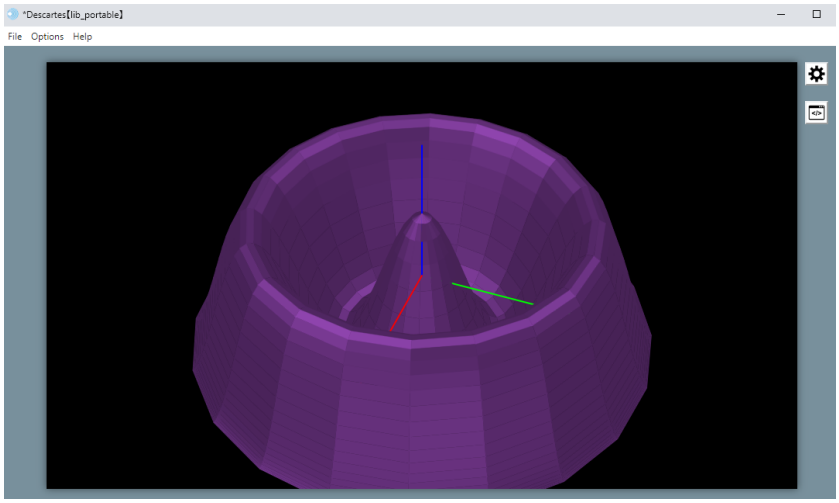


Figure 8.2.8.1: *3D Surface* graphic object example.

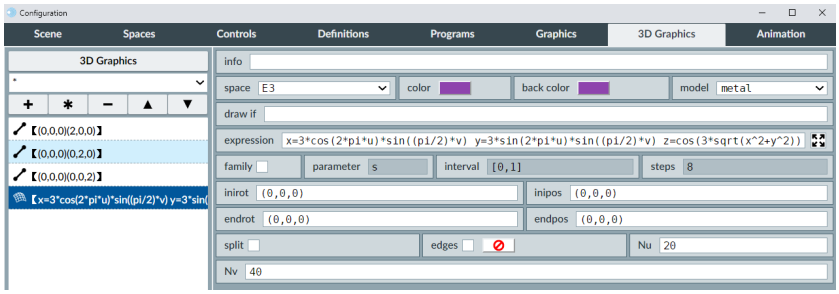


Figure 8.2.8.2: *3D Surface* graphic example configuration. The hex color used is 8E44AD.

8.2.9 *Text* graphic

It works in very much the same way as its 2D counterpart. However, in 3D, texts are not allowed to leave traces. That is the reason for the missing *trace* checkbox. Also, since these type of texts are not housed in a 2D space, their position cannot be defined relative to a cartesian plane. They therefore only use absolute coordinates, which is the reason for the missing *abs coord* checkbox in 3D texts.

8.2.10 *Macro* graphic

This graphic object is very similar to the 2D macro. A difference, though, is that the text field where the path and name to the macro text file is the *expression* parameter, instead of the *file* one in 2D.

Please note that, in order for a macro to display its graphic objects correctly in 3D, the scene importing the macro has to have a 3D space with the same identifier as the one housing the macro's graphic objects. For instance, if the macro's graphic objects were generated in a 3D space with an *E2* identifier, when the macro is imported in another scene, it should also have a space with the *E2* identifier. Otherwise, the macro's graphics will not be displayed.

8.2.11 *Cube* graphic

The cube graphic object is defined by the *width* parameter, which is the length of an inner diagonal (the segment spanned by two completely opposing vertices of the cube). Once the width of the cube is given, the cube's side can be calculated dividing the width by $\sqrt{3}$. It can also be reoriented and shifted using the shift and rotation parameters described in the common 3D graphic parameters.

Figure 8.2.11.1 presents an example of a cube. Figure 8.2.11.2 shows the example's required configuration. Note that the example involves a rotated cube, since the *inirot* parameter does not have its default value.

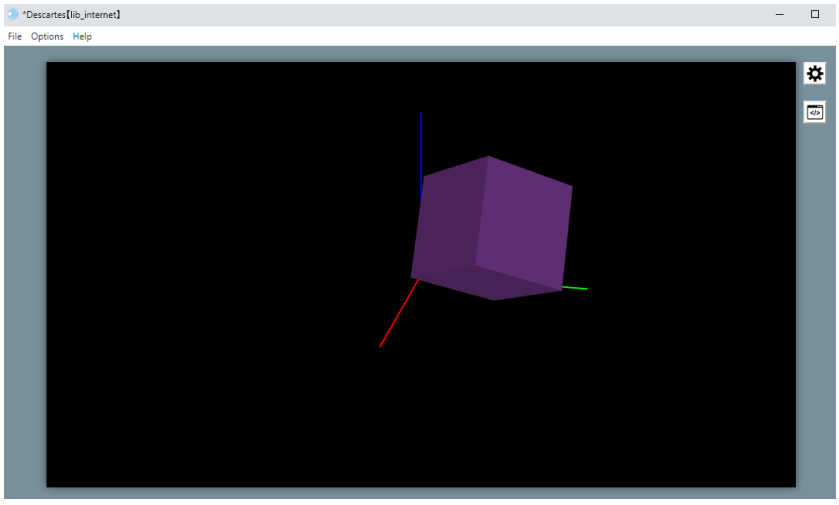


Figure 8.2.11.1: *Cube* graphic object example.

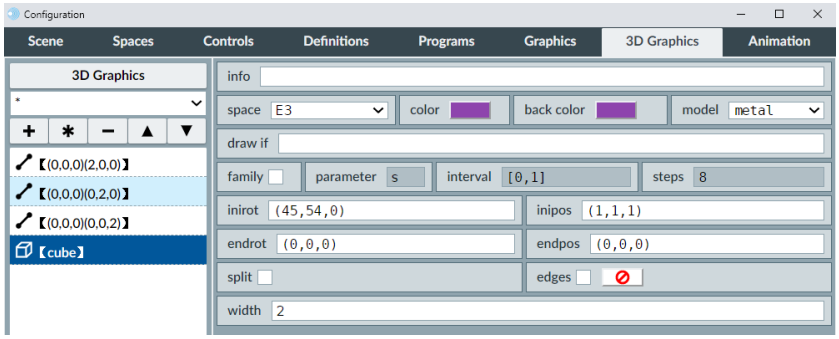


Figure 8.2.11.2: *Cube* graphic example configuration. The hex color used is 8E44AD.

8.2.12 *Box* graphic

Similar to the cube, but with the exception that all three defining sides of this object can be defined separately. These are defined via the *length*, *height* and *width* parameters.

Figure 8.2.12.1 presents an example of a box graphic object. Figure 8.2.12.2 shows the configuration necessary for the example. Note that the box in this example is rotated, since the *inirot* parameter does not have its default value.

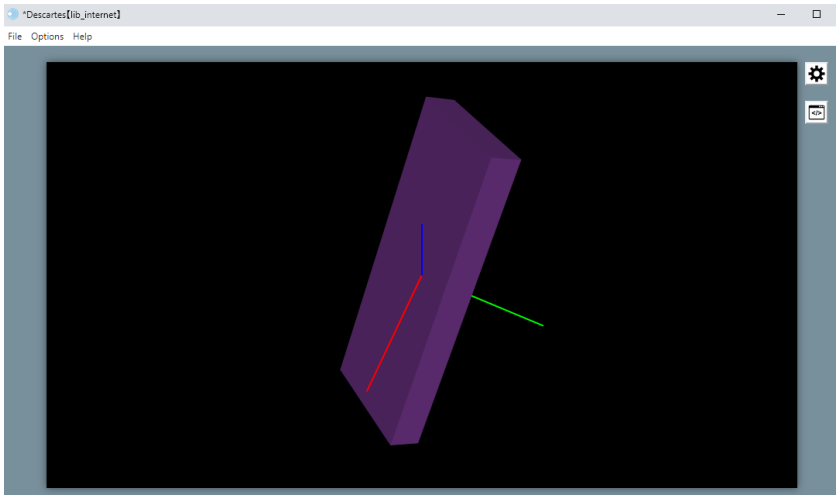


Figure 8.2.12.1: *Box* graphic object example.

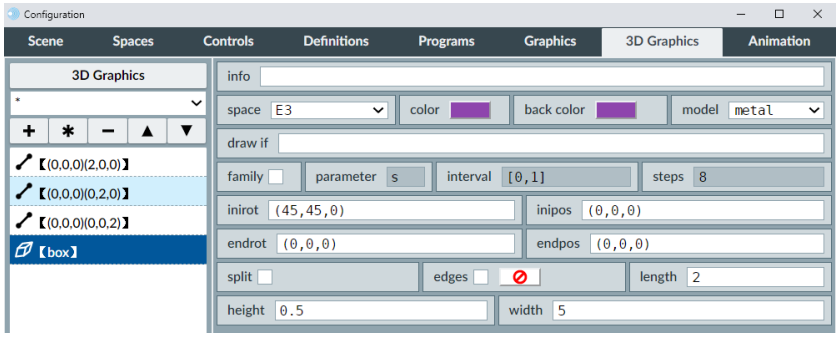


Figure 8.2.12.2: *Box* graphic example configuration. The hex color used is 8E44AD.

8.2.13 *Tetrahedron, Octahedron, Dodecahedron* and *Icosahedron*

These graphics are pre-designed polyhedrons. The only unique parameter they have is their *width* one, which determines their sizes. As with the cube, the width is the length spanned by two opposing vertices of the polyhedron in question. Once defined, its placement can be changed via the shifting and rotation parameters discussed in the common 3D graphic parameters topic.

Figure 8.2.13.1 presents an icosahedron as an example of a polyhedron. Figure 8.2.13.2 shows the configuration required for this example. Note that the object does not appear centered in the origin, since the *inipos* parameter has been changed from its default value.

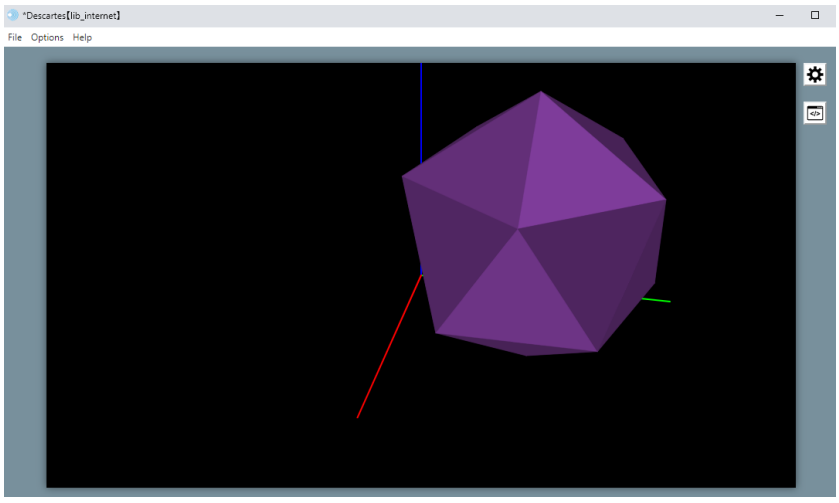


Figure 8.2.13.1: *Icosahedron* graphic object example.

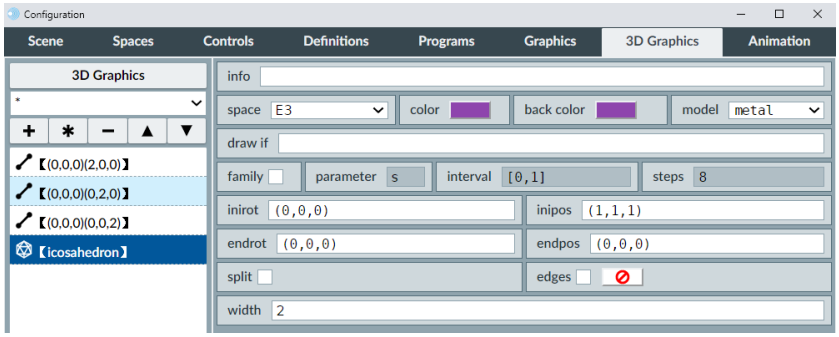


Figure 8.2.13.2: *Icosahedron* graphic example configuration. The hex color used is 8E44AD.

8.2.14 *Sphere* graphic

The sphere is defined by the *width* parameter, in which its diameter is entered. The sphere is rendered depending on the number of meridians and parallels assigned to it. The number of parallels is specified in the *Nu* parameter, whereas the number of meridians in the *Nv* parameter. The higher their values, the more the graphic will resemble a true sphere. However, very high values for these parameters may result in a long time required to render and refresh the scene.

Figure 8.2.14.1 presents an example of a sphere. Figure 8.2.14.2 shows the configuration required for the example.

Note that the parallels and meridians of the sphere in the example have a higher contrast and can be easily identified. This responds to the mark placed on the *edges* checkbox. Note also that the sphere is set to have 7 meridians and 15 parallels. Furthermore, the sphere seems tilted (its axis does not lie along the z axis). This is because the *inirot* parameter includes a rotation using Euler angles. The shifting and rotation parameters are discussed in depth in the common 3D graphic parameters topic.

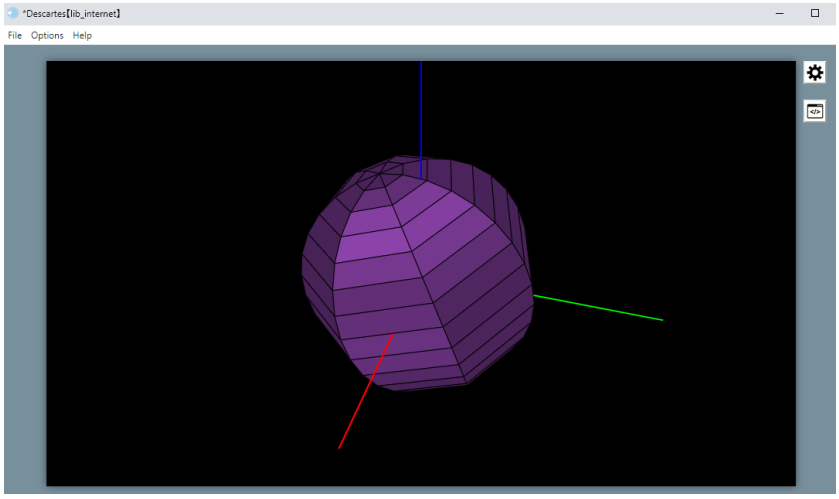


Figure 8.2.14.1: *Sphere* graphic object example.

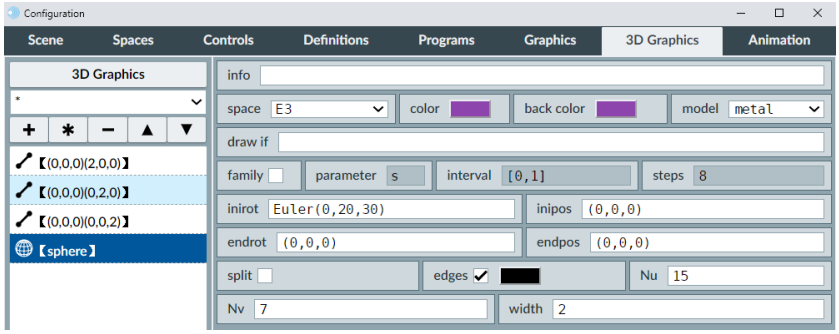


Figure 8.2.14.2: *Sphere* graphic example configuration. The hex color used is 8E44AD.

8.2.15 *Ellipsoid* graphic

The ellipsoid is defined by three parameters: *length*, *height* and *width*. In a sense, it can be thought of as a an ellipsoid constrained outside by a 3D *box* graphic. The ellipsoid's resolution is controlled by the *Nu* parameter, which corresponds to the number of parallels involved, and the *Nv* parameter, which controls the number of meridians.

Figure 8.2.15.1 presents an example of the ellipsoid. Figure 8.2.15.2 shows the configuration required for the example. Note that a certain amount of transparency (77 in hex) has been added to the graphic object's color. Additionaly, white edges are implemented via the *edges* checkbox, so as to stress the effect of the *Nu* and *Nv* parameters.

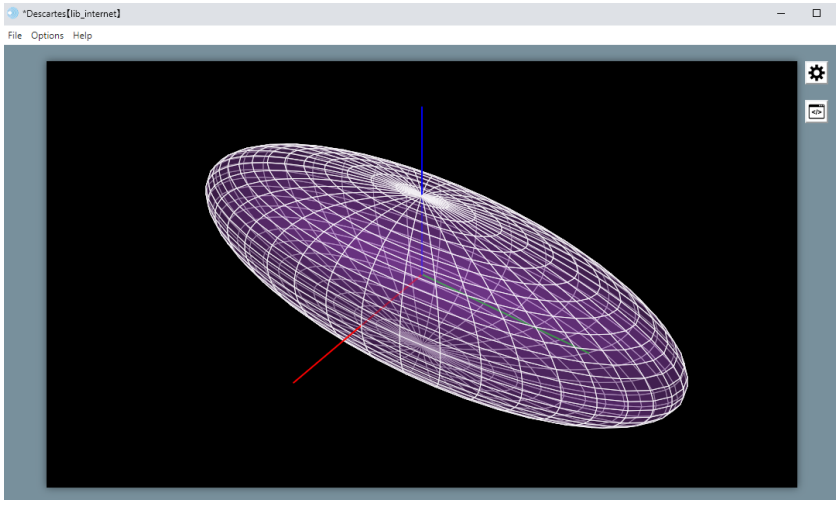


Figure 8.2.15.1: *Ellipsoid* graphic object example.

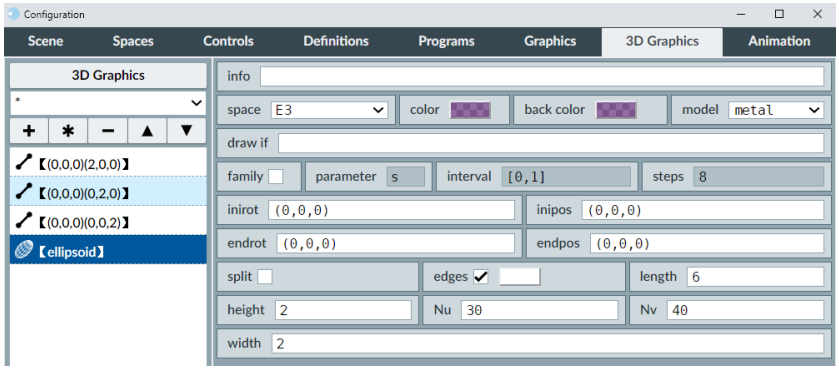


Figure 8.2.15.2: *Ellipsoid* graphic example configuration. The hex color used is 8E44AD, with a hex transparency of 77.

8.2.16 *Cone* graphic

A cone shaped graphic whose base can be an ellipse or circle. By default, its vertex points downward (towards the negative z axis), and its center is at the origin. The ellipse base is defined by the *width* parameter (along the x axis by default) and the *length* one (along the y axis by default). The *height* parameter contains the distance from the base to the cone's vertex (along the z axis). As with the *sphere* graphic object, it also has a *Nu* parameter, related to the number of "parallels", and a *Nv* one related to the "meridians".

Figure 8.2.16.1 presents an example of a cone. Figure 8.2.16.2 shows the configuration necessary for the example.

Note the *edges* checkbox of the graphic is marked and set to draw them white, to better view the effect of the *Nu* and *Nv* parameters. Also, the number of parallels is set to 10

(10 edges dividing the cone's height), and the number of meridians is set to 14. As seen in the *inirot* parameter, the cone is rotated half a turn (180°) around the x axis in order for it to appear base down. The cone's center has additionally been shifted 1 unit towards the positive x axis and 1 unit towards the positive z axis, which can be seen in the *inipos* parameter.

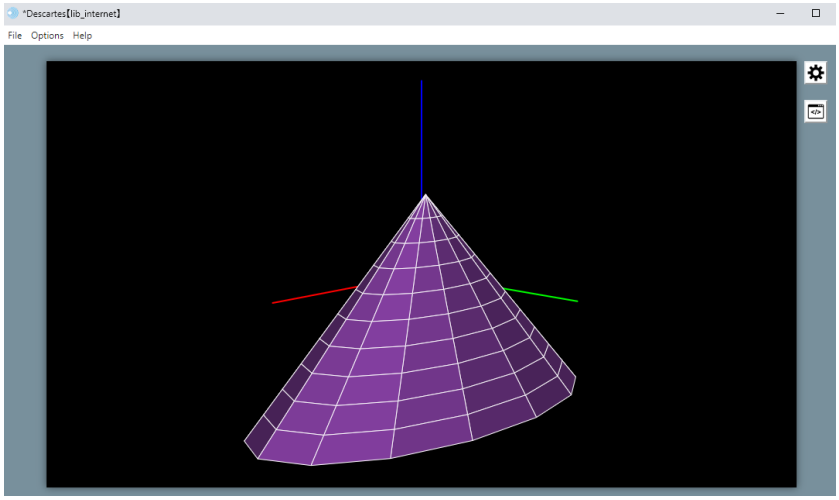


Figure 8.2.16.1: *Cone* graphic object example.

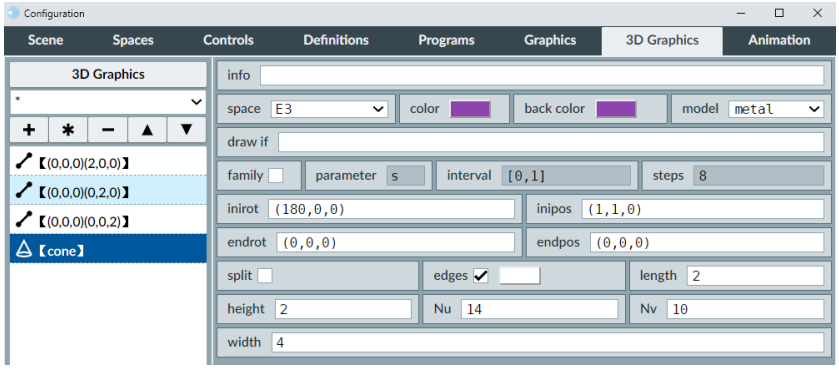


Figure 8.2.16.2: *Cone* graphic example configuration. The hex color used is 8E44AD.

8.2.17 *Cylinder* graphic

A cylindrical whose perpendicular cross section can be an ellipse. By default, it appears with its center at the origin and both its bases parallel to the xy plane. It includes a *width* parameter in which is used to enter its thickness along the x axis. The *length* parameter is used to enter its thickness along the y axis, and the *height* parameter is used to enter its thickness along the z axis. As with for other graphics, the *Nu* parameter holds the number of "meridians" of the graphic object (in a sense, the number of partitions into which the

angular parameter of the figure is broken). The *Nv* parameter specifies the number of "parallels" (or subdivisions in the *z* axis direction) that are used when drawing it.

Figure 8.2.17.1 presents an example of a cylinder. Figure 8.2.17.2 shows the configuration required for the example.

Note the *edges* checkbox is marked, and that the color chosen for them is white, to better display the role of the *Nu* and *Nv* parameters. Also, the object has been rotated 45°around the *x* axis and another 45°around the *y* one (as can be seen in its *inirot* parameter). It therefore appears tilted. The color used has a little transparency to better locate it with regards to the coordinate axes.

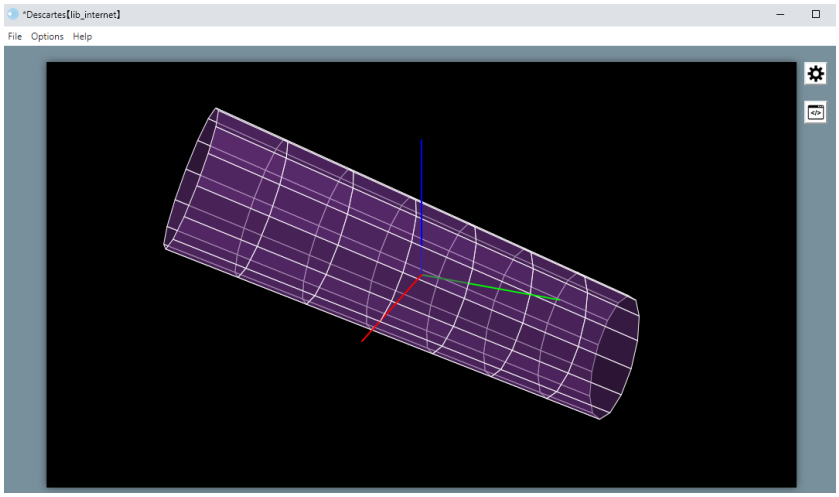


Figure 8.2.17.1: *Cylinder* graphic object example.

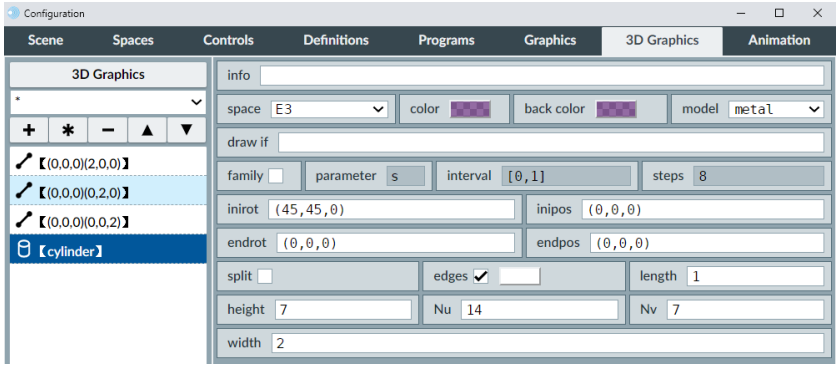


Figure 8.2.17.2: *Cylinder* graphic example configuration. The hex color used is 8E44AD, with a 77 hex value transparency.

8.2.18 *Torus* graphic

A torus is a doughnut shaped figure. It can be seen as the solid of revolution obtained by rotating a circle around an axis also in the circle's plane. The *R* parameter of the torus indicates the radius of rotation (the distance between the circle's center and the axis of rotation). The *r* parameter is the radius of the circle. The *Nu* parameter indicates the number of subdivisions into which the rotation parametrization angle (the one related to the rotation of the circle around the axis) is to be broken. The *Nv* parameter indicates the number of subdivisions into which the circle's angle is to be broken. Once again, higher values for these parameters result in a smoother surface, but may involve slower rendering.

Figure 8.2.18.1 presents an example of a torus. Figure 8.2.18.2 shows the configuration required for this example.

Note the *edges* checkbox is marked, and the custom *gray* color is used to draw them. Additionally, the color is considerably transparent, since we with to be able to see the axes behind the figure to better understand its placement.

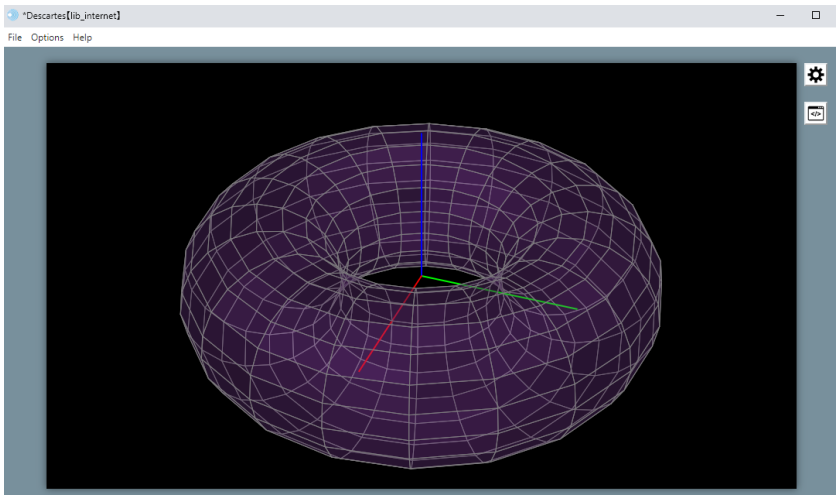


Figure 8.2.18.1: *Torus* graphic object example.

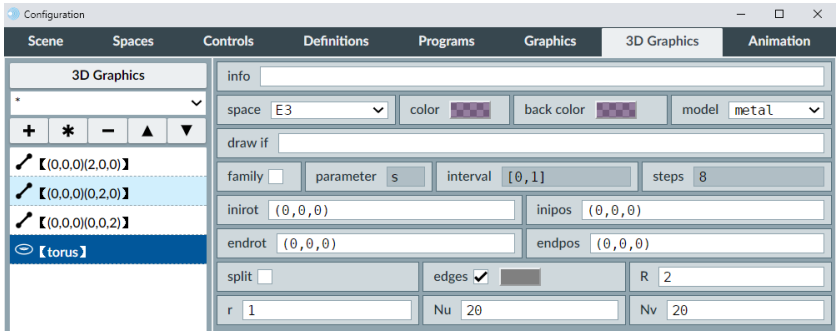


Figure 8.2.18.2: *Torus* graphic example configuration. The hex color value is 8E44AD, with an AF hex value for its transparency.

8.2.19 3D graphics general exercise

We should do an exercise representing the most general aspects of all 3D graphic objects. We will use the ellipsoid. This exercise's interactive scene, along with the instructions to build it, can be found at Graphics 3D. The interactive scene's document as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise shows the general functionality shared by most 3D graphics. The implementation of rotations and position shifts is noteworthy. Though some objects (a *3D face*, for instance) could be built by specifying their individual coordinates, it would take a considerable amount of time and effort which can be saved by using rotations and shifts. This exercise shows also the usefulness of having an initial rotation, followed by an initial position shift, followed then by a second rotation; since the second rotation is implemented on the already shifted object.

When using 3D graphic objects, it is always good practice to display the segments representing the coordinate axes in order to better see the objects' position in space. They are, in this sense, used for debugging purposes. They can be deleted afterwards, or simply hidden via their *draw if* parameter.

We suggest that the user practice adding and editing figures. Also, it is important to notice the effect of marking the *split* checkbox when figures overlap. Finally, the different options in the *model* menu may provide better or clearer displays of the objects involved, depending on the purpose they are used for.

During these examples and in the exercise, the objects' color properties were edited. Remember that more information about this can be found in the color editor topic.

8.3 Parameters common to 2D graphic objects

- **spaces menu**: A menu labeled *space* which shows the 2D spaces available where the graphic object may be lodged. This menu **determines where to put** the graphic

object. There is another menu near the top of the panel listing the available graphics at the left of the *Graphics* tab. That other menu is used to **filter** and display in the list only the graphics in the selected space.

- **background**: A checkbox that, when marked, means that the graphic object in question will only be drawn once, when the scene initially loads. Any subsequent changes in the values of variables that may control the graphic (for instance, its shape or position) will not be evident. The graphic can therefore be thought of as being part of the background (hence the checkbox's name). Objects that are not intended to change should be left as background ones, since no time will be wasted re-rendering them. It can also reduce the scene's loading time.
  NOTE: A graphic object with a mark in its *background* checkbox may still refresh itself. This happens only when the spaces scale is changed, or when its origin is shifted. Therefore, another way to understand the function of the *background* checkbox is that it disables the graphic object's re-rendering via controls or variables that affect the object itself, but it does not disable it when the containing space's configuration changes.

- **draw if**: A text field in which a boolean condition is entered. If the condition turns out to be true, it adopts a 1 value and the graphic object will be displayed. Otherwise, it adopts a 0 value and the graphic object will not be displayed.
  For example, if an *a* variable has a value of 3, and *draw if* has `a==2`, the graphic object will be hidden. This happens because *a* is compared to 2, and it turns out *a* is **not** 2. The condition is then false, and the graphic is not displayed. However, if *a* were to change to 2, the object would be drawn.
  A more in-depth discussion of boolean conditions can be found in the boolean conditions and operators topic. It is also important to notice that this *draw if* parameter is present not only in graphic objects, but in spaces and controls as well.

- **abs coords**: When this checkbox is marked, the graphic is drawn in absolute coordinates. These are measured in px units. The horizontal component of the coordinates is the px distance to the right of the left margin of the space, while the vertical components is the px distance below the top margin.
  If the checkbox is not marked, the coordinates will be interpreted as relative to the space's cartesian plane. In this situation, the graphic objects will respond to changes in the space's offset and scale. An important difference to be expected is that, when using absolute coordinates, increasing the value of the vertical component results in shifting the object downwards, whereas when using relative ones, the shift is directed upwards.

- **text**: A text field, along with its *T* and *Rtf* buttons (these buttons launch the *plain text* and *Rich text* text windows, respectively) in which a text can be entered. The text is associated to the graphic in question. For example, a segment's text will be displayed near the segment's midpoint. A more in-depth explanation of text entering and edition can be found in the text editing tool topic.

- **decimals**: A text field in which a positive integer is entered. It is read as the number of decimals to be printed in the graphic related *text*, when it is used to print numerical values. If the value has more decimals than allowed by the *decimals* parameter, the value will be rounded so that the allowed number of decimals is respected.
- **fixed**: A checkbox next to the *decimals* parameter. When marked, the value will **always** display the number of decimals allowed, whether they are significant or not. If unmarked, the decimals will only be displayed when they are significant and are allowed by the *decimals* parameter.
- **font**: A menu with 3 options: *SansSerif*, *Serif* and *Monospaced*. It is used to set the font style of the text accompanying the graphic object. It is therefore present only in graphics that can have such a text.
- **font size**: A text field where the size of the font is entered. This affects the text accompanying a graphic.
- **bold**: When this checkbox is marked, the text accompanying the graphic object will be printed in bold style.
- **italic**: When this checkbox is marked, the text accompanying the graphic object will be printed in italics.
- **trace**: When this checkbox is marked, the graphic object will leave a trace of its positions. This is used for graphic objects that are defined via variables. When the variables change values, the positions or configuration of these graphics may change. If the checkbox is marked, a trace of the objects' visited positions will be left. The checkbox has color editor button next to it, which can be used to set the color of the trace.
- **family**: When this checkbox is marked, copies of the graphic can be displayed depending on the family's *parameter*, the value interval associated with that parameter, and the *steps* used to form the family. These are described below, and are only active when the *family* checkbox is marked.
- **parameter**: A text field where the variable used as the family's parameter is entered. The values this variable adopts are taken by dividing the *interval* in a number of equal parts. The number of separations between these equal parts corresponds to the number indicated in the *steps*.
- **interval**: A text field with an interval. The interval consists of a couple of values (or variables) indicating the beginning and end of the interval. They are flanked by square brackets and separated by a comma. The first value corresponds to the first value the family parameter will adopt. The second value of the interval corresponds to the last value the family parameter will adopt.
- **steps**: A text field where a positive integer is entered, corresponding to the number of separators dividing the equal parts of the interval.

  As an example, consider an *s* family parameter, a $[1, 11]$ interval, and *4* steps. The interval measures 11-1=10, which is then divided so that there are 4 divisions between

its equal parts. It therefore has to be divided into 5 equal parts, each measuring 2 units. The first value *s* has is then 1, then 1+2=3, then 1+4=5, then 1+6=7, then 1+8=9, and finally, 1+10=11 (which corresponds to the intervals second number).

- **width**: A text field which contains the width of the graphic's line in px. This width is the thickness of the line or curve being drawn, and should therefore not be confused with the width of some graphic objects such as the rectangle.
- **info**: A text field where a brief description of the graphic, or what it does, can be included. This is optional, and this field can be left empty. However, when having multiple graphics of the same type involved, it may be useful to further describe them so as to more quickly locate a particular one. The description entered is also displayed in the list at the graphic objects left list panel.
  NOTE: this feature is also present in almost all other *DescartesJS* elements: spaces, controls, definitions, and programs.
- **line style**: A menu with 3 options. It is used to determine how the line of some graphic objects is to be drawn. The *solid* option results in a continuous line. The other two options result in a non-continuous line (dots and dashes).

8.4 Parameters common to 3D graphic objects

We now discuss the parameters that are common to most three dimensional graphic objects. The list includes only those parameters which have not been explicitly described in particular graphic objects.

- **back color**: A button that launches the color editor, which can then be used to assign a 3D object's back color. Since 3D objects have a front and a back, the first color button described sets the front. This *back color* can be used to set the same or a different color for the back side.
- **inirot**: A text field in which three numbers are entered. They are flanked by parentheses and separated by a comma. These numbers define an initial rotation of an object. The first number is the angle, in degrees, of a rotation around the *x* axis. The second is around the *y* axis. The third is around the *z* axis.
  In some situations, a different kind of rotation may be more helpful, known as an Euler rotation. In these rotations, the first number corresponds to the angle of rotation in degrees of a first rotation around the *z* axis. The second number is then the rotation around the *x* axis (which, if non-zero, would result in a change in a new *z* axis). The third number is then a last rotation again around the **new** *z* axis. If such a rotation type is to be entered, an `Euler` prefix has to be added before the first parenthesis. For instance, `Euler(10,20,10)`.
- **inipos**: A text field with a three dimensional vector (three numbers flanked by parentheses and separated by commas) that corresponds to the initial position or shift given to the objects default position. As usual, the first number is the *x* coordinate,

the second is the *y* coordinate, and the third one the *z* coordinate. This position shift is performed **after** the initial rotation has been implemented.

- **endrot**: A text field that works exactly as the *inirot* parameter. However, the rotation here entered takes place **after** the initial position shift has been implemented. And the rotation fulcrum is also the origin, so if an object has been shifted from the origin, this rotation will also result in further shifting the object.
- **endpos**: A text field that works exactly as the *inipos* parameter. Its position shift occurs **after** the end rotation has been implemented.

  The user can take advantage of the order in which these rotations and position shifts occur. They allow for an easier positioning of certain objects.
- **split**: When this checkbox is marked, three dimensional objects that overlap are carefully drawn so as to consider which parts of them appear in the before or at the back of others from the viewer's perspective. This allows for a better presentation of the intersections. This option should only be used when objects will potentially cross each other, since it involves more calculations that may impact the scene's performance.
- **edges**: When this checkbox is marked, the edges defining certain objects are drawn in a specific color. This color is chosen via the color editor button located next to the checkbox. The edges are related to the *Nu* and/or *Nv* parameters of the custom 3D graphic objects.
- **model**: A menu with four options related to how a 3D object is drawn:
  - *color*: uses a fixed color for the object.
  - *light*: provides a sense of illumination. It acts as if a light source were present. The color given to the object may vary in brightness depending on the orientation of its parts.
  - *metal*: works similar to the *light* option, but enhances the contrast of the bright spots, endowing the surfaces of the objects with a *metallic* condition.
  - *wire*: Used to draw only the edges of the object with the color selected via its accompanying color editor button. The faces of the objects are not drawn.
- **Nu**: A text field in which a positive integer is entered. For graphic objects which depend on a *u* parameter, the number here entered determines the number of parts into which the *u*'s $[0,1]$ interval is subdivided.
- **Nv**: This textfield works in the same way as the *Nu* one. But it is related to the *v* parameter of graphics that require one, such as the *surface* or the *torus*.
- **width**: A text field related to some three dimensional graphic objects as the *ellipsoid*. It is related to the measure of the object, when it has not yet been subjected to position shifts or rotations, in the *x* axis direction.

- **length**: A text field that works similarly as the *width* one. However, it is related to the measure of an object in the *y* axis direction.
- **height**: A text field that works similarly as the *width* and *length* ones. However, it is related to the measure of an object in the *z* axis direction.

9

The *Controls* tab

Now that we have a firmer grasp on the *Graphics* tab, we turn our attention to a new tab: *Controls*. Some exercises have already included a bit of the *control* functionality. This section studies it more in depth.

Controls are objects that act as means of interaction for the user. They enable the user to perform changes in variable's values, launch animations, execute functions, as well as many other actions. The average user has most likely used one of these tools, such as spinners, scrollbars, menus and buttons.

Just as with the graphic objects already reviewed, different types of controls share many parameters. These common controls can be checked near the end of the current section. If a parameter is mentioned in a particular control, it is because its functionality in relation to that control is specific, or because it is only present in that type of control.

The *Controls* tab is shown by clicking on its name tab at the top of the configuration editor. Figure 9.0.0.1 shows the configuration editor with the *Controls* tab displayed, right after a new spinner has been added.

The *Controls* tab also has a panel at the left which houses a list of the added controls. This panel's functionality is the same as for the Graphics tab left panel. This panel allows the user to filter the controls inside a specific space, has a + button to add a new control, a - to delete it, a * to clone a control, and arrow buttons to move a control up or down in the list.

There is a slight difference between how the graphics and controls behave regarding their place in the list at the left panel. Whereas graphics are drawn in the order in which they appear (the first one drawn being the one at the top of the list), controls are drawn in the reverse order. The last one drawn is the one at the top of the list, and if two controls share the same place inside a same space, the one actually visible will be the one nearer the top of the list.

Additionally, whereas graphics can be set in absolute or relative coordinates via their *expression* parameter, the *expression* parameter of controls is always interpreted in absolute coordinates.

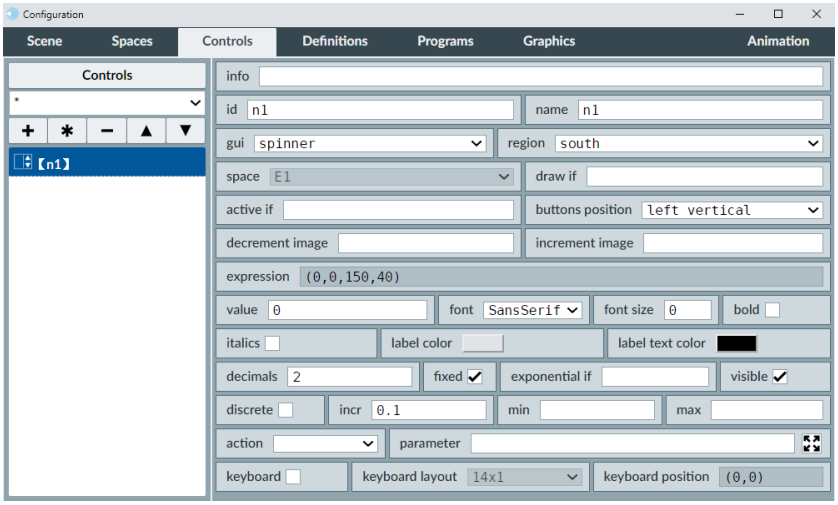


Figure 9.0.0.1: The *Controls* tab.

There are many types of controls. A few are considered *numeric*: the button, spinner, scrollbar, checkbox, text field and menu. There is also only one *graphic* one, which is draggable. There are some other which do not fall into either category: the text, video and audio controls. To add a control, the + button at the top of the left panel of the *Controls* tab must first be clicked. This launches a pop-up dialog where the control type can be selected in a menu. This dialog also has a text field where the control's identifier is entered. Upon clicking *Add* in the dialog, it closes and the control appears in the list at the left panel.

Most controls are lodged by default in the southern area of the scene (an area that appears at the bottom of the screen when a control is placed there). This can be changed by selecting a different option in the control's *region* parameter.

9.1 *Spinner* numeric control

This type of numeric control is associated with a variable named after the control's identifier. The spinner allows the user to increase or decrease the value of the variable in question by means of a couple of buttons included in it. It can additionally have a text field in which the user manually enters a value or expression for it. Most of the spinner's parameters are common to other control types, and can therefore be reviewed in the controls' common elements topic. The ones particular to the spinner are:

- **buttons position**: A menu used to determine the position and orientation of the buttons used to increase / decrease the spinner's variable value. Their position is relative to the text field associated to the spinner, and the orientation can be vertical or horizontal. Since the configuration of the spinner is horizontal by default, the horizontal options in this menu allow for larger buttons. The options are:

  - **left vertical**: (default option) Both buttons are at the left of the text field. The button to decrease lies below the one to increase.
  - **right vertical**: The buttons are vertical, as in the last option, but at the right of the text field.
  - **left horizontal**: Both buttons lie left of the text field. However, each completely spans the control's height. The one used to decrease the value is at the left of the one used to increase it.
  - **right horizontal**: The buttons are set horizontally, as in the previous case, but both lie at the right of the text field.
  - **end horizontal**: The decrease button appears at the left, then the text field at its right, and finally the increase button. In this option, the text field is centered between the buttons.

- **decrement image**: A text field blank by default. A path to an image file can be entered here if the button used to decrease the spinner's value is to have an image background. The path is set relative to the scene's *html* file. Bear in mind that, if an image is set here, it will only be displayed after the scene has been saved (so that the path makes sense), and reloaded. Additionally, the image's original size is rescaled when it does not fit in the button.

- **increment image**: A text field that works very similar to the *decrement image* one. However, the image here included will be set as the background to the button used to increase the spinner's variable value.

Figure 9.1.0.1 presents an example of a spinner control. Figure 9.1.0.2 shows the configuration required for the example.

Note that the spinner in the example is designed to control the scale of the *E1* space. The control's *action* parameter is set to *calculate*, and the parameter of calculation assigns the spinner's value to the scale value of the aforementioned space. A more in-depth review of this type of variables can be reviewed in the space variables topic.

Figure 9.1.0.1: *Spinner* numeric control example.

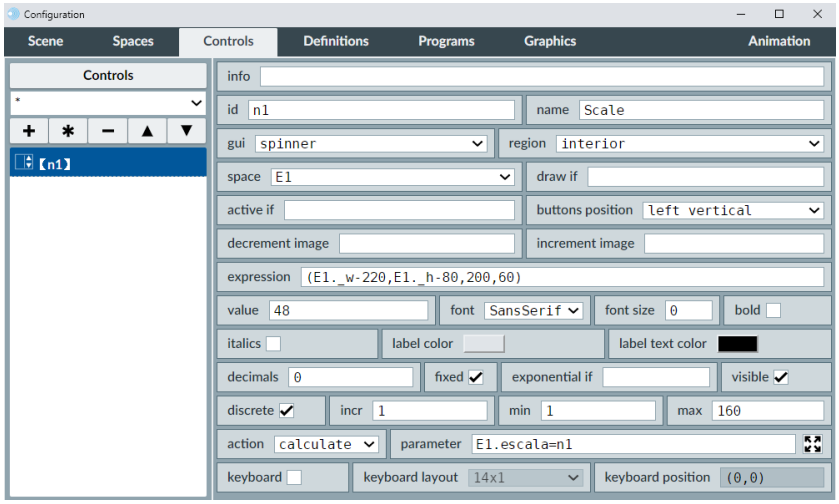


Figure 9.1.0.2: *Spinner* control example configuration.

Note that the spinner in the example has a checkmark in its *visible* checkbox. It therefore displays the text field in which the user can enter the value to use. Beside accepting explicit numeric values, this text field also accepts expressions that can be internally evaluated. For example, if `2+1` is entered, after clicking the *ENTER* key, the spinner's value in the text field automatically changes to 3. Or, if `pi/2` is entered, a 1.5707... value is displayed in the text field (this depending on the decimals allowed for the spinner, and considering that *pi* is an expression which *DescartesJS* internally recognizes as π). All this behavior is also present for the text field control.

We are now ready to try an exercise involving two spinner controls for the width and height of a rectangle triangle. The scene is additionally supposed to display the length of the hypotenuse and the area of the triangle. This exercise's interactive scene, along with

the instructions to build it, can be found at Controls Spinner. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise shows the importance of the *action* parameter of a control. It also stresses the importance of initial calculations, such as the ones used in the *INICIO* algorithm so that the interactive scene is prepared to work from the beginning.

9.2 *Text field* numeric control

This kind of numeric control is a field where text or numbers can be typed. It is particularly useful to provide the user a means of entering information into the scene. Its main parameters are discussed in the controls' common elements topic.

Figure 9.2.0.1 presents an example of a textfield. Figure 9.2.0.2 shows the configuration required for the example.

Note that, in this example, the textfield is used as a means to get the user's name. That is the reason why it is set as text only. Its *name* parameter is left empty, since the user has no need to associate it with a name. The control's initial value is set to a couple of single quotes (' '). This means that a blank character string is its initial value. Additionally, a text graphic object was separately added to print the *What is your name?* question but this functionality is not addressed in this topic.

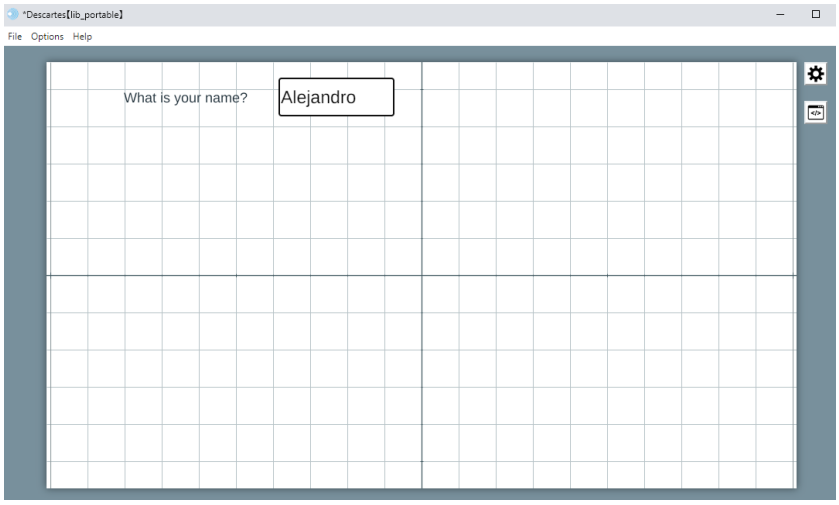


Figure 9.2.0.1: *Textfield* numeric control example.

- **only text**: A checkbox that favors a textual interpretation of whatever is entered in the textfield.
  For example, if the *only text* it is marked for a *t1* textfield, and the *value* is 123, the

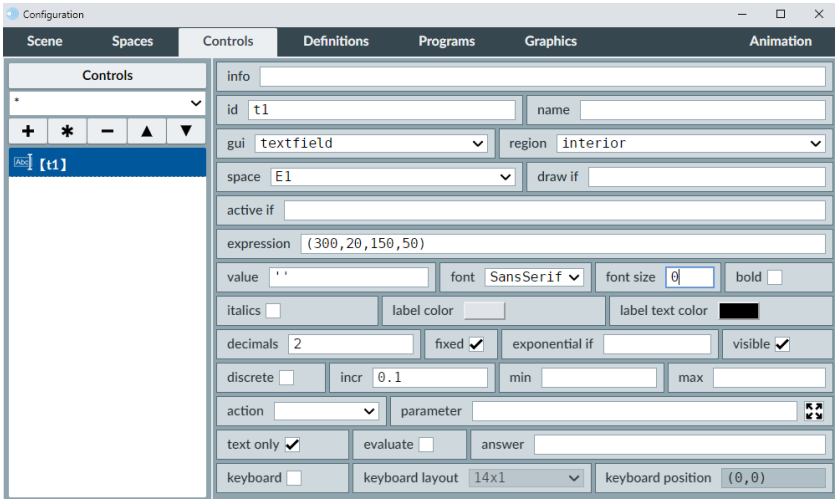


Figure 9.2.0.2: *Textfield* control example configuration.

`t1+3` operation will result in 1233 (the concatenation of the string 123 with 3). Otherwise, `t1+3` results in 126 (the sum of both numbers). *DescartesJS* is relatively intelligent and may operate it if it recognizes what is entered can be interpreted as a number, even when the *only text* checkbox is marked. For instance, `t1*3` will return 369 instead of returning an error message.

Textfields include an automatic expression evaluation tool. It works if the *only text* checkbox is unmarked. This behavior is shared with other control that include their own textfields, such as spinners. As an example of this functionality, consider entering `2+1` in a textfield. After pressing ENTER, the textfield will have a 3 value inside. Similarly, the user could enter `cos(pi)` (remember *DescartesJS* has an internal *pi* variable with an approximation of π). Upon pressing ENTER, -1, the expression is evaluated and a -1 value now appears in the textfield. If, however, the textfield is set to be text only, the expression is not evaluated and the entered text remains there. Some situations make this automatic evaluation undesirable. In such cases, the *_Num_()* function can be used. This function is described more in depth in the *DescartesJS* language functions topic.

We are now ready to try an exercise involving a textfield. This exercise shows the different ways in which the user can interact with it. It also stresses the difference between a text only textfield and one that is not. It also includes a bit of practice with the text edition tool. For more information on this functionality, read the text editing tool topic.

This exercise's interactive scene, along with the instructions to build it, can be found at Controls Textfield. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise allowed us to see many things. An important one being the difference between a text only textfield and one that allows for numerical values. The *NaN* error message typically appears when attempting a division by zero, or the extraction of a square

root of a negative value. However, it is also displayed when operating non-numerical elements. One last important thing to consider is that, even though a number is entered in a text field, it does not mean it will be taken as such. It can also be taken as a character string in which the characters are numbers.

9.3 *Menu* numeric control

This control is a typical menu. It is typically used to provide the user with options from which to choose within the interactive scene. The identifier of the menu holds its value, which is a positive integer. If the menu's first choice is selected, its identifier has a 0 value. If the second choice is selected, the identifier's value is 1, and so on. Many of this control's parameters are generic to all controls and can be reviewed in the controls' common elements topic.

Figure 9.3.0.1 presents an example of the menu control. Figure 9.3.0.2 shows the configuration required for the example.

Note that the example presented could serve the purpose of providing the user a means (multiple choice) to answer a given question. For instance, where a number lies in the real line relative to 2 numbers (a and b). The menu here has four options (as can be seen in its *options* parameter). However, the first one is used only to present its title, whereas the final 3 are actually possible answers.

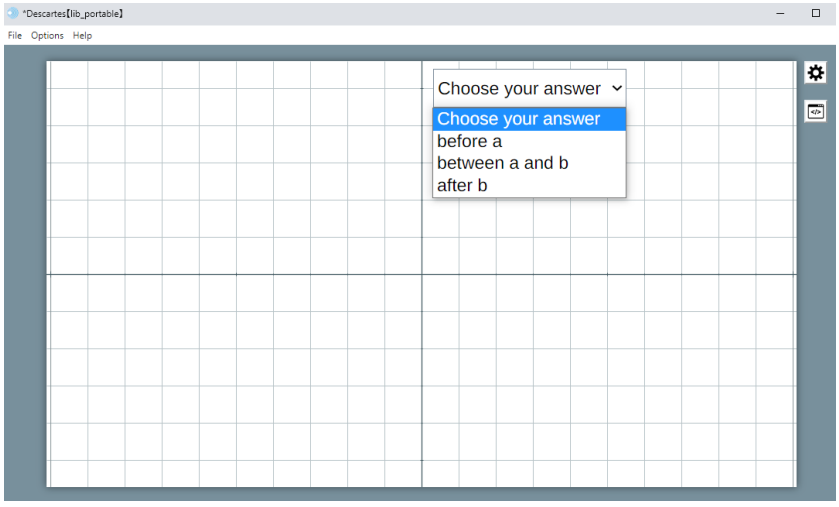


Figure 9.3.0.1: *Menu* numeric control example.

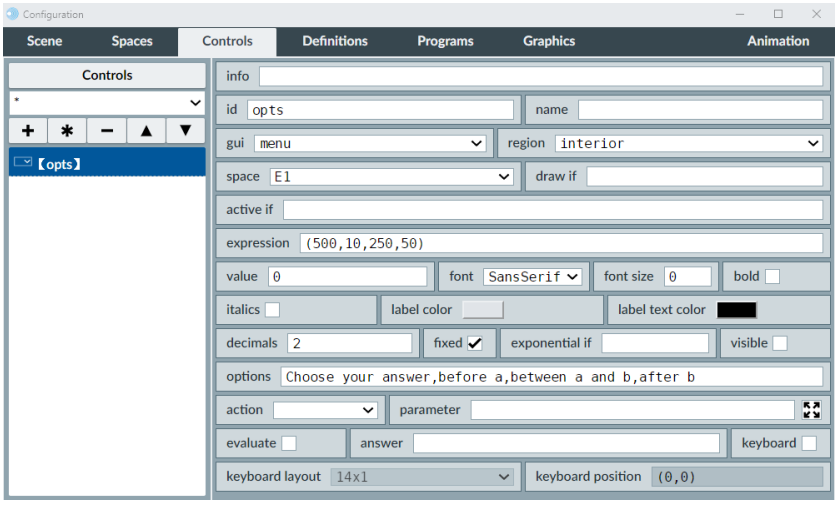


Figure 9.3.0.2: *Menu* control example configuration.

- **value**: A text field in which a positive integer value is entered, which corresponds to the menu's initial value. This value determines the value given to the identifier from the start, and therefore determines which option is selected in the menu when launching the scene.
- **options**: A text field where the options of the menu are entered in order. The comma (,) character is used to separate the options. If the first options is to remain blank, the list can start immediately with a comma.

We are ready to try an exercise in which a menu is used to determine which geometric figure is shown from various different ones. Besides including a menu, this exercise also uses conditionals and text graphic objects.

This exercise's interactive scene, along with the instructions to build it, can be found at Controls Menu. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise has a bit more difficulty and is a bit more professional than other that have come before. A little more attention is paid to aesthetics, such as centering the menu and hiding the cartesian plane when it becomes unnecessary. The menu's identifier is also now chosen by the programmer. Graphic objects also include a description via their *info* parameter, making them easier to identify in the list.

9.4 *Scrollbar* numeric control

This type of numeric control consists of a draggable bar which can be set vertically or horizontally, much like the progress scrollbars in many computer windows. When the bar is dragged, the numeric control changes value. All the parameters related to this control

are discussed in the controls' common elements topic, so no particular elements are discussed for the scrollbar.

Figure 9.4.0.1 shows a scrollbar example. Figure 9.4.0.2 shows the configuration required for the example.

Note that the bar in the example is set to control the scale of the *E1* space (its *action* is set to *calculate*, and the calculation parameter assigns the control's value to the space's scale variable). This functionality is not inherent to the bar, and can be reviewed in the space variables topic.

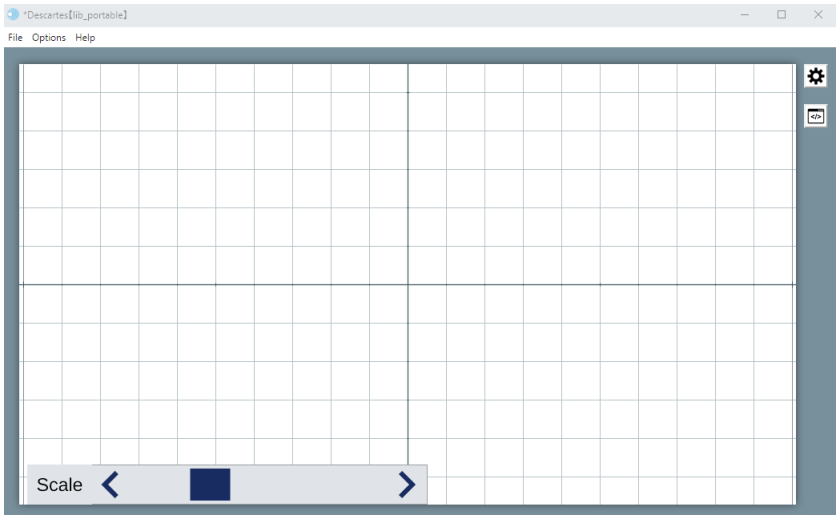


Figure 9.4.0.1: *Scrollbar* control example.

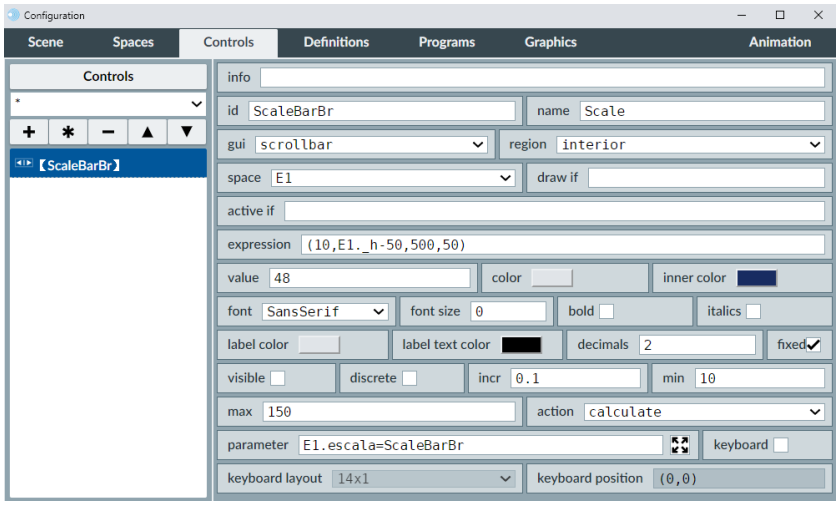


Figure 9.4.0.2: *Scrollbar* control example configuration.

- **color**: a button which launches the color editor tool to set the scrollbar's background color or image.
- **inner color**: a button which launches the color editor tool to set the scrollbar's handle color or image, and the buttons at the scrollbar's edges. If a color is assigned using the *Gradient* tab of the color editor tool, the edges colors correspond to the gradient vector's edges colors; and the scrollbar's handle changes color when moved near one edge or the other.

We now do an exercise to practice the use of this type of control. We also take this opportunity to practice additional *DescartesJS* functionality such as the use of variables related to spaces. This exercise's interactive scene, along with the instructions to build it, can be found at Controls Scrollbar. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise allows us to see how a control can directly change the value of a variable, which in turns allows it to directly control the scale of a given space. This can be used as a means to zoom in or out inside the space in question. The equation graphed in the example oscillates infinitely at any given interval around the origin. The user can use the scrollbar to zoom in and notice this effect. Special care should be taken when handling scale variables. If a scale variable is set to 0 or negative, the scene might crash. That is the reason why a minimum 10 value is set for the scrollbar.

A scrollbar can also be vertically oriented. This is achieved by setting the scrollbar inside a space (with its *region* parameter set to *interior*), and assigning it a width lower than its height.

9.5 *Button* numeric control

This control is the simplest there is. It consists of a button that, when clicked, does a specific action. Many of its parameters are common to most other controls, and can be reviewed in the controls' common elements topic.

Figure 9.5.0.1 presents an example of the numeric button control. Figure 9.5.0.2 shows the configuration required for the example.

Note that the example shown is used in a case in which the calculation done by the button determines whether the user's answer is correct or not. The answer provided is stored in another numeric control (*AnswerField*), a text field. Note also that the button's calculations involve a variable named *correct* which is 1 if the answer is correct, and 2 otherwise. The functionality involved in this particular calculation is reviewed in the boolean conditions and operators topic.

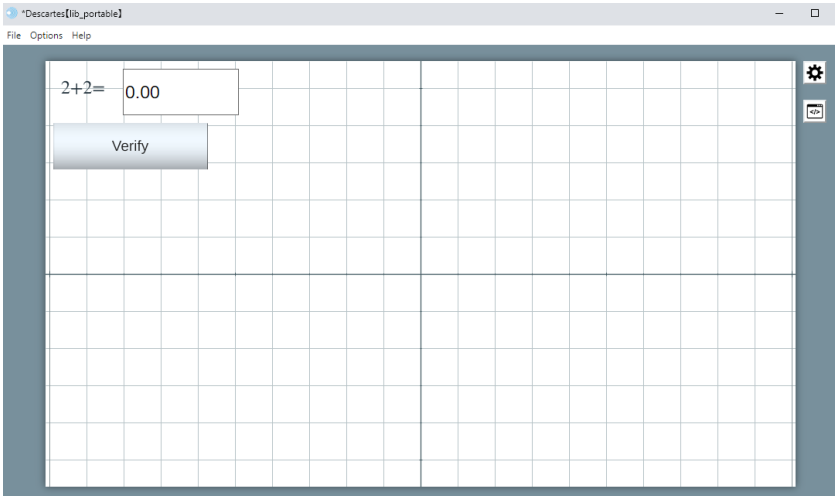


Figure 9.5.0.1: *Button* numeric control example.

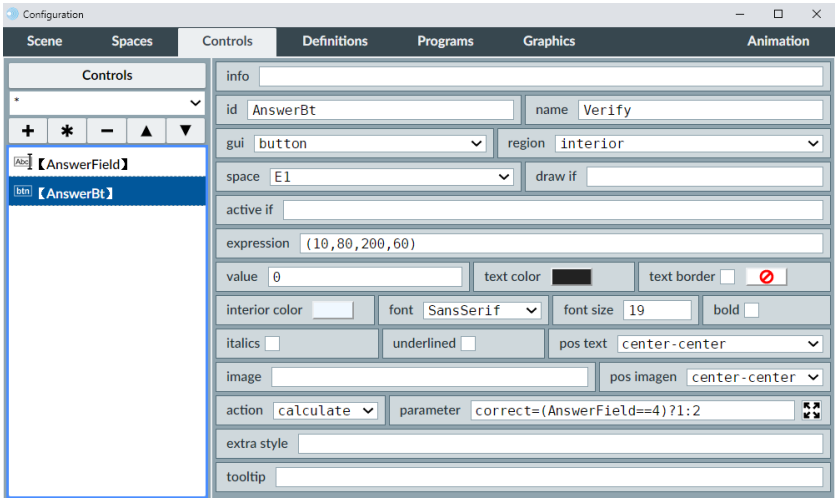


Figure 9.5.0.2: *Button* control example configuration.

- **text color**: A button which launches the color editor dialog to select the color of the button's text.
- **text border**: A checkbox that, when marked, draws a border around the text. The color of the border is selected using the color editor button right of the checkbox. This button is only active if the checkbox is marked.
- **interior color**: A button that launches the color editor, which is used to set the inner background color of the button. As will be mentioned soon, it is possible to make buttons display images. If such images have a certain degree of transparency, the button's *interior color* transparency should be set to its maximum, so as to avoid undesired effects due to compounded transparencies.

- **font**: A menu to select the font to be used on the button's text. The available options are *SansSerif*, *Serif*, and *Monospaced*.
- **font size**: A text field in which the size of the font is specified in points.
- **bold**: A checkbox to set the font in bold style.
- **italic**: A checkbox to set the font in italics.
- **underlined**: A checkbox to underline the font.
- **pos text**: A menu that controls the position of the text relative to the anchor point indicated in the *expression* parameter of the text graphic. Its options are *top-left*, *top-center*, *top-right*, *center-left*, *center-center*, *center-right*, *bottom-left*, *bottom-center*, and *bottom-right*. As an example, if the user chooses *top-right*, you can imagine the text to be contained in a rectangle. The rectangle's top-right corner will be the one placed in the coordinates indicated by the text's *expression* parameter.
- **image**: A text field where an image file (*jpg* or *png*) to be used as the button's background is indicated. It should include the path to the file relative to where the scene's *html* file is located. It should also include the file extension. If the file is in a subfolder, remember to use the / character as separator. For instance, for a *btn.png* file inside an *images* folder placed at the same level as the *html* file, this parameter should contain `images/btn.png`. A couple of additional image files may be added for a down functionality (the image appears while the button is pressed down), and an *over* functionality (the image appears when the mouse hovers over the button without clicking or pressing it).
  - *down*: The image file to be used has the same name as the button's background image file, but also includes a *_down* suffix. This file must be stored in the same folder as the background file. For example, if the main image file is named *btn.jpg* the *down* file should be named *btn_down.jpg*, and stored in the same folder as *btn.jpg*. When the button is being pressed, this image is displayed.
  - *over*: Similar to the *down* functionality, an *over* file can also be included in the same folder by using an *_over* suffix. For our *btn.jpg* example, a *btn_over.jpg* file can be included in the same folder as the main image file. This *over* image is displayed when the the mouse hovers over the button.
- **pos image**: A menu with the same options as the *pos text* one. The selected position is the anchor point, inside the button's area, in which the image's top left corner is to be placed.
- **tooltip**: A text field where a tooltip contextual indication can optionally be included. This indication will briefly be shown when the user hovers the button in the scene for longer than 1.5 seconds. It usually consists of a text explaining what the button does.
- **extra style**: A text field that can be used to implement additional style to the button. The style indicated via this parameter is different from the *over* and *down* images

that can be associated to the button. For instance, a button could have the following string, which involves all the available edition that can be used in this parameter:
`border=3|borderRadius=10|borderColor=ff0000|overColor=e0a12b|downColor=0000ff|inactiveColor=c0c0c0|font=Serif|shadowTextColor=a6620e|shadowBoxColor=808080|shadowInsetBoxColor=b46100|flat=1`
All this text is written run on (no breaks). Each edition block is separated from the next by a | symbol (commonly known as *pipe*), which is typically found left of the 1 key in the alphanumeric keyboard. After an edition parameter, an = sign follows, followed by its assigned value. If one edition parameter is not indicated, it is ignored an the button uses the default configuration for it. The following list includes all available edition parameters:

- **border**: a number associated to the number of px used for the width of a border that surrounds the button.
- **borderRadius**: the number of px used as the radius of a circle to define the curvature at the corners of the button (a greater radius corresponds to a larger part of the corner being curved). Its default value is 0, and it means the button will have right angles for corners.
- **borderColor**: a hex color code for the border of the button (the one defined by the *border* edition parameter). For more information on hex color codes, visit the color editor topic.
- **overColor**: a hex color code to specify the button's background color when the mouse hovers over it for more than 1.5 seconds.
- **downColor**: a hex color code to specify the button's background color when the button is being pressed down.
- **inactiveColor**: a hex color code to specify the button's background color when the button is in its inactive state. That is, when the condition inside the button's *active if* parameter is not being met.
- **font**: the name of the font to be used. Its options are *SansSerif*, *Serif*, and *Monospaced*.
- **shadowTextColor**: a hex color code to specify the *shadow generated by the text*. If this edition parameter is not included, no shadow will accompany the text.
- **shadowBoxColor**: a hex color code to specify the color of the shadow generated by the button's edges. This shadow lies *outside* the button itself, and the shadow is therefore not clickable. If it is not included, no such shadow is implemented.
- **shadowInsetBoxColor**: a hex color code to specify the color of the shadow inside the area of the button. Since it is inside, this shadow is actually part of the button and is therefore clickable. If this edition parameter is not included, no such shadow is implemented.
- **flat**: an edition parameter that determines if the button's design is to be flat. A zero value results in a button with its default color gradient. A 1 value results

in a flat design button (no color gradient). Note that, when gradient is present, it is drawn vertically: the button's center being drawn clearer than the top and the bottom.

We now do an exercise involving a button whose function is to calculate the value of a numeric sequence when its two initial values are specified. This exercise's interactive scene, along with the instructions to build it, can be found at Controls Button. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise allowed us to practice the use of the *button* numeric control as a means to calculate sequences of values. Note that the *sum* variable in each step corresponds to the values of the Fibonacci sequence, its initial values being 1, 1, 2, 3, 5, 8, 13, 21, 33, ... A value of a certain number in the sequence is the sum of the two preceding ones. A noteworthy difference found in this exercise is that it stresses the importance of initializing variables in some situations. Were the variables in this example not initialized with a 1 value, the value of *sum* would be always 0. Variables not initialized are all given a default 0 value.

9.6 *Checkbox* numeric control

The checkbox is a control which adopts only two values: 1 for marked, 0 for unmarked. They are typically used as means to answer *yes/no* questions and in multiple choice exercises (when under their radio button functionality, as described below).

This control's functionality is twofold: its default one is as a checkbox. However, it can also be used as a radio button, which is a control that can also be only marked or unmarked, but that belongs to a group with other controls, and only one of all can be marked. Imagine a survey takes place where the user can select his/her favorite genres of movies. In this example, checkboxes are the choice to use, since the user can check more than one, even all if that is the case. However, if the behavior is mutually exclusive, as in the selection of intervals for a date of birth, the radio button is more adequate. In this case, if the user selects an option, all other options are cleared of their mark.

Figure 9.6.0.1 presents an example of the numeric checkbox control. Figure 9.6.0.2 shows the configuration required for the example.

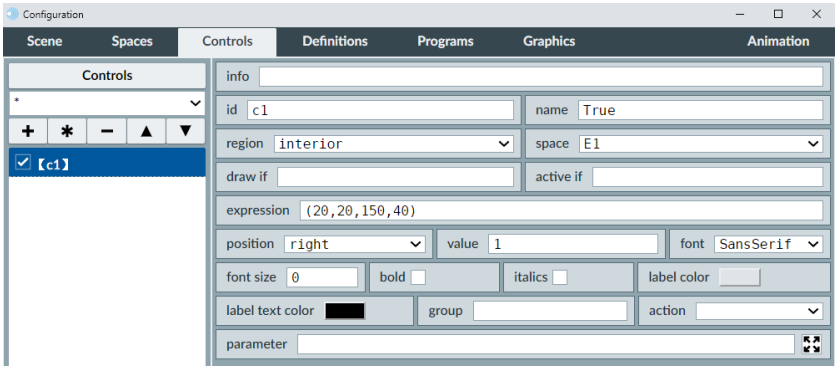


Figure 9.6.0.1: *Checkbox* numeric control example.

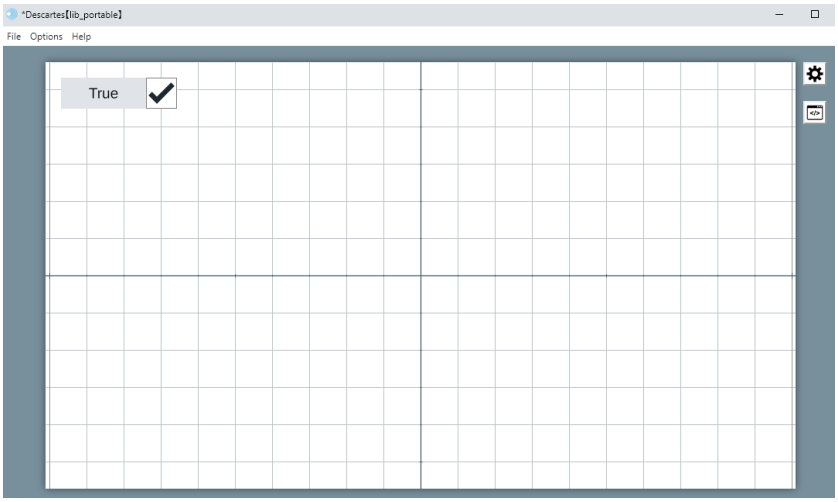


Figure 9.6.0.2: *Checkbox* example configuration.

- **id**: A textfield for the identifier of the checkbox, which is also the internal variable holding the value of the control.
- **value**: A textfield for the initial value assigned to the control. For checkboxes and radio buttons, 0 means the control is unmarked and 1 means it is marked.
- **group**: A text field where the name of a group of controls is entered. This is used only if the control is to be used as a radio button rather than as checkbox. All controls belonging to the same group, and therefore acting as mutually exclusive from one another, should have the same group name entered in this parameter. This ensures that when one is selected, all other in the same group become unmarked.
  If this text field is left blank, the control will act as a checkbox
- **position**: A menu to choose if the checkmark (or radio button) is to be placed at the *right* of its name or at its *left*.

A checkbox and a radiobutton have a graphic difference: the former is a square box and its mark is a checkmark, whereas the latter is a round area and its mark is a small black circle inside it.

We now do an exercise involving the checkbox control under its checkbox functionality and under its radio button one, so as to better understand the difference. Suppose the user is to determine which of four different units is the odd one out. This requires a radio button, since the question is designed so as to only have **one** odd option out.

This exercise's interactive scene, along with the instructions to build it, can be found at Controls Checkbox. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

On the one hand, the checkbox is a control allowing a binary value (0 or 1), and it is not coupled with other controls. On the other hand, the radio button is also binary, but is coupled with other similar controls. The coupling is done via the *group* parameter of the control.

Again, note that this exercise required determining whether the answer given by the user is correct or not. The boolean conditions and operators topic has information regarding this functionality.

9.7 *Graphic* control

Graphic controls, unlike numeric ones, consist of points in a 2D space which can directly be dragged using the mouse or by pressing and dragging them in mobile devices. These controls are added by clicking the + button in the *Controls* tab, and selecting the *graphic* option from the menu in the pop-up dialog. They also have identifiers, and it can be set directly in that dialog, or specified later on via the control's *id* parameter. Once added, the dialog closes and the graphic control appears also listed in the left hand side panel of the *Controls* tab.

Figure 9.7.0.1 presents an example of a graphic control in a scene. Figure 9.7.0.2 shows the configuration required for the example. This example also has an *equation* graphic object present used to draw the equation of constraint of the graphic control. The equation's expression is the same indicated in the graphic control's *constraint* parameter: $y^2 - x^2 = 1$. The purpose of drawing the equation is to show that the graphic object is *constrained* and can only be present in that part of the 2D space. Visit the equation graphic object topic for more information on this type of graphic.

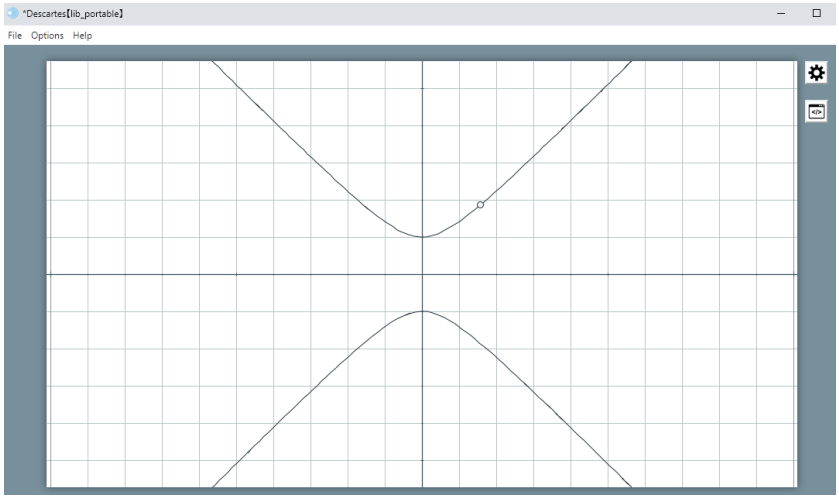


Figure 9.7.0.1: *Graphic* control example.

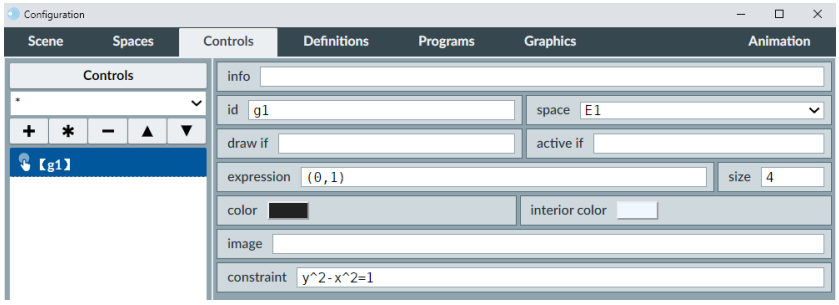


Figure 9.7.0.2: *Graphic* control example configuration.

- **id**: A text field to enter the graphic control's identifier. This identifier is also used as a prefix to variables related with the control's horizontal and vertical positions relative to the cartesian plane, and to a variable which determines if the control is being used or not. This constitutes a difference with the information the identifiers of numeric controls. Numeric controls' identifiers only held a value. Graphic controls' identifiers are associated with three variables related to the control's state.
- **space**: A menu where the space lodging the graphic control is selected.
- **expression**: A text field with the initial coordinates of the graphic control. The origin is its default. If the control is to be constrained to a curve and the point in the expression does not lie in that curve, the control will appear in the point on the curve nearest to the one in the *expression* parameter.
- **size**: A text field with the radius value (in px) of the circle representing the graphic control. Small sizes may result in difficulty in clicking and dragging the control.
- **constraint**: A text field to enter the constraint equation for the graphic control. Its default value is blank, which means the graphic control is not constrained. If

an equation is entered here, the graphic control will only be able to move on the graph of the constraint equation. For example, a `x^2+y^2=4` constraint will allow the graphic control to only move in a circle centered at the origin with a radius of 2 units. Were the constriction `x^2+y^2<=4`, the control would also be allowed to live **inside** the circle, but not outside.
If the *expression* given for the graphic control's initial position is not part of the constraint, the point in the constraint nearest the expression is chosen instead.

- **color**: A button that launches the color editor dialog the outer color of the circle that represents the graphic control.
- **interior color**: A button that launches the color editor to select the graphic control's inner or fill color (the color inside the circle representing the graphic control).
- **image**: A text field where a path relative to the scene's *html* file may be entered. This path points to a *jpg* o *png* image that will represent the graphic control instead of the circle. The image then will be clickable and draggable as the graphic control.

Some situations demand more detailed information of where a graphic control is, or further restrictions to its movement than those specified in the *constraint* parameter. There are two variables related to the horizontal and vertical coordinates of a graphic control relative to the cartesian plane of the containing space. These are <*control identifier*>.*x* and <*control identifier*>.*y*. For example, if a control has a *grf* identifier, *grf.x* contains its horizontal position value and *grf.y* contains its vertical position value. These variables can be used both to print the values and know exactly where the graphic control is, or to enter the horizontal and vertical values so the used can place the control exactly where desired. Another related variable is <*control identifier*>.*active* (or, also, <*control identifier*>.*activo*), which has a 1 value when the graphic control is selected or being dragged, and 0 otherwise. For our current example, *grf.active* would be the related variable.

Furthermore, *spinner* type numeric controls can be created with their identifiers corresponding to a graphic control's horizontal and vertical coordinates so as to restrict the positions allowed for the graphic control. For instance, if a spinner control with a *grf.x* identifier is added, with an initial 0 value, 1 unit discrete increments; if the user then drags the *grf* graphic control, it will not move around freely horizontally, but will rather skip, only falling on integer values for its horizontal component.

In order to make all this functionality clearer, we first undertake a first exercise. Suppose we want a scene with a histogram of the number of persons with ages 26 and 27. The histogram will involve a couple of columns whose height is directly draggable (using a graphic control for that purpose). As a part of this exercise, the average age is also calculated.

This exercise's interactive scene, along with the instructions to build it, can be found at Controls Graphic 1. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This scene shows us how draggable objects are useful to control certain variables. This can potentially be extended to control the sides of geometric figures, the width or heights of images, etc. This ability to drag objects is handled via graphic controls. We also saw an example of how to use the *CALCULOS* algorithm to perform instructions almost continuously. However, as mentioned in the topic dealing with that algorithm, the programmer has to be careful when using it. Saturating it with instructions may be unnecessary and will only make the scene respond slower. We also saw a strategy to avoid showing errors such as a *NaN* error. An alternative *No average can be calculated with zero persons involved* text could even be displayed when the sum of persons is zero. Finally, the use of the *external region* is made patent as a place where the programmer sets aside variables that may require to be controlled only for debugging purposes, but that are not to be made available to the end user.

We now do another exercise to further our knowledge of graphic controls. This exercise additionally provides a useful programming tool: the identification of a position via both its relative and absolute coordinates. Even though graphic controls are based on relative coordinates, sometimes it is useful to know where they are in absolute ones. This exercise's interactive scene, along with the instructions to build it, can be found at Controls Graphic 2. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise provides a means to know both the relative and absolute coordinates of a graphic control, regardless of any offset to which the origin of the space may be subjected. If the programmer needs to place a button at a particular place, the graphic control could be placed there and the absolute coordinates known so as to place the button there.

These represent tools useful to the programmer. Once the objects have been placed where they belong, the graphic control and its accompanying text can be hidden, or even removed. They can be hidden by introducing a `0` in their *draw if* parameter. In this way, the can be re-used should they be necessary again.

During this exercise, we used a *E1._w* and a *E1._h* variables, as well as a *E1.Ox* and a *E1.Oy* one. Finally, we used the *E1.escala* variable. All these variables are *DescartesJS* intrinsic variables, which can be reviewed in the space variables topic.

9.8 *Text* control

These controls are blocks of text inside a box. Though an initial text can be entered directly in the text control in the configurations editor, it is also possible to edit it later on in the scene. The text control in the scene also has a button located at its bottom right corner. This button can be used to alternate a question / answer text.

Figure 9.8.0.1 presents a text control example. Figure 9.8.0.2 shows the configuration required for the example.

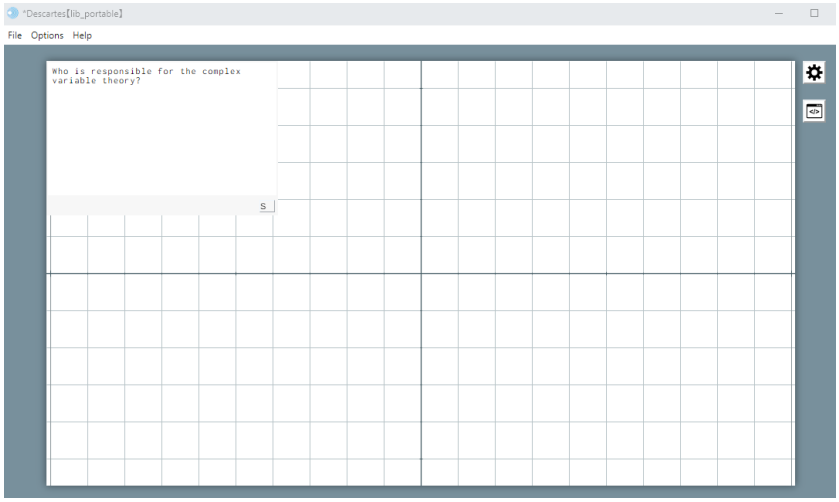


Figure 9.8.0.1: *Text* control example.

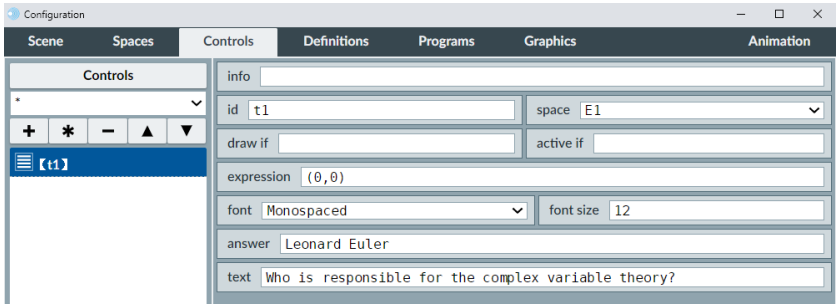


Figure 9.8.0.2: *Text* control example configuration.

- **font**: A menu to choose the font used fo the text.
- **font size**: A text field with the font size in points.
- **answer**: A text field where the answer to a question entered in the *text* parameter is entered. This is used usually when the text control is intended to have a question / answer functionality. Note that this answer is only displayed. There is no evaluation done to determine whether what the user may have entered is right or wrong.
- **text**: A text field to enter the contents of the text control. If it is a question, the *answer* parameter can also be set to contain the answer to it.

The interactive scene will display the text entered in the *text* parameter inside the control in the scene. The user can type text in that field also. If there was anything included in the *answer* parameter, a small button with an *S* letter inside appears at the bottom right corner of the text control. If pressed, the text in the *answer* parameter is displayed and the button's letter changes to *T*. If pressed again, it displays the text in the *text* parameter again and the button reads *S* again.

The text control identifier is a variable that holds the text entered it via its *text* parameter. If the text is modified by the user in the scene, the variable's content is updated.

9.9 *Audio* control

This control consists of an audio player that can be used inside the interactive scene. It can only be lodged inside a space. It provides support for *mp3* and *wav* audio formats.

Figure 9.9.0.1 presents an example of an audio control. Figure 9.9.0.2 shows the configuration required for the example.

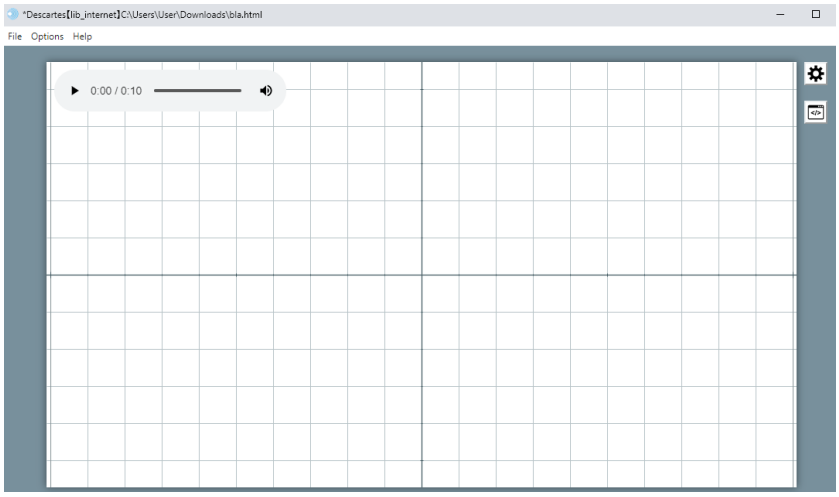


Figure 9.9.0.1: *Audio* control example.

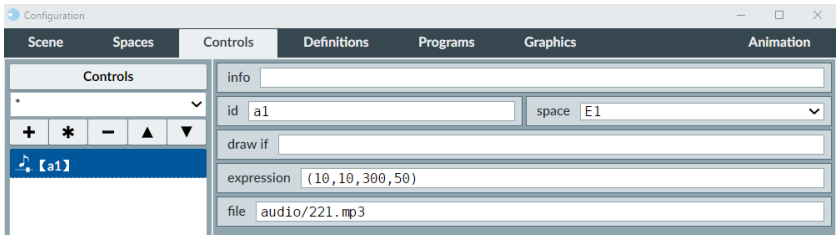


Figure 9.9.0.2: *Audio* control example configuration.

- **file**: A text field with the path and file of the audio file. Remember that the simple slash / is used to separate subfolders relative to the folder where the interactive scene is saved. For instance, *audio/noise.mp3*.

There are some functions and variables related to audio controls. In order to identify to which control they are associated, the control's identifier is used as a prefix. A more in-depth description of these variables can be found in the audio and video control functions

and the audio and video control variables topics. When the functions do not require arguments, no text is entered in their parentheses. For example, a <*identificador del control de audio*>.*play*() function via which the playback of an audio file is controlled.

We now do an exercise in which two different audio files, corresponding to two different frequencies, are played back when their respective options are selected in a menu. This exercise's interactive scene, along with the instructions to build it, can be found at Controls Audio. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file. The audio files used in this exercise can be downloaded from 221.mp3 and 371.mp3, and are also included in the *DescartesJSDocumentation.zip* file.

The way an audio control is displayed in a browser depends on the browser itself, and on its version. So, it will not necessarily be identical to the one displayed in the *Descartes* main editor. Additionally, it may also change in time due to changes performed in a same browser. Nonetheless, the controls inside the audio stay the same, even if their position inside the control may change.

Note that we had to add an animation with the sole purpose of refreshing the text printing the playback time. Should the user not need to print such data, the animation could be turned off. The Animation topic contains a more in-depth discussion on animations.

9.10 *Video* control

This control is a video player that can be included in an interactive scene. It is very similar to the *audio* control. It can play *mp4*, *WebM* and *ogv* files.

Figure 9.10.0.1 presents an example of a video control. Note at the top that the scene has been saved as *bla.html*. Figure 9.10.0.2 shows the configuration required for the example. A path to a file is included in the *file* parameter. In this example, the video file is housed in a *video* folder placed alongside the scene's *html* file.

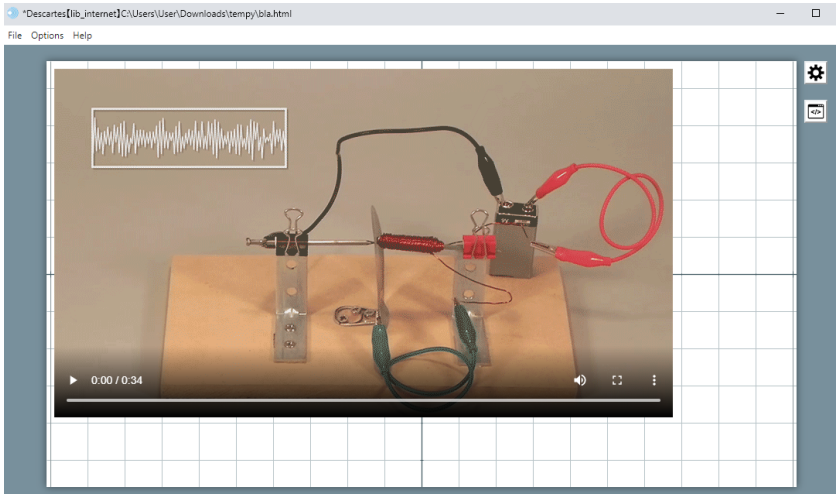


Figure 9.10.0.1: *Video* control example.

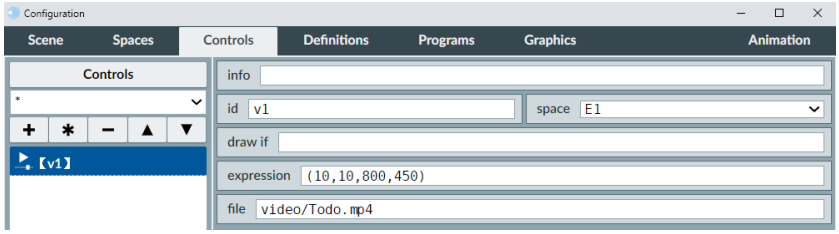


Figure 9.10.0.2: *Video* control example configuration.

- **file**: A text field where a path to a video file is entered. This path is relative to the folder where of the scene's *html* file is saved.

DescartesJS video controls have a few intrinsic functions: *<control id>.play()*, *<control id>.stop()*, *<control id>.pause()*, *<control id>.currentTime(<start time>)*. There is also an intrinsic *<control id>.currentTime* variable (note it has no argument parentheses as functions usually do), which holds the playback time in seconds. These intrinsic functions and variables are shared with audio controls. They are explained further in the audio and video control functions and audio and video control variables topics.

Similar to the audio control behavior, the video control in one web browser may differ from the one displayed in a different browser. For instance, some may have the button to display the video full-scree, some may not. It all depends on the browser's video player, and does not depend on *DescartesJS*.

Even though a minimum of two coordinates is required in the video control's *expression* parameter, we suggest including all four entries (x and y coordinates, and the *width* and *height* of the video) for better results. If possible, the width and height should be the same width and height of the video being used or, at least, have the same height to width ratio.

When the video control is first displayed, it appears stopped and the user has to click play to see it. There is no preview for the video, and the video control appears empty. If the programmer wants to display a preview in this state, a *png* image file has to be taken and saved alongside the video file and with the same name as the video (though its extension will be different). This image file should have the same dimensions as the video. Once the scene is reloaded, this image will be placed as a first-frame preview of the video. The example in figure 9.10.0.1 uses this functionality.

Since video controls are very similar to audio ones, no particular exercise is included for them.

9.11 Elements common to multiple controls

The parameters present for various controls are listed as follows:

- **info**: A text field to enter a description of the control in question. This information will not reach the scene, and is used only to aid the programmer so as to find a given control with ease. If something is entered in the *info* parameter, the control's default description in the list at the left panel will be replaced with the one in *info*.
- **id**: A text field where the control's identifier is entered. The identifier is sort of the "inner name" given to it inside the program. The user usually never knows the identifier of a given control. However, the identifier is usually related to a variable holding information about the control. For instance, the value of a numeric control is stored in its identifier. The identifier may also be indirectly associated to it. For example, a graphic control's identifier is the suffix to three variables related to it: its x position, its y position, and its state of activity. Since they are related to variables, no identifier can start with a number. However, they can start with an underscore (_) character.
  Identifiers for a control can be defined via the dialgo that pops-up when a control is added or cloned. Alternatively, they can be defined after the control has been added by simply editing the *id* parameter. Just remember the related variable will then be differently inside the program.
- **name**: A text field for the *name* of the control. This is **not** the identifier, but a name that can actually be displayed in the scene for the user to see. Though a control may have the same name as its identifier, it is good practice not to mix them. For example, a spinner control used to store a distance could have a *distSp* identifier, while its name could be *Distance control*.
  Additionally, since names are not variables, they can start with numbers and handle uncommon characters. If a control's name is not to be shown, this parameter is be left blank, or can simply contain a `_nada_` text.
- **gui**: A menu to determine the control's *graphic user interface*. Only *spinners*, *buttons*, *scrollbars*, *textfields* and *menus* have this menu. These are also the options

available in the *gui* menu. So these controls can be interchangeable. This may be useful when the design of a scene changes and this impacts on how the user will enter information. In this case, it may be necessary to change the interface of a given control.

- **region**: A menu set by default at *south* when adding a new control. It determines the region where the control is housed.
  - **south**: Places the control in the southern margin of the scene, outside from any space. Many controls can be simultaneously housed in this area. This area takes space from scene's dimensiones. So, when a control is added and left in this area, the available height left for spaces and other elements will be reduced. This area, as well as *north*, *east* and *west* usually house controls when they are to be displayed always. Otherwise, it is better practice to house them inside spaces.
  - **north**: Places the control in the northern (top) margin of the scene, outside from any space. This northern row sacrifices some of the scene's height, as does the *south* one.
  - **east**: Places the control in the eastern (right) margin of the scene, outside from any space. When this eastern bar is displayed, it takes space from the scene's dimension, and will therefore sacrifice some of the scene's width left for spaces and other elements.
  - **west**: Places the control in the western (left) margin of the scene, outside from any space. This western margin sacrifices width from the scene's width, as does the *east* one.
  - **external**: Places the control in the external region. This region consists of a panel apart from spaces and other regions. This region **can only be displayed if the *show external region* checkbox is marked**. This checkbox is located in the *Scene* tab. When marked, if the user right-clicks the scene, the panel will be displayed. All controls with their *region* set to *external* will be found in this panel, along with some other default buttons. This region is a good place to store controls that are not to be shown to the end user, and are only used for, for instance, debug purposes. Only the programmer will know of them and use them while building the scene. Once finished, the external region can be disabled and the end user need not know of them.
  - **interior**: Places the control inside a space. When this option is selected, the *space* parameter of the control activates. In it, the desired space can be selected where the control is to be logde. The *expression* parameter will also activate, where the absoluto coordinates where the control is to be located inside the space are entered, along with the control's width and height.
- **space**: A menu with the various spaces available as options. The selected one is

where the control in question is to be lodged. It is only active when the *region* parameter is set to *interior*.

- **draw if**: A text field where a boolean condition is entered. If true, the boolean condition has a 1 value and the control is displayed. Otherwise, the boolean condition has a 0 value and the control is not shown. Its default value is empty, which corresponds to the control always being displayed.
- **active if**: A text field where a boolean condition is entered. If true, the boolean condition has a 1 value and the control is active. Otherwise, the control is not active, though shown. Its default value is empty, in which case the control will always be active. Inactive controls may be shown, but are slightly gray and unresponsive.
- **expression**: A text field to enter the control's coordinates and its width and height. The expression typically is a set of parentheses inside which four numbers are entered and separated from one another via commas (,). The first two numbers are the *x* and *y* absolute coordinates of the control's top left corner. The third and fourth are the width and height (in px) of the control.
- **value**: A text field where the initial value given to the control is entered. Depending on the setting of the control, it may be a number, a character, a string of characters, or even a variable.
- **font**: A menu where the font used to print the control's name label is set. The available options are *SansSerif*, *Serif* and *Monospaced*.
- **font size**: A text field where the size (in points) of the font used to print the control's name label is entered.
- **bold**: A checkbox that, when marked, makes the control's name label be printed in bold.
- **italics**: A checkbox that, when marked, makes the control's name label be printed in italics.
- **label color**: A button that launches the color editor dialog to select the color and transparency of the control's name label background.
- **label text color**: A button that launches the color editor dialog to select the color and transparency of the control's name label text.
- **decimals**: A text field where a positive integer is entered, corresponding to the number of decimal places that are to be displayed when printing the control's value. If the true value exceeds the number of decimals allowed, the value printed is rounded so as to only use the allowed number of decimals.
- **fixed**: A checkbox that, when marked, forces the control's printed value to show all the allowed decimals set in its *decimals* parameter. If unmarked, these will only be displayed if they are significant.
- **exponential if**: A text field where a boolean condition is entered. If the condition is false, its value is 0 and the exponential notation will never be used when printing

the control's value. However, if the condition is true, its value is 1 and the control's value will be printed using exponential notation if the *DescartesJS* engine deems it necessary. Its default value is empty, which is equivalent to the control's value never being printed in exponential notation.

- **visible**: A checkbox that, when marked, results in printing the control's value and, when unmarked, keeps it hidden.
- **discrete**: A checkbox that, when marked, forces the values adopted by the control to be such that their differences with the control's initial value (the one given in the *value* parameter) are exactly multiples of the control's increment (the one given in the *incr* parameter). This works as described only if the increment is constant (not given via a variable), and has the same number of significant decimals as indicated in the control's *decimals* parameter.
  For example, if a spinner has its *value* parameter at 0.25, the *increment* at 0.15, is set to use 2 decimals, and the *discrete* checkbox is marked, it will start at 0.25 and, if increased, will change its value to 0.4. If it is decreased, it will adopt a 0.1 value.
- **incr**: A text field where a number is entered, that corresponds to the size of the increment allowed for the control. This applies to the decrement as well (in general, any change in value).
- **min**: A text field where the minimum value allowed for the control is entered. If left with its default blank value, no lower limit is set for the control. If the initial *value* given to the control lies below the minimum value, the control starts with the minimum value (it overrides the *value* parameter in such a case).
- **max**: A text field where the maximum value allowed for the control is entered. If left with its default blank value, no upper limit is set for the control. If the initial *value* given to the control lies above the maximum value, the control starts with the maximum value (it overrides the *value* parameter in such a case).
- **action**: A menu used to determine what action is to be taken every time the user uses the control. The available options are:

  calculate: the calculations in the *parameter* field right of the *action* menu will be done when the control is used. The can be value assignments to variables, arrays or matrices. Calls to functions can also be done here.

  init: the scene is loaded from its saved version when the control is used. This action is the same as when the *init* button (enabled in the *Scene* tab by marking the *button init* checkbox) is used.

  clear: any traces left from graphic objects (those set to leave a trace by having their *trace* checkbox marked), are cleared. This action is the same as when the *clear* button (enabled in the *Scene* tab by marking the *button clear* checkbox) is used.

  animate: if there is an animation defined in the *Animation* tab, it is launched when the control is used.

open URL: opens a URL address when the control is used. The address is specified in the *parameter* field right of the *action* menu. The path can be relative to the *html* file of the interactive scene, or it can be an absolute path to a web page.

open scene: opens a preexisting scene housed in the same folder as the current scene's *html* file. The scene is specified in the *parameter* field right of the *action* menu.

play: plays back an *mp3* audio file. If the file is in the same folder as the scene's file, only its name is entered in the *parameter* field right of the *action* menu. Files in subfolders are also enabled, using the / character as folder separator. The file is played / paused every time the control is used.

- **parameter**: A text field placed at the right of the *action* menu, with an "expand" button at its right. Single and short instructions or parameters can be directly entered in the *parameter* field. However, if more than one instruction is to be entered, or if the instruction is lengthy, it is better to do so via the "expand" button. It launches a larger editor where multiple lines (instructions) can be defined. Separate lines are separated with a semicolon (;) character in the single line *parameter* field.

  When the control's action is *calculate*, the paramter consists of assignments and / or call to functions.

  When the control's action is *open URL*, the parameter contains a URL address relative to the folder lodging the scene's file in a server. The URL address can also be an absolute address starting with *https://*. It is also possible to append the text `target=_self` in order to have the page opened in the same tab of the browser where the original scene is instead of in a new one (which is the default behavior). For example, `./tst.html target=_self` opens the *tst.html* page located in the same folder as the original scene in the same tab instead of in a new tab.

  When the control's action is *open scene*, the parameter is the name or path to the scene to be opened, along with its *html* file extension, relative to the calling scene's location. As with the *open URL* action, it is possible to include the `target=_self` text to indicate that the scene is to be opened in the same tab housing the original scene instead of in a new one, as by default.

  When the control's action is *play*, the parameter has the name or path to the file to be played back, along with its *mp3* file extension, relative to the scene's file location.

- **evaluate**: A checkbox common to text fields and menus that, when marked, enables the automatic evaluation of the control. This behavior is particularly useful when including evaluation questions.

- **answer**: A text field present for menus and text fields. It holds the elements to compare as "correct answer" patterns. Many patterns can be entered as correct, using the | character as separator. This parameter is only useful if the *evaluate* checkbox

is marked. The "correct" answers are listed and, if one of the user's answer matches, the answered is regarded as correct.

  - when the answer is numeric, the range of valid answers is an interval. For instance, [a,b], (a,b), (a,b] o [a,b).
  - for text field numeric controls with their *only text* checkbox marked, the comparison is done character by character.
  - the asterisk character (*) works as a wildcard ending or beginning. It can be placed at the end of a character string, and only the beginning of the string is compared to determine if the answer is correct. Alternatively, it can be placed at the beginning of a character string, and the ending of the character string is compared to determine if the answer is correct. When an asterisk is present both at the beginning and the end, it is only checked whether the character string in between is part of the user's answer to regard it as correct.
  - if the answer is not to be sensitive to capital or non capital letters, the proposed answer is simply flanked between single quotes. For example, *'answer'*.
  - if accents are to be ignored, or the difference between n and ñ is to be ignored, then the proposed answer is flanked between a grave accent (‘) and an acute one (´). For example, if the proposed answer were *'estaria´*, the *estaría* word would be also regarded as correct.
  - the question mark works as a wildcard representing characters whose exact match can be ignored. The ? is entered instead of the character to ignore.
  - if the control is a text field and is left empty by the user, the system considers no answer is given.
  - the evaluation system defined by the evaluation administrator is the one that decides how the correct, incorrect, or blank answers are to be interpreted.

- **keyboard**: A checkbox to implement the virtual keyboard. It is only available for controls involving a text field, such as the spinner, text field, menu and scrollbar.

  When the checkbox is marked, the *keyboard layout* and *keyboard position* parameters (described below) are activated.

  Virtual keyboards are particularly useful in mobile devices. By default, when clicking the text field of a control in a mobile device, the device's native keyboard is launched. These keyboards typically take up half the screen's area, reducing the scene's size so as to be able to fit the native keyboard. The *DescaretsJS* virtual keyboards appear **on top** of the scene, and the programmer can determine its size, complexity and position.

- **keyboard layout**: A menu with the following options: 14×1, 7×2, 10×2, 4×4, 5×4, $10 \times 4\_alfa$, $10 \times 4\_num$, 11×3 y 11×4. the number before the $\times$ symbol is the number of rows and the one after is the number of columns of the keyboard to use,

so the programmer has an idea of its size and complexity. The *_alfa* suffix in an option implies the keyboard is mainly alphabetic. The *_num* suffix implies a keyboard that is mainly numeric. Figure 15.3.0.1, in the virtual keyboard topic, displays the available virtual keyboards in the order in which they were listed previously.

- **keyboard position**: A text field for the absolute coordinates where the keyboard is to be placed. For example, a `(100,50)` keyboard position would place the top left corner of the virtual keyboard 100 px to the right of the left margin and 50 px below the top margin.

  For more information regarding this functionality, make sure to visit the virtual keyboard topic.

The *action* menu's options are executed whenever the control is used. This not only means using the control's main interface. For instance, though spinners are typically used via their increment and decrement buttons, a text or value can be entered in their associated text fields. When entering the text, their action (set via their *action* menu) is launched.

We are now ready to practice a bit of this functionality by doing an exercise. This exercise's interactive scene, along with the instructions to build it, can be found at Controls Common. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise allowed us to see some of the most typical functions of parameters common to most controls. The *wdthBr* control was placed external to the scene since the width of a figure is something that is to remain fixed in the interactive, and the end user will not have to deal with it. Perhaps the programmer may need to try some widths to see which one works best, but once one is chosen, the width control should not be made available to the end user.

10

The *Programs* tab

The *Programs* tab includes part of the hard programming work done in *DescartesJS*. It includes two algorithms present by default in a *DescartesJS* new scene: an *INICIO* one and a *CALCULOS* one. *INICIO* is Spanish for the starting algorithm, while *CALCULOS* is Spanish for calculations. Both are algorithms. The only difference is that the first one is done when the scene is launched, and the second is repeated constantly when the user interacts with the scene. Besides algorithms, the *Programs* tab has another element known as an *event*.

Just as the other tabs, *Programs* has a panel at the left where the programs involved in the scene are listed. This panel, as usual, has a + button to add elements to the list. It has a * button to clone the selected list element. It has a - button to remove the selected list element. And it has the upward and downward pointing arrows used to move the selected list element up or down in the list. The *Programs* button at the very top of the left panel displays the code related to the elements in the *Programs* tab. The elements available in the *Programs* tab are:

10.1 INICIO

The *INICIO* algorithm is present by default. Its name can be manually changed via its *id* parameter. It is set to be executed only once when the scene launches. Initial value assignments and calls to functions that are to be done only to prepare the scene are included in this algorithm. Figure 10.1.0.1 shows the parameters related to this algorithm.

- **id**: A text field for the algorithm's identifier. The default *INICIO* is entered for the existing one. It is good practice not to change it.
- **evaluate**: A menu to set when the instructions in the algorithm are to be carried out:
  - *only once*: The algorithm will be carried out only when the scene launches. This is the default setting for the *INICIO* algorithm. Assignments and function calls that are meant to prepare the scene only initially should be included in this algorithm. Since the instructions are done only at the beginning, they do not take up much of the system's resources, and therefore do not affect the scene's performance.
  - *always*: The algorithm is evaluated every time the user interacts with the scene. Instructions that are required to be done constantly should be included in algorithms that are evaluated *always*. Since these types of algorithms repeatedly

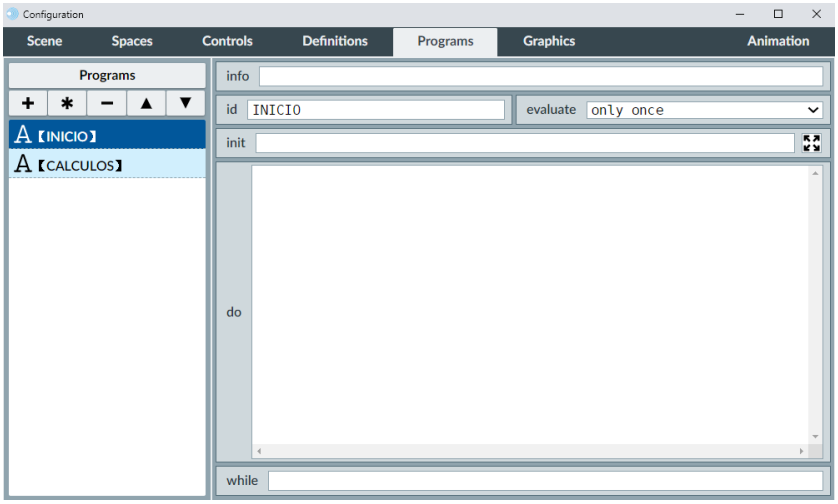


Figure 10.1.0.1: *Inicio* algorithm.

carry out the instructions, saturating this type of algorithms with many complex instructions may affect the scene's performance. It is best to keep algorithms using the *always* option as short and as efficient as possible. This option is the default setting for the *CALCULOS* algorithm.

- **init**: A text field in which the algorithm's initial assignments and function calls are entered. It can be written run-on in the same field, separating each instruction with a semicolon (;). Alternatively, the *expand* button at the right of the field can be used to expand the field to a window where different instructions can be entered in their respective lines (i. e., separating each one by an *ENTER*). Whatever is entered here is done prior to the instructions in the *do* panel, which can be set to repeat instructions (loop them) depending on the condition set on the *while* parameter.
- **do**: A text panel where multiple instructions can be entered. Each one is separated from the next by an *ENTER*. If the algorithm is to behave cyclically, the instructions in this *do* panel are the ones to be repeated, depending on the condition in the *while* parameter. If such parameter is left blank, the conditions here entered shall only be done once. In this situation, it turns out to be the same as to have them in the *init* parameter.
- **while**: A text field where a boolean condition is entered. If the condition is met, this parameter will have a 1 value, repeating the instructions in the *do* panel again. Otherwise, the parameter will have a 0 value, and the repetition of the *do* panel instructions will cease. In its default blank value, its value is 0 and the *do* panel instructions will be done only once.

  There is a limit to how many cycle repetitions can be done. A maximum of 100,000 repetitions are allowed. Even if the condition in the *while* parameter is always true, and more repetitions should be done, these will stop once that maximum number

is reached. This behavior is by design, and its purpose is to prevent the scene from crashing due to an incorrect *while* condition resulting in an infinite loop.

IMPORTANT: The *init* and *do* panels are the first example we see where the programmer enters instructions (assignments and function calls). Other places where these instructions are entered are the *calculation parameter* of controls, algorithmic functions instructions, etc. Besides this type of instructions, comments can also be added. Comments are lines of code that are ignored by *DescartesJS*, and usually serve as pointers so the programmer knows details about a particular part of the code. A comment line starts with // (double slash). Whatever comes after that in the line is ignored by *DescartesJS*.

10.2 CALCULOS

The *CALCULOS* algorithm is an algorithm just like the *INICIO* one, its sole difference being that the *evaluate* parameter of the *CALCULOS* one is set to be evaluated *always* by default. Apart from that, it behaves as the *INICIO* one.

Only the two algorithms added by default to a new *DescartesJS* scene are the ones available. No other can be added. Even though the *evaluate* parameter can be changed from its default value, we suggest keeping them as they are and entering calculations to be done initially in the *INICIO* and those to be done repeatedly in the *CALCULOS* one.

Let us try an exercise to practice the use of these two algorithms. This exercise's interactive scene, along with the instructions to build it, can be found at Algorithms Various. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file. By the way, the *INICIO* algorithm provided in this example already contains a few additional instructions that are not part of the steps to follow when building the scene. They are there as a means of communication between the scene and the container in which it is hosted, and can therefore be ignored.

This exercise allowed us to see how the *INICIO* algorithm is used for instructions happening only once, as opposed to the *CALCULOS* one used for instructions that are done repeatedly. Once again, the programmer should be careful not to saturate the *CALCULOS* algorithm with complex instructions, so as not to harm the scene's performance.

Once again, if the *while* parameter of the algorithm in question is left empty, the *do* panel instructions will not be looped. This means that the instructions could be placed either at the *init* or the *do* panels.

10.3 Events

An event is an action, or a group of actions, that are done when a certain condition is met. The frequency with which they are performed is up to the programmer, as described below. The list of actions that can be launched when the condition is met are the same

that as the ones that can be performed with a control's *action* parameter. Figure 10.3.0.1 presents the various parameters of an *event*.

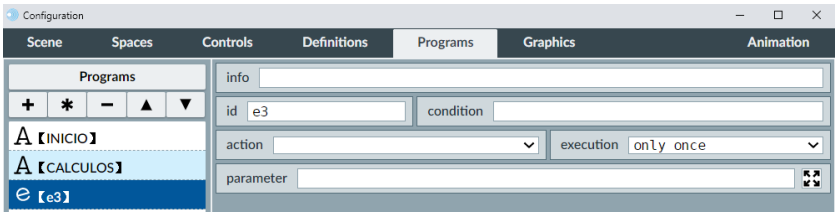


Figure 10.3.0.1: Event program example.

- **id**: A text field for the event's identifier. Event identifiers are mainly used to locate a particular one from the list, though references to them are seldom used in other parts of the program.
- **condition**: A text field where a boolean condition is entered. When the condition is met, and depending on the *execution* parameter setting, the event's *action* is launched. For more information on these conditionals, review the boolean operators and conditionals topic.
- **action**: A menu to determine the action that will be launched when the condition is met. This menu has the same options as the action menu available in controls. For more information, please review that section.
- **execution**: A menu to determine the event's repetition whenever the condition is met. It has the following options:
  - **only once**: The menu's default option. When this option is selected, the action will be executed only once when the condition is met (when it transitions from being false to being true). After this, if the condition eventually transitions from false to true again, the action will not be executed again, since the menu is set to *only once*.
  - **alternate**: When this option is selected, the action will be executed every time the condition transitions from a false value to a true one.
  - **always**: When this option is selected, the action will be executed every time the condition is true, and not only when it transitions from false to true.
- **parameter**: A text field where the parameter of the selected *action* is entered. For example, for the *open URL* option, the parameter is a hyperlink to the URL, whereas for the *action* option, the parameter is the set of instructions to follow. The parameters for each action are described more in depth in the action menu parameters topic. This text field has an expand button to its right. When clicked, it displays a text editor which supports multiple lines, which is particularly useful when the parameter involves many instructions.

We are now ready to do an exercise and practice the use of events. We focus the attention on the different execution modes. The purpose of the scene is to include an image that sticks to the mouse when it is pressed. This exercise's interactive scene, along with the instructions to build it, can be found at Algorithms Event. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise allows us to see that a condition, by itself, is insufficient to specify what the event is to supposed to do. The *execution* parameter further specifies how and when an event's action is implemented.

The exercise involves mouse related variables intrinsic to *DescartesJS*. For more information, please visit the mouse variables topic. Since conditionals are also involved throughout the exercise, it may also be useful to review the boolean conditions and operators topic as well.

Though there is an *animate* option in the *action* menu of a button, sometimes a button used to animate also has other functions besides launching the animation. A workaround for this is to use *calculate* as the button's action, and besides everything else the button is supposed to do, an variable's value can be changed. This change in value can then be associated to the animation via an event. This way, the button does all the other calculations it needs to do, plus the variable's value change related to launching the animation.

11

The *Definitions* tab

Definitions is a tab in *DescartesJS* that also includes a good deal of a scene's proper inner programming. There are many kinds of definitions: *variables*, which can adopt a value or the value returned by an expression; *arrays*, which can be better understood as a variable name associated to a numbered set of values; *matrices*, whose functionality is similar to that of arrays, but involving tow numbered sets of values; functions, which are either expressions that return a value, or algorithms that comprise a set of different instructions; and libraries, which are a means to group the other various definitions so as to have a clearer and better ordered code in this tab.

From here on, we visit each of the definitions in more detail. We also pay special attention to the ways in which these definitions allow for a better, more simplified and agile handling of the data involved in a scene. They may also be used to significantly reduce the amount of code in an interactive.

Figure 11.0.0.1 shows the *Definitions* tab. The element displayed is a matrix. As can be seen in the figure, this tab also has a left panel where the definitions are listed (in this example, there is only one present: the *M1* matrix). This panel has all the buttons with which we are already familiar: the + to add a new definition, the * to clone the selected definition, the - to delete the selected definition, and the up and down arrows to move the selected definition up or down in the list. A recent *DescartesJS* upgrade is a filter. This filter, marked red in the figure, displays a list when clicked of all the available *DescartesJS* libraries, as well as an *escena* (*scene* in Spanish) option. Libraries will be dealt with later on. Suffice it to say that they are compendiums of various definitions used to have a better organization of the definitions. So, this menu can be used to only show definitions corresponding to a particular library, or to show the definitions that do not belong to a library, but directly to the scene (those displayed when selecting the *escena* option).

11.1 *Variable* definition

The *variable* definition involves a variable, which is the identifier of this type of definition. A value can be assigned directly to the variable in the parameter after the equal (=) sign. Alternatively, a variable can be also assigned an expression (for instance, `2*3`), or even a function (for instance, `abs(z)`, to calculate the absolute value of a *z* variable). Since, in its most general sense, it is assigned a function, this type of definition has fallen into disuse, and the *function* definition is now favored over it. More information on this type of definition can be reviewed in the function definition topic, which will be addressed

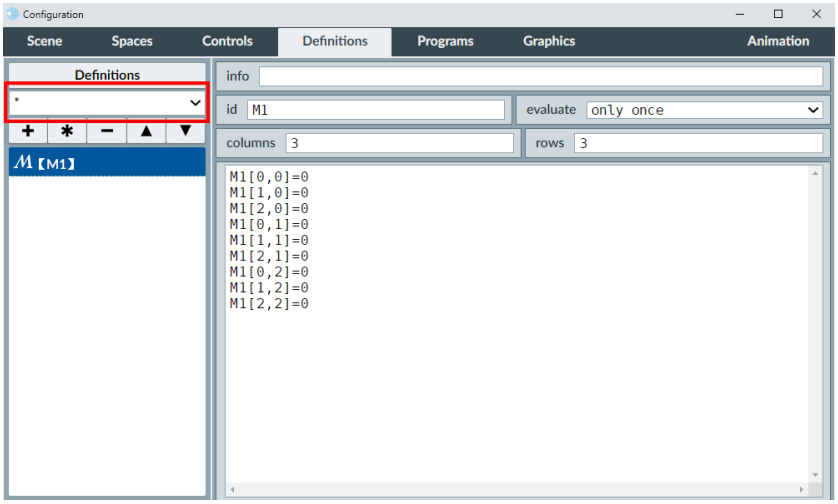


Figure 11.0.0.1: The *Definitions* tab. The red box shows the definitions' *filter*.

shortly.

Figure 11.1.0.1 presents an example of the *variable* definition. Figure 11.1.0.2 shows the configuration required for the example.

Once again, note that the variable in this example is assigned an expression corresponding to the distance of an $(2,3)$ point to the origin: $\sqrt{x^2+y^2}$. The text in the scene prints the value of *v1*, the variable's identifier. Obviously, the x and y variables have been assigned their respective 2 and 3 values previously. One way to assign them their values can be via the *INICIO* algorithm. This way, when the variable does its calculations, the x and y variables involved are already initialized.

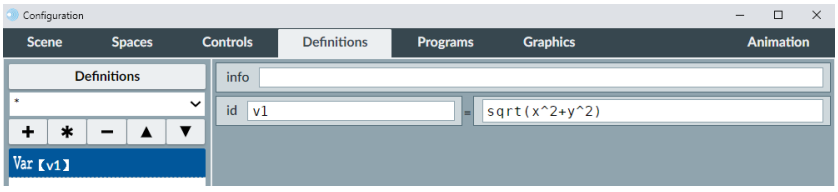


Figure 11.1.0.1: *Variable* definition example

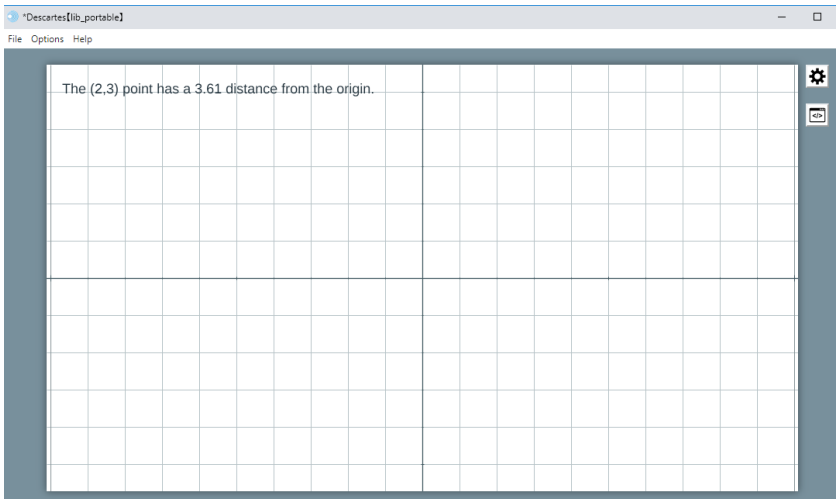


Figure 11.1.0.2: *Variable* definition example configuration.

- **id**: A text field with the variable's identifier. The identifier is the variable, as such, which holds the value returned by the expression at the right hand side of the equal (=) sign.
- **the value of the variable**: A text field, at the right of the equal (=) sign, where a value or expression (including functions) is entered. The expression is evaluated, and the value returned is assigned to the variable.

We are now ready to try an exercise to practice and better understand this type of definition. This exercise's interactive scene, along with the instructions to build it, can be found at Definitions Variable. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise shows that variables can, in their most general functionality, receive a value from an expression. Functions do the same thing. However, since functions have a more general functionality (they can receive arguments and behave as cyclic algorithms), they are usually preferred over variables.

11.2 *Array* definition

Sometimes, a single variable just is not enough when the programmer wishes to store a lot of information. Though a lot of different variable could be added, this usually turns out to be exhausting and the control on single variables is lost when handling many of them. *Arrays* are a type of definition that allow us to tackle this problem. You can think of them as "storage units with many drawers". The name of the unit is the same, but each drawer is numbered or indexed.

When an array is added, the name of the storage unit is the array's identifier. For instance, we can speak of a *V1* array. The *V* letter is used as their default prefixes, since arrays are called *Vectores* in Spanish. Once an array exists, the value held in its n-th drawer is

stored in a variable with the identifier as prefix, followed by the index (*n*) flanked by square brackets. For example, if a *V1* array has already been defined, its 5th drawer is *V1*[*4*]. Note that the drawers are counted starting from the zeroth one. Hence the relation of the 5th drawer (counting from 1) with the 4 index in the array (counting from 0). Each index of an array (each drawer) can store the same type of information as a regular variable: a number value, or string of characters.

Figure 11.2.0.1 shows the parameters of an *array* definition. Note that, when adding an array, it has a few assignments in its main panel. It has a default size 3 value, and so, its first 3 entries are initialized with a zero value. This initial assignments are usually dispensed with.

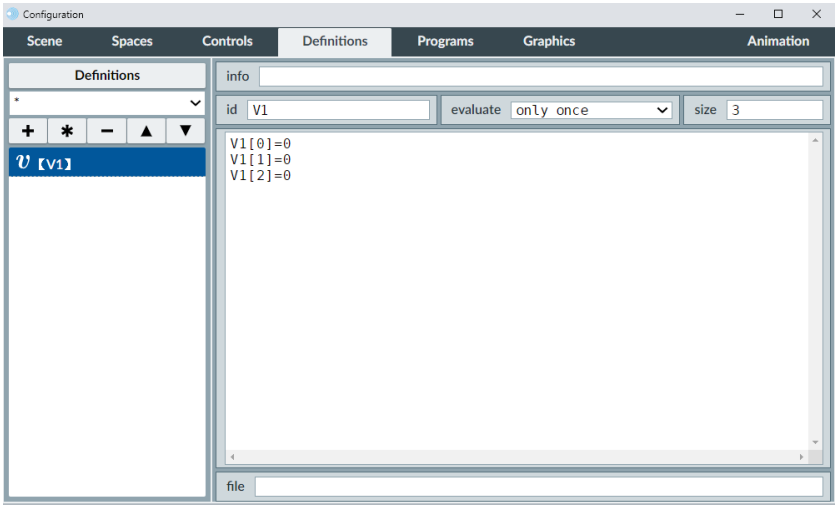


Figure 11.2.0.1: *Array* definition example. Note the default zero value assignments for the three entries of the array.

- **id**: A text field where the array's identifier is entered. The identifier is the means to make a reference to an array. Each of the array's entries are referenced to using the identifier as a prefix, followed by the index of the entry flanked by square brackets. Remember the index starts counting from zero. For example, the 3nd entry of a *Vc* array is *Vc*[*2*].
- **evaluate**: A menu with a single *only once* choice. This option refers to whether it is evaluated when the user interacts with the scene, or only at the beginning. It is usually ignored, since it has only this option. When the contents of an array are to be modified, this is usually achieved via a function, and so the *evaluate* parameter can be safely ignored.
- **size**: A text field where the number of entries (of drawers) are entered. Remember that if *n* entries are considered, the highest index available is then *n-1*, since the starting index is 0 rather than 1.

- **array's initial assignment panel**: A multi-line text central panel where several assignments are done. Though this panel is related to the array, its assignments need not only deal with the array's entries. Other variables may be initialized here as well. As mentioned, since entries are usually assigned values via a function, this panel is frequently left blank.
- **file**: Array entries can be initially filled using data from a text file. This *file* parameter is the text field where the path leading to such a file is entered. The path is relative to the location of the scene's *html* file. This is particularly useful when dealing with very large arrays which require being loaded from the results of, say, a simulation. In such examples, entering the data by hand can prove to be tasking and prone to mistakes.
  The recommended text coding for the file is UTF-8. Each entry value assignment is entered in a new line.
  This file-to-array functionality is available both in the editor as in browsers such as Mozilla Firefox. In Chrome, this functionality may not be initially visible. There may well be trouble reading the file due to some of the browser's security settings. However, it is possible to disable them in Windows using the `-disable-web-security` tag when launching the browser from a command window.
  In order for an *html* scene generated in *DescartesJS* to successfully assign values from a file to an array, the scene's *html* file must first be saved in some folder and the path to the text file should be a valid one relative to the scene's location. Otherwise, the text file will not be found.
  It is also possible to save the text file **inside** the scene's *html* file as a *script*. Figure 11.2.0.2 shows an example of the scene's *html* file code with a *script* block near the end of the file. The *DescartesJS* code is the one in the block above, inside the `<ajs></ajs>` block. After all that block, and its contiaining *div* block, the *script* block can be found. The entries will be assigned the values in the list. The *file* parameter of the *array* definition for this example should be the same as the indicated in the scripts *id* (that is, `./fle.txt`). Note that the script's type involves a `vectorFile`, since arrays are called *vectors* in Spanish. For a *V1* array, its entries will be assigned the values listed in the script. Some browsers' security settings do not allow them to read data from local files. This may result in a *DescartesJS* scene not being able to read a text file holding the values. However, if the data is embedded in the *html* file via a *script*, then it can be successfully read. Though this constitutes a workaround, it is important to be aware that this may result in large sizes for the scene's file.
  An array can initially be filled using a text file as described above. If the programmer wishes to include the file's data **also** as a script inside the *html* file, it is first necessary to have the *array* sub-option marked in the *Add To HTML* option of the *Options* menu is selected. When the scene's file is saved, it can then be opened in a text editor and the corresponding *script* block is present near the end of the file. For more information on the aforementioned option, review the Add to HTML topic.

Now that we are more familiar with an array's functionality, let us do an exercise in

```
      <param name="E_01" value="tipo='R2' id='E1' ancho='100%' alto='100%' ">
      <param name="A_01" value="id='V1' vector='si' evaluar='una-sola-vez' tamaño='3' archivo='./fle.txt' tipo='vector' ">
      <param name="A_02" value="id='INICIO' algoritmo='si' evaluar='una-sola-vez' ">
      <param name="A_03" value="id='CALCULOS' algoritmo='si' evaluar='siempre' ">
      <param name="G_01" value="espacio='E1' tipo='texto' color='20303a' coord_abs='si' expresión='(20,20+17*s)' familia='s'
      s.intervalo='[0,5]' s.pasos='5' texto='[V1[s]]' ancho='1' align='a_left' anchor='a_top_left' ">
    </ajs>
  </div>

<script type="descartes/vectorFile" id="./fle.txt">
1.5
2
3
3.7
2.89
6.21
</script>

</body>
</html>
```

Figure 11.2.0.2: Array *Script* inside the scene's *html* file. An array's entries can be filled from this data.

which a couple of die are thrown, and a frequency table is built for all the possible results of adding the die's values (the possible values ranging between 2 and 12). The frequencies are saved as values of an array's entries. This exercise's interactive scene, along with the instructions to build it, can be found at Definitions Array. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise can be used to show how, when throwing 2 die, it is more likely to get a value sum of 7 than any other value sum. This responds to the fact that to get a 2 sum, each dice has to have a 1 value, whereas to get a 7 sum, there are many more configurations: 1 and 6, or 2 and 5, or 3 and 4, or 4 and 3, or 5 and 2, or 6 and 1 (6 different configurations). By the way, a suggested extension to this exercise could be to graphically represent the frequencies using a histogram. This can be achieved by using a family of rectangle graphic objects.

We were also able to see that arrays reduce the programming work. Instead of adding 11 different variables, a singe 11 entry array was used. Array's indexes can also be related more easily to graphic objects (the printed text in this example). If 11 variables were used instead of the array, 11 lines of text would have been necessary, instead of a single line of text while using the *family* functionality of graphic objects.

Since functions are the most useful choice to assign values to array's entries, please be sure to review the function definition topic. Another functionality used in this exercise was the use of the *rnd* variable to generate random values. The *DescartesJS* general variables topic contains more information on this functionality.

One other thing: it is always good practice to declare first the array and afterwards the function (or functions) handling it. This means it is better to place the array **before** the function using it in the list of definitions in the panel a the left inside the *Definitions* tab. *DescartesJS* usually no longer presents a problem if this instruction is not followed. However, it is a suggestion to potentially improve the scene's performance.

Finally, note the close relationship between functions and arrays (which is also extensive to functions and matrices). A cyclic algorithmic function can be made to sweep a

value related to the index of an array. In this way, a few lines of code can assign value to a great number of entries.

11.3 *Matrix* definition

We have just seen how powerful a tool arrays can be. However there is an even more powerful tool at storing information known as a *matrix*. Instead of handling entries only via one index, it uses two. So, a matrix can be thought of as a two dimensional unit comprised of lines and columns. One index handles in which line the drawer is, while the other handles in which column it is found.

Matrices can be thought of as a table of values as in a spreadsheet. To point to a specific value of the table, the square brackets following the matrix's identifier now hold not one, but two, different positive integers separated by a comma. For instance, *M[3,7]* would be the fourth column, eighth row entry of the matrix with an *M* identifier. Remember the indexes start counting from 0 and not from 1; hence the 1 unit discrepancy. The notation is, therefore, *M*[*#column,#row*].

Figure 11.3.0.1 shows the parameters related to a matrix.

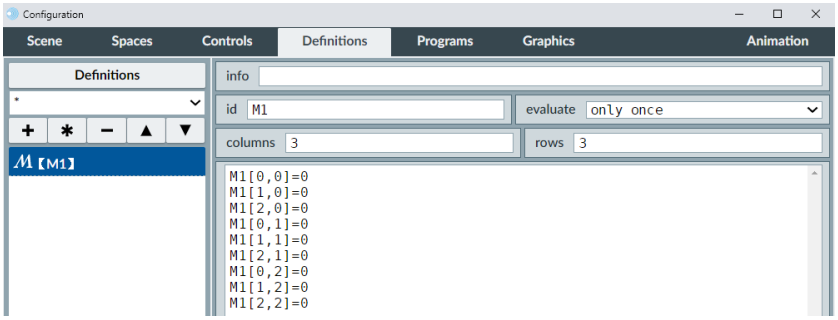


Figure 11.3.0.1: *Matrix* definition example.

- **id**: A text field for the matrix's identifier. Whenever a reference to a matrix is made, it is done via its identifier. As already mentioned, the reference to a particular entry is *M*[*#column,#row*].
- **evaluate**: A menu with a single *only once* choice. This option refers to whether the entries of the matrix are evaluated when the user interacts with the scene, or only at the beginning. As with arrays, it is usually ignored, since it has only this option. When the contents of an matrix are to be modified, this is usually achieved via a function, and so the *evaluate* parameter can be safely ignored.
- **columns**: A text field to enter the number of columns making up the matrix. Since matrices are two dimensional, they have a *columns* and a *rows* parameter; not only a *size* one as arrays.

- **rows**: A text field to enter the number of rows making up the matrix.
- **matrix's initial assignment panel**: A text introduction panel where the initial assignments for the entries of the matrix may be entered, as well as other instructions not necessarily related to the matrix. As with arrays, it is usually left blank since assignments are typically done via a function.

We can now do an exercise to note the usefulness of the matrix element. This exercise handles a large number of particles contained in a box. Some are red and some are blue. The color and initial positions are assigned randomly. Each particle has two coordinates (a particle's horizontal and vertical components). This exercise's interactive scene, along with the instructions to build it, can be found at Definitions Matrix. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise shows how matrices allow for an easy manipulation of large amounts of data. Had this scene been attempted using arrays only, several arrays would have been necessary (one dealing with a particle's color, and a couple other dealing with its components). Other examples could possible require even more elements, and therefore more arrays. Matrices allow to join all these into a single identifier. This, in turn, allows for an agile an compact data manipulation.

Through this exercise, we were also made aware that a same matrix can handle different types of data. Some entries stored numeric data (the coordinates), while others stored a character string (*blue* or *red*).

Furthermore, note that it was a function again which dealt with assigning the values for the matrix's entries.

Conditionals were also used to assign the color of the particles. The boolean conditions and operators topic can be reviewed for more information. Additionally, since the color of the particles were changed, remember the color editor topic deals with this functionality.

11.4 Function

Functions involve a set of instructions grouped under a same entity: the function itself. These instructions can be set to loop given a certain condition, or can be implemented only once. All this is decided depending on the expected functionality of the function. Functions may well be considered the nucleus around which the general programming of a scene is gathered.

Figure 11.4.0.1 shows the elements of the *function* type definition.

- **identifier of the function**: A text field where the identifier is entered. For functions, the identifier is the function's name itself, and has a set of parentheses where the function's arguments are entered separated by commas (,). If the function uses no

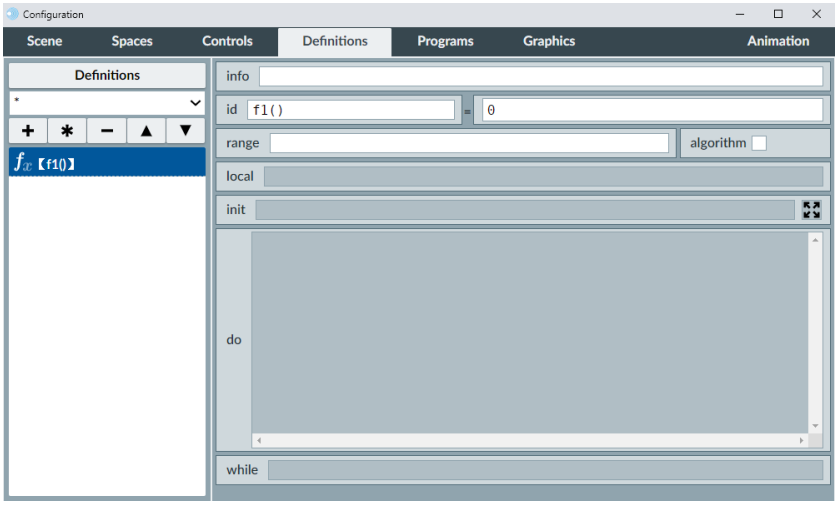


Figure 11.4.0.1: *Function* definition example.

arguments, the parentheses should still be included, though nothing is entered between them.

- **return value of the function**: A text field at the right of the equal (=) symbol, where a value or expression is entered. This value, or the value of the evaluated expression here entered, is what the function will return when finished. Functions are sometimes assigned to a certain variable. When the function is run, the variable to which the function is associated will receive this return value. Functions can sometimes be set only to run, changing various variables of a scene, without any need to return a particular value to a variable. When no return value is required, this text field can be left blank.
- **domain or range**: A text field where a condition is entered. The function will only be evaluated in points where the condition is true. For example, a function depending on x with a $(x > 1)\&(x < -1)$ domain will exclude the $(-1,1)$ interval when being evaluated.
- **algorithm**: A checkbox that, when marked, enables the functions *algorithmic* functionality. When enabled, the *local*, *init*, *do* and *while* parameters are activated.
- **local**: A text field where the variables that are to be local to the function only are entered, separated by single commas (,) or semicolons (;). A local variable here entered can only change its value **inside** the function. Suppose there is an i variable used both outside the function and inside the function. But they are in charge of different things. You would not want the function changing the value of the i variable inside the function, and affecting the value of the one outside. So, what *DescartesJS* does when this i is entered in the *local* field, is that it internally assigns it a different name, so as not to confuse them with outer variables with the same name. However, if the intention of the function is to change the value of this i to impact outside the function, then it should not be entered in the *local* field. The *local* field therefore

"protects" same name variables inside the function.
Additionally, it is important to bear in mind that, when a function involves arguments, these are treated as internal local variables as well.

- **init**, **do** and **while**: These text fields behave in the same way as the ones in the INICIO and CALCULOS algorithms. They are active only when the *algorithm* checkbox is marked.

A quick example of a function might help clear some doubts. Consider a *calcHyp(x,y)* function with a return value *sqrt(x^2+y^2)*. Its purpose is to calculate the length of the hypotenuse given the lengths of the two legs. This function has a *calcHyp(x,y)* identifier. The identifier already includes two different arguments, which enter the function as local variables *x* and *y*. Outside the function, the user could assign the value to an *hy* variable. For instance, if the lengths of the right triangle are *leg1* and *leg2*, the call to the function could be *hy=calcHyp(leg1,leg2)*. This passes the lengths of the legs as arguments, which inside the function will be regarded as the *x* and *y* variables. This endows the function with a general behavior. If another right triangle is present, the same function could be used just changing the arguments in its call. Note also that the function is assigned to a *hy* variable. So the function's return value (the length of the hypotenuse) is assigned to that variable.

We now do an exercise to practice the use of functions. The purpose here is to do an interactive scene that indicates the length of the three sides of a triangle whose vertices are graphic controls. This exercise's interactive scene, along with the instructions to build it, can be found at Definitions Function 1. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise involved a single function that returns a value. Note that, in order for a function to return a value, the text field after the equal (=) symbol has to have the expression of the value to be returned. In this case, such expression is the square root of the sum of the square of the legs of the rectangle triangle whose hypotenuse is flanked by the (ax, ay) and (bx, by) points. Note that the text field can have a value, a variable, or (as in the present case) an expression for the value to be returned. The returned value is then assigned to the variable of choice for the function (in this case, the *distg1g2* and other distance variables).

We only calculated three distances. However, they could have easily been twenty, and a single function would suffice. The function is generic and works as a single instruction with the potential of being used many times, even though its code is only entered once.

This exercise used *g1.x* type variables and $sqrt()$ functions. All these variables and functions are intrinsic to *DescartesJS* and can be reviewed in the graphic control variables and functions common to various programming languages topics.

Let us try a slightly different exercise involving a function that does not return a value, and that is used to do several assignments at a time. The idea behind the scene is for it to indicate not only the distance between two graphic controls, but also to print the

horizontal and vertical distances (the absolute horizontal and vertical differences between the controls). This exercise's interactive scene, along with the instructions to build it, can be found at Definitions Function 2. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise allowed us to see that, in some situations, functions need not return values, but only do several calculations. These calculations can be done by turning on the *algorithm* functionality of the function. Additionally, not all functions require arguments. They can directly use the scene's variables (such as *g1.x*, *g1.y*, *g2.x* and *g2.y*).

A valid question that arises is: can the instructions inside the function be placed directly inside the *do* panel of the *CALCULOS* algorithm? The answer is yes. However, the three assignments have to do with a similar objective: calculating distances. So, a better option is to group them inside a single function, and the function is then called inside the *CALCULOS* algorithm. This results in a better ordered program structure. This structure is more modular. If, at any time, the three assignments need be made elsewhere in the program, there is no need to do them individually, since a single call to the function takes care of that.

This exercise used *g1.x* type variables, $sqrt()$ functions, and $abs()$. All these variables and functions are intrinsic to *DescartesJS* and can be reviewed in the graphic control variables and functions common to various programming languages topics.

Let us do a more serious exercise with a function that involves the calculation of the greatest common divisor of two positive integers using Euclid's algorithm. This exercise's interactive scene, along with the instructions to build it, can be found at Definitions Function 3. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise involves a different, more complex, type of function: a recursive or iterative function. In this case, the instructions are repeated while a condition is true. After these repetitions, or iterations, the function spits out a value: the greatest common divisor, or *gcd*. Note that this *gcd* is useful to simplify fractions. This is done by dividing the numerator and denominator by their *gcd*. It can also be used to calculate the least common multiple (*lcm*) of two positive integers. This *lcm* is obtained by dividing the product of the numbers by their *gcd*.

A boolean condition is used in this exercise in the *while* parameter of the function. This functionality can be reviewed in the boolean conditions and operators topic. Additionally, an *ent()* function was used. This function, along with many others, is addressed in the functions common to various programming languages topic.

11.5 Library

Certain *DescartesJS* scenes can be complicated enough to contain a lot of elements in the *Definitions* tab. This may result in difficulty finding one particular element. Besides, some

elements in a scene may require constant upgrades, and should be readily available, while others may be set in stone since their functionality is static.

The *library* definition is a tool which allows the grouping of certain definitions. Which to group depends on the programmer. However, certain definitions which serve a single purpose, or do similar actions, may be grouped together in a library. And once they reach their end state and are not to be subjected to further edition, the definitions can be stored in a library if the programmer sees that as a good option.

Placing definitions as a part of a library also makes it easier for the programmer to find them. There filter functionality already discussed in Figure 11.0.0.1 allows the user to only display the definitions of a selected library, or those of the scene (those not grouped as a library).

A library is stored as a text file. The path to it is given relative to the folder where the scene's *html* file is stored. Even though the file is external to the scene itself, its information can also be included inside the scene's *html* code.

Figure 11.5.0.1 shows the configuration of a library. Its text file is stored in a *libraries* folder placed in the same folder as the scene's *html* file. The example in mind is a group of definitions dealing with the engine of a mechanical simulation. *The simulation engine* description is entered in its *info* panel, so that the programmer can easily find all the definitions of this engine neatly grouped.

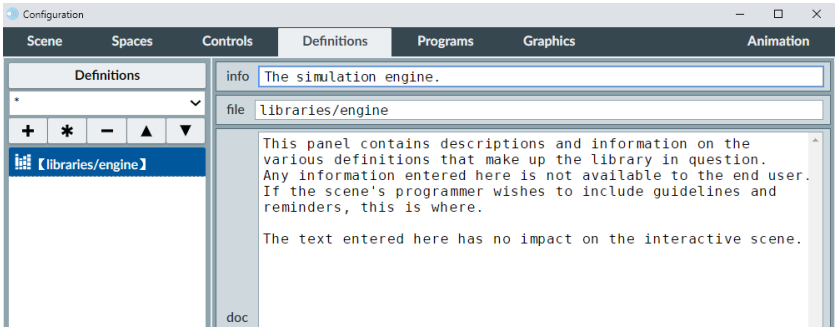


Figure 11.5.0.1: Library *definition example.*

- **file**: A text field where the path to the library's text file is entered relative to the scene's *html* file folder.
- **doc**: A mutli-line text panel where the programmer writes down notes, reminders, definitions, pointers, etc. related to the library. The text entered there is not code, so it will not impact on the scene's functionality in any way. It is for the programmer's benefit only, and is therefore not intended for the end user.

A library can be imported in different scenes. So, it is also a way to optimize code by avoiding its repetition in different scenes. And, if the library is to be embedded as part of the scene's *html* code, it can be directly edited when editing that particular scene.

Let us try an exercise to have a better grasp of this functionality. This exercise's interactive scene, along with the instructions to build it, can be found at Definitions Library. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise has many interesting features. On the one hand, we see that functions that have been completely defined and need not further changes (such as *CalcAverage*() and *CalcStdDv*()) can be stored in a library so they are not in the way when searching for other specific definitions. This particular example only has two such functions, but a more complicated scene may have many more. It would prove difficult to find a specific definition in a scene involving, say, 60 or more definitions. This is where grouping them in libraries comes in handy.

It is always possible to remove the *engine* library by extracting its content and adding it to the *Definitions* input panel launched by clicking the button at the top of the left panel in the *Definitions* tab. This will ensure the definitions are available in the scene's definitions filter, and not in the library's one. If a library is to be removed, it is also a good practice to remove the information from the *script* block in the scene's *html* file (if it was decided to show its code via the *Add to HTML > library* option in the *Options* menu). Besides removing the script, the *library* definitions should be also removed in the *Definitions* tab.

When manually editing a library text file, the file's coding should be set to *UTF-8* (preferably *without BOM*). Many text editors save their file using *ANSI* coding by default. This coding frequently enters characters which are not interpreted by *DescartesJS*, resulting in failure to input the information correctly. If the file's coding is *UTF-8*, the information will be correctly read.

There is something worth mentioning regarding the exercise. Note that the *i* variable used as a counter is used in 3 different functions (*AssignValues*(), *CalcAverage*(), and *CalcStdDv*()), this does not constitute a problem. This, however, does not represent a problem, since each of these functions is completely done at a different moment than when another is done. A problem would arise if the functions overlapped: if one started while another was halfway through. For example, when a function calls another inside it, and both have the same counters, there is bound to be an error. If this were the case, the solution would be to set the *i* variable in the functions as a *local* variable (see the local topic for more information). All this is also true for the *sum* variable: both *CalcAverage*() and *CalcStdDv*() functions use the same variable. Yet, since there is no overlap, no problem arises.

Another important detail can be extracted from this exercise: the order in which the *CalcAverage*() and *CalcStdDv*() functions are called. It is imperative to run the former before the latter, since the latter uses a value required from the former: the *average* variable. If they were called the other way around, the *average* variable would not have the correct value with which to calculate the standard deviation.

One last thing to notice is that the contents of the *engine* library in this example can be accessed via the filter in the left panel of the *Definitions* tab, **only once the scene with the included library has been saved**. Once the library is selected in the filter menu, its

definitions appear listed in the panel (they are filtered from all the definitions associated to the scene). There, the individual definitions can be selected and edited as well. Any edition performed in one such definition will also be saved in the library's text file, and in the *script* block inside the scene's *html* file code (if the *Add to HTML > library* option is implemented in the *Options* menu) when the scene is saved in *DescartesJS*. This allows for a more agile handling of libraries' information as opposed to manually editing the *script* block and the text file.

Having the *script* block present as part of the scene's *html* file ensures the library's code will **always** be available. In some situations, a web browser will not have permission to access content external to the *html* file being displayed. This results in the library's definitions not being available. However, if the library is stored in the *script* block, they will be available even of the text file is missing.

Note that text files need not have their *txt* file extension. If they do, however, the file extension must be included in the *file* parameter of the library for it to be read correctly.

To get the most out of the libraries, definitions similar to each other, or that deal with a specific functionality, can be grouped together in the same library. For example, functions and arrays necessary for a mechanical simulation could be in one. While functions dealing with the correct placement of graphic objects could be in another. The functions in a library will be displayed when selecting that library in the *Definitions* tab filter menu. As a reminder, the *scene* option in the filter filters the definitions associated directly with the scene. And the * option of the menu list the scene's definitions along with the libraries included.

A scene may have two identical definitions. For example, consider two functions with the same identifier, but that follow slightly different instructions. One may be imported from a library and the other could be set as a function directly belonging to the scene. The one that takes precedence is the one belonging directly to the scene. Though it is possible to juggle both, it is better to have only one version. This scenario usually happens when a function that is part of a library is to undergo further edition. The original is kept in the library as a backup, and its copy is placed as though it belongs to the scene. The copy is the one being edited and tested. And, since it is in the scene, the programmer knows this is the one being tested. Once its correct functionality is confirmed, the backup can be overwritten with the new version, both in the *script* block and in the library text file.

Once again, remember that arrays read from file, as well as libraries, can have their information stored inside the scene's *html* code via a *script* block. To do this, the Add to HTML submenu in the *Options* menu (in the *DescartesJS* main editor) should have the *array* and *library* options marked. All the options in this submenu are marked by default. Thanks to this default behavior, the respective *script* block was always present in the *html* code when the exercises' final scenes were saved.

12

The *Animation* tab

Animations are used to visualize changes in the scene in real time, instead of skipping directly from one state of the interactive to another. They allow the user to *see how the scene changes* as time goes by. Since time is involved, it is necessary to indicate how fast time is to move in the animation. And, evidently, what is to change in each step of the animation.

Animations can by cyclic (when they reach the end of the animation, they start from the beginning again). They can also be set to start immediately when the scene is launched, or respond to the user's interactions.

Animations share the algorithmic functionality present in the *INICIO*, and *CALCULOS* algorithms; and available also in functions. This means they have the *init*, *do* and *while* parameters as well.

Figure 12.0.0.1 shows the contents of the *Animation* tab.

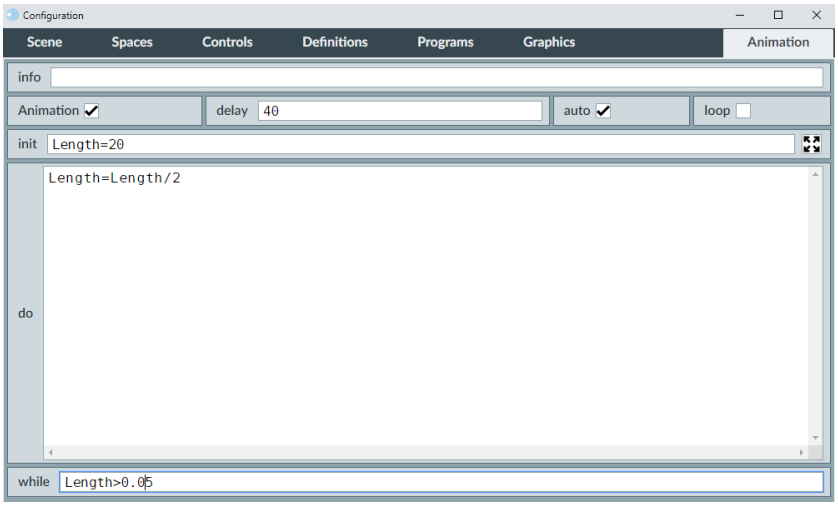


Figure 12.0.0.1: The *Animation* tab.

- **Animation**: When this checkbox is marked, all the parameters in this tab are activated, and an animation can be configured. If a scene does not involve an animation, this checkbox should be cleared.
- **pause**: A text field where a value or an expression is introduced. It is read as the number of milliseconds comprising the pause between succesive loops in the animation. Animations involving very many or numerically complex instructions may

exceed the pause time indicated in this parameter. However, if the instructions are straightforward, this time interval is likely to be respected.

- **auto**: When this checkbox is marked, the animation will start immediately when the scene is loaded. The user needs not manually activate it.
- **loop**: If this checkbox is marked, the animation will loop indefinitely, regardless of any condition entered in the *while* parameter. Checking this box is equivalent to entering a 1 in the *while* parameter (remember 1 is true, and will therefore implement an indefinite looping of the animation).
- **init**, **do** and **while**: These text fields are the generic ones found in algorithmic functions and in the *INICIO* and *CALCULOS* algorithms, and can be reviewed in the INICIO topic.

When an animation is running and the gear button is clicked to launch the scene's configuration editor, the animation pauses. This avoids unnecessary processing when the attention is not focused on the scene itself.

We now build, as an exercise, a scene that simulates the seconds and minutes hands of a clock. This exercise's interactive scene, along with the instructions to build it, can be found at Animation 1. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise involved a simple animation, where only the value of one counter is relevant. Animations can be potentially much more complicated and even involve complex functions in each loop. Note here that the 1000 ms pause in this example corresponds almost exactly to a second between ticks in the real world. This is only possible because the instructions done in each loop are done almost instantaneously (the processor does them almost immediately). However, when using very complex instructions, or a great number of them, the processor may take some real time to do them, and the time between ticks in real time may exceed the one indicated in the *pause* parameter.

Sine and cosine trigonometric functions were used in this exercise. Note the angles are handled in radians. The functions involved can be reviewed in the functions common to various programming languages topic. A boolean condition was also entered in the *while* parameter of the animation. These can be reviewed in the boolean conditions and operators topic. Finally, the colors of the hands were changed. More information on this functionality is available in the color editor topic.

Let us do a second exercise involving more complex animations. This exercise's interactive scene, along with the instructions to build it, can be found at Animation 2. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

So as not to start from zero, this exercise is done by editing the scene resulting from the matrices exercise (Definitions Matrix). This exercise used a matrix to store the particles' positions. It might be a good idea to review that exercise first before attempting the animation related one.

This exercise builds on the previous one in that the particles are now animated. The can be said to follow a sort of brownian motion (they move as specks of dust in the air).

This exercise uses an animation that does something more complicated in each step. It calls a function which changes the positions of the particles by randomly changing values in their defining matrix. The function is run, changing the positions of all particles, and then the animation is refreshed. The end result is that the user sees all particles change position simultaneously.

In a part of this exercise, both the animation variable and the function index variable were the same: *ianim*. Any conflict that could result from having both change the value of the variable could be avoided by entering *ianim* in the function's *local* parameter. This results in the program treating the function one as a protected, different variable, and so changes in the function would not impact the *ianim* variable of the animation. However, we decided against this, and in favor of changing the name of the function's index variable to *i*, since it is better practice to have different names for variables dealing with different aspects of the interactive scene.

We used boolean conditions in the *while* parameter of the function and animation, and in the assignment of new values for the particles' positions. Remember the boolean conditions and operators topic can be reviewed should questions arise.

13

DescartesJS intrinsic functionality

DescartesJS has many functions and variables already built in. This saves the programmer the trouble of programming the most basic ones.

Since there are many such variables and functions, it may be difficult to recall them all. This section can therefore be thought of as a rapid access to all this *DescartesJS* intrinsic functionality.

As in previous sections, some exercises are included that involve various of the most important functions and variables.

13.1 *DescartesJS* intrinsic variables

DescartesJS has many variables related to the properties of spaces, to the state of the mouse, to the state of graphic controls, etc. Many of these variables have a twofold functionality: on the one hand, their values provide information about the interactive scene (for example, the scale of a space); and on the other hand, the user can modify their values, thus modifying the properties of the interactive scene (for example, the scale of a space once again). We now list these variables depending on the type of action they do, or the functionality group to which they belong.

13.1.1 Space variables

Space variables store the width, height, offset and scale of a space. They also allow the user to control these features.

For a given space, its identifier, followed by a period, is the prefix of its space related variables (<*space id*>.) For example, the scale of a space with an *E1* identifier is related to the *E1.escala* variable.

- **<space id>._w**: This variable is related to the width of the space in px (hence the *_w* suffix).
  For example, we can print the value of the *Sp._w* variable to know the width of a space. It can be used to control the width in spaces with their resizable checkbox marked.

- **<space id>._h**: Its functionality is the same as the *._w* variable, but it is related to the height of the space.

- **<space id>.Ox**: This variable is related to the horizontal (hence the *x*) offset (hence the *O*). A plane's origin is centered in the space by default. This offset represents the number of px that the origin of the space is shifted to the right from the center of the space. Leftward shifts are represented by negative values.
  This value can be used to inform the value and can be edited to set the space's horizontal offset. It is associated to the *O.x* parameter present in 2D spaces.

- **<space id>.Oy**: The same as the *.Ox* suffixed variable, but related to the vertical offset. Negative values are associated with upwards shifts and positive ones to downward shifts.
  Assigning it a value will change the value of the vertical offset. It is associated to the *O.y* parameter present in 2D spaces.

- **<space id>.escala**: *Escala* is scale in Spanish. So, this variable is related to the scale of the space: the number of px comprising a unit length in the cartesian plane.
  The variable can be printed to know the scale value, or a value assigned to it to change the scale of the space in question.

- **<space id>.rot.y**: This variable is only valid in 3D spaces. It stores the rotation (in degrees) of the space around the *y* axis.
  The default perspective from which 3D spaces is described as follows: The *xy* plane is viewed perpendicularly from the *x* positive axis; while the *y* axis points to the right and the *z* one upwards. In this default configuration, the *.rot.y* variable has a zero value.
  The *.rot.y* rotation can be better understood in the following way: consider the *xz* plane is viewed from the positive *t* axis. The rotation is, then, the number of degrees of a counterclockwise turn of this plane around the *y* axis. Negative values of the rotation are related to clockwise turns.
  Besides using it to know the rotation value, this variable can be assigned one to change the perspective from which the space is viewed.

- **<Nombre del espacio>.rot.z**: The same as the *.rot.y* variable, but related to the rotation around the *z* axis.

By design, only the two aforementioned rotations are available (no *.rot.x* one is supported).

Many of the user manual exercises up to this point have used some of these variable, so no particular exercise is included here.

13.1.2 Mouse variables

Mouse variables are used to identify its state. For example, if its left button has been clicked, or is being pressed down. Its coordinates relative to the cartesian plane can also be known using mouse variables.

These variables start, as usual, with the identifier of the space on which the mouse state is checked followed by a period (<*space id*>.)

- **<space id>.mouse_x**: This variable is related to the horizontal coordinate, relative to the plane, where the mouse is clicked or dragged. The value is only refreshed if the mouse left button is active. If the mouse is just hovered around the value does not change.
  This variable is only informative. Its value can be printed, but no value assignment can be made to it.
  For example, *Sp.mouse_x* holds the horizontal coordinate of the mouse click when used on a space with a *Sp* identifier.

- **<space id>.mouse_y**: The same as the *.mouse_x* suffixed variable, but for the vertical coordinate relative to the plane.

  NOTE: It is possible for these variable to refresh their values even when the mouse is only hovered around. However, in order for this to work, the sensitive to mouse movements checkbox of the space (having the space in question selected in the *Spaces* tab) has to be marked. However, as already mentioned, this involves a lot of inner calculations that may impact on the scene's performance. So, it is recommended not to use this functionality save in cases in which it is absolutely necessary.

- **<space id>.mouse_clicked**: This binary variable has a 1 value when the mouse has been clicked at least once on the associated space, but is not being currently clicked. It has a zero value if the mouse has never been clicked on the space, and when it is continuously pressed down, rather than clicked. Another way to understand this is that its value is one after releasing a mouse click.

- **<space id>.mouse_pressed**: Another binary variable with a 1 value when the left mouse button is being continuously pressed down, and 0 otherwise (its value returns to zero after releasing a mouse click.

We now do an exercise that may help clear doubts regarding the mouse variables functionality. Additionally, this exercise shows a way to know other states of the mouse not included in the *DescartesJS* intrinsic variables. This exercise's interactive scene, along with the instructions to build it, can be found at Mouse. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

We see in this exercise that new states of the mouse can be known using states of the intrinsic mouse variables. This comes in handy since it is sometimes necessary to know, for instance, when the mouse is being dragged (some scenes require launching events when the mouse is being dragged or after release). Note that the *CALCULOS* algorithm is used since we need to know the values of the variables constantly while the user is interacting with the scene (by manipulating the mouse). In the exercise, precise instructions were

entered to calculate the new states of the mouse. We now analyze just how the variables change values in the scene, so as to remove the mystery of why those assignments were made.

- The *MPA* auxiliary variable starts with a 0 value. When we press the mouse, the *CALCULOS* algorithm is executed. Note that *MP=1* so, naturally, !*MP=0*. *MPA=0* so, naturally, !*MPA=1*. So,
  MDown=1&1=1,
  MDragged=1&0=0, and
  MReleased=1&(!1)=1&0=0.
  Of *MDown*, *MDragged* and *MReleased*, the only variable with a 1 value is *MDown*, as expected.
  The end value of *MPA* is 1, since at the end it is assigned the value of *MP*.
- The *MPA* auxiliary variable starts now a value of 1 (see the previous bullet point). Therefore, !*MPA=0*. When the mouse is dragged while pressed, the *CALCULOS* algorithm is executed again. The *MP* variable has a value of 1, since the mouse is being pressed. So, therefore, !*MP=0* again. *MPA* is still 1 and, consequently, !*MPA=0*. Using these values,
  MDown=1&(!1)=1&0=0,
  MDragged=1&1=1, and
  MReleased=1&(!1)=1&0=0.
  Of *MDown*, *MDragged* and *MReleased*, the only variable with a 1 value is *MDragged*, as expected.
  The end value of *MPA* is 1, since *MP* is still 1 (the mouse is being pressed).
- The *MPA* auxiliary variable starts with a value of 1 (see the previous bullet point). When the mouse button is released, the *CALCULOS* algorithm is executed again. Now, *MP* has a value of 0 (the mouse button is no longer pressed). So, !*MP*=1. *MPA* still carries its 1 value, so !*MPA*=0. Using these values,
  MDown=0&(!1)=0&0=0
  MDragged=0&1=0
  MReleased=1&(!0)=1&1=1
  Of *MDown*, *MDragged* and *MReleased*, the only variable with a 1 value is *MReleased*, as expected.
  At the end of the algorithm, *MPA* is assigned the value of *MP*. So, both *MP* and *MPA* end up with a zero value. The initial configuration is thus recovered.

We see that the *MPA* auxiliary variable plays a fundamental role in determining the new states of the mouse. Using these simple instructions, it is possible to ascertain whether the mouse has just been clicked, if it is being dragged, or if it has been released.

This exercise involves quite a few conditions. For more information regarding this functionality, visit the boolean operators and conditions topic.

By the way, even though we refer to *mouse* manipulations throughout this topic, the mouse variables functionality extends to mobile devices, in which the pressing is done by pressing the finger down on a space, releasing is done by lifting the finger, and so on.

13.1.3 Text field control variables

There is only on text field control variable. It is built using the control's identifier and appending *.active* (*.activo* is also supported). As an example, the variable associated to a text field control with a *t1* identifier would be *t1.active* (or *t1.activo*). This variable has a 1 value when the field is being interacted with, and 0 otherwise. This variable is read-only, meaning the user can only change its value by interacting with the control, and not by directly assigning it a value.

13.1.4 Graphic control variables

Graphic control variables are used to know the coordinates of a graphic control relative to the cartesian plane. Values can also be assigned to them in order to place a graphic control in a particular point in the space. They can also be used to know if a certain graphic control is active or not.

The identifier of the graphic control, followed by a period, is used as a prefix in these variables (<*gc id*>.) For instance, a *g1* graphic control has a *g1.x* variable related to the controls horizontal coordinate.

- **<gc id>.x**: This variable can be printed to know the value of the horizontal coordinate of the graphic control relative to the cartesian plane. It can be assigned a value, and the control will place itself in that horizontal coordinate. For a control with a *grph* identifier, the variable would be *grph.x*.

- **<gc id>.y**: The same as the *.x* suffixed variable, but this one is related to the vertical coordinate.

- **<gc id>.active** (**<gc id>.activo** is also supported): This variable has the state of activity of a graphic control. If the control is active, the variable's value is 1, and otherwise 0. A graphic control is considered active when it is being dragged, or if it is still selected because it was the last object with which the user interacted. It will remain in its active state as long as it is selected, and it is necessary to click elsewhere in the space to revert the control to its inactive state.

It is time for an exercise to practice the use of graphic control variables. This exercise's interactive scene, along with the instructions to build it, can be found at Graphic control. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise shows us how to use the graphic control variables to determine the coordinates of a certain point in the cartesian plane, as well as to place graphic controls at will

at a given location. Knowing the coordinates of a point is sometimes useful when placing objects such as text graphic objects. Other situations sometimes require a certain action be taken when a graphic control is manipulated, or when it crosses a certain part of the space. And yet another situation that often happens is that a graphic control is accidentally placed outside the area of a space. This happens, for instance, when the space is not fixed and the user drags it, along with the graphic control, until the latter lies outside the space. The control then becomes inaccessible. A workaround is to associate an action of a certain control to the re-positioning of the graphic control in a spot accessible to the user.

For more information on the type of controls used in this exercise, review the graphic controls topic.

13.1.5 Audio and video control variables

There is one variable and one function related to audio and video controls. Both use the control's identifier, followed by a period, as a prefix (<*control id*>.)

The variable is <*control id*>.*currentTime*, and it stores the time in seconds in which the control's playback is currently in. In order for it to update correctly, an animation is required to be present. There is a <*control id*>.*currentTime(t)* function with a different functionality. This one has function parentheses and an argument, and it can be reviewed in the audio and video control functions topic.

Feel free to review the Audio control exercise, where this functionality is put to practice.

13.1.6 Array and matrix variables

There is one basic variable related to arrays which stores the size of the array. And there are a couple of variables related to matrices, one which stores the number of columns, and one which stores the number of rows of a given matrix.

Array variable:

- **<array id>.long**: This variable stores the size of the array (as in the *size* parameter of the array definition). It is basically the number of entries of the array. For instance, for an array with a *V1* identifier which has 5 entries, the value of the *V1.long* variable is 5.

Matrix variables:

- **<matrix id>.filas**: This variable stores the number of rows of a given matrix. *filas* stands for *rows* in Spanish.
- **<matrix id>.columnas**: This variable stores the number of columns of a given matrix. *columnas* stands for *columns* in Spanish.

  As an example consider a matrix with an *M3* identifier, 4 columns and 5 rows. In this example, the *M3.columnas* variable has a value of 4, while the *M3.filas* variable has a value of 5.

13.1.7 Path variables

There is a means for the user viewing an interactive scene **in a server** to provide a parameter for the *DescartesJS* scene straight from the path used to open the scene on a browser. The name of this variable is *URL.<tag>*, where *tag* stands for the name of the tag provided in the path, along with its associated value (which is a character string). This may be hard to grasp from the description alone, so we include an example.

Consider a *DescartesJS* interactive scene, stored, **on server**, in the following path: *https://sitio.abcd.mx/index.html*. If the user types this path on the browser, the browser will open the scene as expected. However, an additional tag may be provided in the path, along with its desired value. Consider now the following path:
https://sitio.abcd.mx/index.html?view=horizontal. If this path is entered in the browser, the scene is opened in the browser. But, additionally, a *URL.view* variable internal to the scene is assigned the *horizontal* character string (that is, 'horizontal').

To enter a path variable, a question mark (?) is included at the end of the path to the scene, followed by the name of the tag, followed by an equal (=) symbol, followed by the character string that will be assigned as its value. It is important to remember that the inner variable in the scene has a *URL.* prefix. So, if the tag was *wow*, the inner variable would be *ULR.wow*. If no value is defined for a tag (i. e., if nothing is entered after its = symbol), the *URL.<tag>* variable will be assigned a 0 numeric value.

Note that variables set in paths are not defined **inside a scene**. It is the user who "creates" them by appending their information at the end of the scene's server path.

Multiple path variables may be defined in a path; it is only necessary to separate each from the next with an ampersand (&) character. Consider the following path:
https://sitio.abcd.mx/index.html?view=horizontal&step=7
When this scene is loaded from the server in a browser, the following variables are created internally: *URL.view* (assigned a value of 'horizontal'), and *URL.step* (assigned a value of '7').

These path variables allow the user to prepare the scene to open under specific settings. Digital interactive book editors created in *DescartesJS* use this functionality. The same editor may open a myriad of books present in the server. However, if a specific book is to be opened, its name could be indicated via a path variable. The *view* path variable example has also been used in order for the user to indicate whether the page layout of a digital interactive book is to be *horizontal* or *vertical*.

Path variables only make sense for scene's stored in a server. If the scene is opened locally, they are not taken into account.

13.1.8 *DescartesJS* general variables

There is only one *DescartesJS* general variable: *rnd*. This variable, whose name stands for *random*, generates a random value in the $[0, 1)$ interval (i. e., it can potentially be zero, and

its largest possible value is almost 1, but 1 itself not being included). A different random number is generated when a new reference to *rnd* is entered in the code. The probability distribution in this interval is uniform (one number has basically the same chances of being chosen as any other in the interval).

The *rnd* variable is frequently associated with functions such as *ent*(). Some situations require random numbers to be generated in a different interval from the [0,1) one. Others may require generating random numbers from a discrete value set. In order to satisfy such conditions, *rnd* may need to be operated using basically products, so as to shift, and "stretch" (or "compress") the interval. And the *ent*() function is used to "discretize" the available values.

All this may be better understood by doing an exercise. The purpose of this exercise is to generate vertical parabolas using the $y = ax^2 + bx + c$ equation. The a coefficient must adopt random integer values between -3 and 3, but excluding zero ($a = 0$ corresponds to a line, not a parabola). The b coefficient must adopt random semi-integer values between -4 and 4 (i. e., -4, -3.5, -3, ..., 3.5, 4), including 0. And the c coefficient may adopt real values between 0 and 3. This exercise's interactive scene, along with the instructions to build it, can be found at General Variables. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This is a complicated exercise, but allows us to understand many aspects. Each variable involved has a different condition regarding the values it can be assigned.

- The value assigned to the a variable is configured as follows:
  First, the sign of the variable is set to be positive if the *rnd* variable is lower or equal to 0.5 (0.5-*rnd* is positive in such a case); and negative otherwise (0.5-*rnd* is negative when *rnd*>0.5). This ensures that there is a 50% chance the argument of the sign function is positive. The sign function then only takes the sign of the argument, so it will have a 50/50 chance of being positive or negative. This is then applied to the expression which generates a 1, 2 or 3 value. All this results in the generation of positive or negative integer values between 1 and 3. It also excludes the possibility of getting a 0 value.

- The b variable has no non-zero restriction. This simplifies the expression assigned to it. It is simply the lowest value possible plus the an 8 value (resulting in a maximum +4 value) broken in 16 parts, so that semi-integers are used.

- The c variable does not have an integer or semi-integer restriction. So, there is no need to use the *ent*() function. It is only necessary to scale the *rnd* interval from [0,1) to the [0,3) by multiplying times 3.

- The expressions assigned are such that the available values each have the same probability of being chosen. Take a as an example. The `3*rnd` in the expression generates a random number between 0 and 2.9999... When floored using the *ent*() function, only the 0, 1 and 2 integers are extracted, each with almost the same probability. By adding 1, they are shifted, so the integers become 1, 2 and 3, as desired.

- Rich text formatting, as opposed to plain text, allows a single text graphic object to print different values using different settings. In the present exercise, the difference was the number of decimals involved.

Yet another important conclusion is extracted from this exercise: the use of debug texts. The text printed at the end of the exercise allows the programmer to see whether the program is behaving according to specifications. Maybe this text is not to be showed to the end user, but the programmer can confirm the variables' values are within specifications by printing them. Once confirmed, the text can be removed or simply hidden from view.

The *ent*() and *sgn*() functions were used in this exercise. These functions are already defined in *DescartesJS*, and their functionality can be reviewed in the *DescartesJS* intrinsic functions topic. The graph included corresponds to a type of graphic object, which can be reviewed in the equation graphic topic.

13.1.9 Numerical constants

There are two numerical constants pre-defined in *DescartesJS*. One is *pi*, a numerical approximation of π; and the other is *e*, a numerical approximation of Euler's number (Napier's constant). The user may print their values or use them in expressions.

It is possible to redefine their values by assigning them a new one. However, in order to avoid confusion, we recommend not changing their values.

13.1.10 Information and customization variables

- **device**: a variable which, when printed, indicates via a text string the type of device in which the Descartes scene is used. The options are *desktop* (a computer), *tablet* and *mobile* (a mobile phone).

- **dispositivo**: a variable equivalent to the *device* one mentioned above (*dispositivo* stands for *device* in Spanish).

- **_NUM_MAX_ITE_ALG_**: a variable which contains the maximum number of iterations Descartes allows in an algorithm with repetition. It was originally set to 100,000 iterations (the variable's default value). The purpose of this number is to stop an endless loop which would render a scene inoperable, usually following an error in the user's code. However, since it may be desirable to exceed this limit, particularly when using Descartes to perform simulations, the user may reassign a new, larger number to this variable, thus allowing for more cycles in an algorithm.

The *device* and *dispositivo* variables are useful, since they allow a control with the virtual keyboard enabled to be displayed depending on whether the scene is opened in a

computer, tablet, or phone. A control, which has the virtual keyboard disabled, can be displayed in a scene if it is opened in a computer. Another control, with the virtual keyboard enabled, can be displayed in its stead if the scene is opened in a tablet or phone. Remember that the device's native keyboard is displayed by default in tablets and phones, and this keyboard usually takes up a lot of space, thus reducing the size of the scene and breaking its aesthetics. However, using the aforementioned approach, the virtual keyboard can be used instead in a position decided by the programmer, thus in keeping with the aesthetics and functionality of the scene.

13.2 *DescartesJS* intrinsic functions

DescartesJS has several intrinsic functions, most of which are available in almost any programming language. However, there are some particular to *DescartesJS* related to audio, video, expressions' evaluations, etc. which will be reviewed in short. The functions here reviewed are categorized according to their functionality.

13.2.1 Common functions

There are several functions in *DescartesJS* common to various programming languages. These are listed alphabetically. Most of them are mathematical functions. Some functions' names may vary, but their functionality is preserved.

- **abs(x)**: This function receives a numeric argument *x* and returns the absolute value of the same. Example: *abs*(-2) will return a 2 value.
- **acos(x)**: This function receives a numeric argument *x* and returns the arc cosine function (the angle, in radians, whose cosine is *x*). For example, *acos(-1)* will return a *3.141592...* value (which corresponds to 180°).
- **_AnchoDeCadena_(string,font,style,size)**: This function receives four arguments and returns a positive integer, which corresponds to the length (in pixels) of a string of characters printed using the formatting indicated by each argument. *ancho de cadena* is *string length* in Spanish. *string* is a character string, *font* stands for the font used (the available options are 'Monospaced', 'Serif' and 'SansSerif'), *style* stands for the the style attributes (the available options are 'PLAIN', 'ITALIC', 'BOLD', 'ITALIC+BOLD' and 'BOLD+ITALIC'), and *size* stands for the size in points of the chosen font. The function returns the length in px that the string will have using the formatting.
  For example, *_AnchoDeCadena_('hi, how are you?','Monospaced','ITALIC+BOLD',18)* calculates the length (in px) that the *hi, how are you?* character string will have when printed in a *monospaced* font, using italics and bold style, and in an 18 point font. This function comes in handy, for example, when the programmer needs to know if a string is long enough so as to not be able to fit in a given space.

- **asin(x)**: This function receives a numeric argument *x* and returns the arc sine function (the angle, in radians, whose sine is *x*). For example, *asin(1)* will return a *1.570...* value (which corresponds to 90°).
- **atan(x)**: This function receives a numeric argument *x* and returns the arc tangent (the angle, in radians, whose tangent is *x*). For example, *atan(1)* will return an *0.785...* value (which corresponds to 45°).
- **ceil(x)**: This ceiling function receives a numeric argument *x* and returns the immediately upper integer to *x*. As examples, *ceil(-4.2)* will return a -4 value, and *ceil(5.2)* will return 6.
- **charAt(string,n)**: A function that receives a character string as its first argument (*string* in the example), and a positive integer as the second (*n* in the example). The function returns the character found in the string at the *n*-th position. Counting starts at zero. For example, *charAt('Hey there!',3)* will return *t*. This function is equivalent to its Spanish counterpart, *_letraEn_(string,n)*.
- **cos(x)**: This function receives a numeric argument *x*, interprets it as radians, and returns its cosine. For example, *cos(pi/2)* will return 0. We remember that $\frac{\pi}{2}$ corresponds to 90°.
- **cot(x)**: This function receives a numeric argument *x*, interprets it in radians, and returns the cotangent, which is the reciprocal of the tangent of that number. The reciprocal of a number is equivalent to 1 divided by the number. For example, *cot(0.785398163)* will return a value near 1, since *0.785398163* radians is near 45°.
- **csc(x)**: This function receives a numeric argument *x*, interprets it in radians, and returns the cosecant, which is the reciprocal of the sine of that number. The reciprocal of a number is equivalent to 1 divided by the number. For example, *csc(pi/2)* will return a value near 1.
- **ent(x)**: From the Spanish word *entero*, which stands for *integer*. This function receives a numeric argument *x* and returns the integer immediately below this number. For example, *ent(-3.2)* will return -4.
  Adding a 0.5 value to the argument of this function turns it into a *rounding* function. For example, *ent(3.78+0.5)* will return 4, which is the rounded value of 3.78.
  Additionally, note that, when the function's argument is positive, the function simply *trims* the decimals from the number.
- **exp(x)**: This function receives a numeric argument *x* and returns the Neperian exponential. For example, *exp(2)* will return 7.389... (e^2 = 7.389...).
- **floor(x)**: This function receives a numeric argument *x* and returns the integer immediately below it. For example, *floor(-4.2)* will return a -5 value and *floor(5.2)* will return a 5 value.
  In this sense, this function does the opposite from *ceil()*, and is equivalent to the *ent()* function.

- **indexOf(string,str)**: This function receives two arguments. The first (*string* in the example) is a character string of the main text. The second (*str* in the example) is a character string of a text potentially present in *string*. The function returns an integer: the index of the first found instance of the second text inside the main text. Counting starts at zero (the first character is indexed as the zeroth). If no instances are found, a -1 value is returned. For example, *indexOf('Hi there!','he')* would return 4.
- **_índiceDe_(string,str)**: This is the Spanish counterpart of the *indexOf(string,str)* function. It is one of the few functions that involve an accent (the first *i* in *índiceDe*).
- **lastIndexOf(string,str)**: This function is similar to the *indexOf(string,str)* one. It also receives two character strings as arguments. The first one (*string* in the example) is the main string, while the second (*str* in the example) is the character string of the text that is potentially present in the main string.
  This particular function returns the index (counting from zero) for which the second string appears inside the main string **for the last time**. As an example, consider *lastIndexOf('This is a list','is')* would return an 11 value (the *is* inside *list* is the last appearing instance of *is* in *This is a list*, and it appears in the 11th place if you start counting from zero).
- **_letraEn_(string,n)**: This function receives two arguments. The first (*string* in the example) is a character string. The second (*n* in the example) is a positive integer, including 0. *n* is the index of the character (the count starting from zero) in the string that the function returns. For example, *_letraEn_('Salutations',3)* will return the *u* character.
  This function is basically the Spanish counterpart of the *charAt(string,n)* one.
- **_longitud_(string)**: *longitud* is Spanish for *length*. This function is basically the Spanish counterpart for the *strLength(string)* function.
- **log(x)**: This function receives an *x* numeric value as an argument and returns the Neperian logarithm of the same. For example, *log(7.389056099)* will return a value close to 2, since $e^2 = 7.389056099...$.
- **log10(x)**: This function receives an *x* numeric value as argument and returns its base 10 logarithm. For example, *log10(1000)* will return a value of 3 since $10^3 = 1000$.
- **max(a,b)**: This function receives two numeric values as arguments (*a* and *b* in the example) and returns the highest one. For example, *max(-6,-3.59)* will return -3.59.
- **min(a,b)**: This function receives two numeric values as arguments (*a* and *b* in the example) and returns the lowest one. For example, *min(-7,-4.2)* will return -7.
- **raíz(x)**: *raíz* is Spanish for *root*. This function is the Spanish counterpart for the *sqrt(x)* one.
- **replace(string,str1,str2)**: This function receives 3 character strings as arguments. The first one (*string* in the example) is the main string. The second (*str1* in the example) is a target string inside the main string. The third one (*str2* in the example) is

the string that will substitute the target string. The function returns the main string, but having replaced all instances of the second string with the third string. For example, *replace('123123','2','two')* will return a *1two31two3* character string.

- **round(x)**: This function receives an *x* numeric value and returns the rounding integer of the number. For example, *round(5.8)* will return 6, while *round(5.2)* will return 5.
  The rounding of an *x* number can be understood as the nearest lower integer of *x+0.5*. So, *round(x)=ent(x+0.5)=floor(x+0.5)*.
- **sec(x)**: This function receives a numeric value *x*, interprets it as radians, and returns the secant, which is the reciprocal of the cosine of that number. For example, *sec(pi)* will return a -1 value.
- **sen(x)**: This function is the Spanish counterpart of the *sin(x)* function (*seno* is *sine* in Spanish).
- **sgn(x)**: This function receives an *x* numeric value and, if it is positive, a +1 value is returned. Otherwise, a -1 value is returned. This way, the function only extracts the sign of the number, disregarding its magnitude.
- **sin(x)**: This function receives a numeric argument *x*, interprets it as radians, and returns its sine value. For example, *sin(pi/2)* will return a value of 1 (bearing in mind that $\frac{\pi}{2}$ radians are equivalent to 90°).
- **sqr(x)**: *sqr* stands for *square*. This function receives a numeric value *x* as argument and returns its square. For example, *sqr(3)* will return a value of *9*.
- **sqrt(x)**: It stands for *square root*. This function receives an *x* numeric value as argument and returns its square root. For example, *sqrt(9)* will return a value of 3.
- **strLength(string)**: This function is used to obtain the *length* of a character *string*. It receives an character string (*string* in the example) as an argument, and returns the number of characters included in that string. For example, *strLength('Salutations')* will return 11.
  This function has a Spanish counterpart: *_longitud_(string)*.
- **substring(string,a,b)**: This function receives 3 arguments. The first (*string* in the example) is a character string. The second and third ones (*a* and *b* in the example) are integers which refer to the indexes that flank the substring to extract. As usual, counting starts at zero. The function returns a character string that is a substring of the *string* which goes from the *a*-th character (including such character) upt to the *b*-th one (excluding such character). For example, *substring('hiya',1,3)* will return a *iy* character string.
- **_subcadena_(string,a,b)**: This is the Spanish counterpart for the *substring(string,a,b)* function.
- **tan(x)**: This function receives a numeric value *x*, interprets it as radians, and returns its tangent. For example, *tan(pi/4)* will return a 1 value (bearing in mind that $\frac{\pi}{4}$ radians are 45°).

- **toFixed(x,n)**: This function receives an *x* real value as first argument, and an *n* integer as a second. It returns, in the form of a text string, the real adjusted value of *x* using *n* decimals. For example, consider a graphic text object set to print 7 fixed decimals. If `[toFixed(1/3,3)]` is entered in its *text* parameter, it will print 0.333 instead of the 0.3333333 it would print if only `[1/3]` were entered.
  Note also the difference between what `[toFixed(1/3,3)]` returns and what is returned by `[toFixed(1/3,3)*1]`. The first prints only 0.333, since the *toFixed* function has precedence over the 7 fixed decimals set for the text graphic. The second prints 0.3330000 since when it performs the product, its result is considered a new number, no longer fixed by its *toFixed* related function factor.
  When multiplying numbers returned by the *toFixed()* function, *DescartesJS* interprets them as numbers and operates them as such. However, when attempting to add them, the result is the concatenation of both strings (since *toFixed()* returns the numbers as text strings). For example, `[toFixed(1/3,3)+toFixed(1/7,2)]` will print the concatenation of both values returned by each *toFixed()* function. That is, the concatenation of the *'0.333'* string and the *'0.14'* one. The value printed is the *'0.3330.14'* text string. The *_Num_() DescartesJS* language function can be used in order to add them as numbers. Each value returned by the *toFixed()* function is then used as the argument of the *_Num_()* function, which turns the strings to numbers, and then the values are added. For example, `[_Num_(toFixed(1/3,3))+ _Num_(toFixed(1/7,2))]` prints the *0.473* numeric value.
- **toLowerCase(string)**: This function receives a character string (*string* in the example) and returns the same string, but replacing all upper case characters with their lower case counterparts. For example, *toLowerCase('Now Everything Is Lower Case')* will return a *now everything is lower case* character string.
- **toUpperCase(string)**: This function is similar to the *toLowerCase(string)* one, but it converts all characters in the string to their upper case. For example, *toUpperCase('Now everything is upper case')* will return a *NOW EVERYTHING IS UPPER CASE* character string.
- **trim(string)**: This function receives a character string as argument (*string* in the example) and returns the same string after having trimmed it of any leading, and trailing, blank or space character. For example, *trim('.'+' Hi '+'.')* will return a . *Hi* . character string, while *'.'+trim(' Hi ')+'.'* will return a *.Hi.* character string. Note that the first example finds no trailing blank spaces in the argument of the *trim* function since the periods are the first and last characters. The second example does trim blank spaces and then concatenates the trimmed text to the preceding and trailing periods.

There are also some hyperbolic trigonometric functions (sine, cosine and tangent). They are the same as the common functions, but an *h* has to be appended at the end of the function (*sinh()*, *cosh()*, and *tanh()*).

It may be necessary sometimes to obtain logarithms in bases different from *e* and

10. The $log_a(x) = \frac{log(x)}{log(a)}$ property (or, alternatively, the $log_a(x) = \frac{log_{10}(x)}{log_{10}(a)}$ property) can be used to define a function which returns the base a logarithm of a number. For example, a *logbase(a,x)* function can be added in the *Definitions* tab with a return value of `log10(x)/log10(a)`. This function receives the value of the base (a) and the number for which the logarithm is to be extracted (x); and it returns the value of $log_a(x)$.

13.2.2 *DescartesJS* language functions

There are a few functions unique to *DescartesJS*. One of them is ***_Eval_(string)***. This function receives a character string argument (*string* in the example), and evaluates it as a mathematical expression. This has the potential to even let the user write down a function to be evaluated and then graphed.

As a first example, consider a *text* graphic object which has `[_Eval_('sin(pi/6')]` entered in its *text* parameter. If the changes are applied, the printed text should be 0.5. In this example, the character string was the expression *sin(pi/6)* to evaluate. This shows how *DescartesJS* uses this function to interpret a string as a math expression. If there is an assignment such as `c1='sin(x)'` elsewhere in the scene, *_Eval_(c1)* will be associated to the sine function applied to the x variable. Thanks to this, *DescartesJS* is not constrained to graphing equations provided solely by the programmer, but makes it possible for the end user to enter functions of his/her own.

An exercise is in order, and it will likely clear some of the confusing aspects of this function. On the one hand, this exercise shows how the end user can graph a function of his/her own. On the other, the function can be evaluated at a specific value of its independent variable. This exercise's interactive scene, along with the instructions to build it, can be found at Descartes Functions. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

There are several things worth pointing out in this exercise:

- Whenever the *_Eval_()* function is used, its argument has to be a character string. This means that, if its argument is a variable, this variable should contain a character string. If it is a text field numeric control, it should preferably be text-only. And its initial value should preferably be flanked by single quotes. For example, if a numeric control's value is 1 instead of '1', errors may appear when trying it as an argument of the *_Eval_()* function.

- The code is clearer and easier to manipulate when text field evaluations are appointed to functions.

- There is a difference between the evaluation of a function along an entire domain and only at a given value. The equation graphic object draws the *fn()* function (which has no arguments), while the *fn2(x)* function (which has an argument) is evaluated in only one point. Consider a case in which a student is asked to enter a formula as an answer to a question. There may be subtle differences between the same formula

(for instance, *x*x* is the same as *x^2*). A way to know whether the one provided by the student is correct or not is to evaluate it at different points and compare the value of each to that of the target formula's evaluation in the same points. If they match in all cases, the student's formula is most likely correct.

This exercise involves graphing some common functions. More information on these is provided in the functions common to various programming languages topic. The *text field* numeric control topic may also be of interest to the reader, since they were used twice in the exercise.

The *_Eval_()* function can also be used to assign values to variables. By this we do not mean assigning the function's return value to a variable, but rather to implement the assignment of a value to a variable as part of the evaluation. A special combination of symbols is used for such a purpose: `:=`. Left of this symbol is the variable to which the value is assigned, and to the right, the value. Boolean conditions can also be implemented inside the *_Eval_()* function. These use their same symbols as usual. For example, a button could be set to calculate the following series of instructions:

```
_Eval_('zzz:=zzz+1')
_Eval_('v1:=(zzz%2==0)?1:0')
_Eval_('v2:=(zzz%2!=0)?1:0')
_Eval_('v3:=(zzz<=3)?1:0')
```

We see that the first line assigns *zzz+1* to *zzz*, thus increasing its value by 1. The second line assigns a variable, *v1*, a 1 value if *zzz* is divisible by 2, and a 0 value otherwise. The third line assigns another variable, *v2*, a 1 value if *zzz* is not divisible by 2, and a 0 value otherwise. Line 4 assigns another variable, *v3*, a 1 value if *zzz* is lower than 4, and a 0 value otherwise. If all four variables' values are printed, their behavior can be seen. Note that assignments were all done inside an *_Eval_()* function.

What is so useful about operating elements inside this function? The answer is that the variables to which values are assigned can be given by the user, and they need not be defined by the programmer. Additionally, it has the potential of using a single control to change the value of a multitude of variables. To illustrate this, we do a brief exercise. This exercise's interactive scene, along with the instructions to build it, can be found at Evaluation assignment. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

ExecBlock(string,label) is a proper *DescartesJS* language function similar to *_Eval_()*. However, this one allows the evaluation of not only one command line, but of a whole block. The block is flanked by a *label* whose name is chosen by the programmer, and contains the assignments or instructions. The structure is very similar to HTML. For example, suppose the following text is entered as a character string in a *str* variable:

```
Text beginning
<LBL>
tx=‘miscellaneous text’
```

```
x=2
</LBL>
More code follows...
```

This text could have be read from a file, for instance. The line skips would then have to be entered by pressing *ENTER* in a UTF-8 (without BOM) text editor, or by including a `\n` code that is a line break in *DescartesJS* (in which case the whole of the text would be run-on). The command `_ExecBlock_(str,'LBL')` will then search for the text contained in a block flanked by a `LBL` label (i. e., between `<LBL>` and `</LBL>`), and implement the assignments found therein. If this example included the printing of the *tx* and *x* variables after executing this block, the *x* variable would have a value of 2 and the *tx* one would contain a character string (*miscellaneous text*).

An exercise is in order so as to clear as many doubts as possible. This exercise's interactive scene, along with the instructions to build it, can be found at Block execution. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

In this example, a block involving a couple of assignments is executed. Both assignments are executed upon calling the *ExecBlock()* function. And only the commands inside the block flanked by the *LBL* label are executed, the rest is ignored. Additionally, when an assignment contained in the block, such as the numeric assignment `x=2`, the variable adopts the 2 numeric value, not the 2 character. Should the need arise to interpret it as text, the `x='2'` would be in order (note the 2 is flanked by single quotes, indicating it is to be regarded as text).

Num(string) is another proper *DescartesJS* function which receives a character string as an argument (*string* in the example). This character string corresponds to a numeric decimal expression. The function returns the number as such (as a number, not a string anymore). If the string is anything different than a numeric decimal expression, the function returns *NaN* (*not a number*). To better understand this function's usefulness, consider a *t1* text field. A student is supposed to answer 1+2=? in that text field. The correct answer is 3. However, the student may still type `1+2` in that text field, and upon entering it, the text field numerically converts it to 3, resulting in a correct verdict. However, since the idea is for the student to enter 3, the text field's *text ony* checkbox can be marked. If the student enters `1+2`, it will no longer be numerically interpreted. Furthermore, *_Num_(t1)* will return *NaN* if `1+2` is entered in the field, since the 1+2 string cannot be numerically interpreted. However, if the student enters `3` directly, it is converted to a number, which can later be used to know if the answer was correct or not.

So, this function can be used to avoid the use of *DescartesJS* intrinsic calculator functionality present for text field controls. This, in turn, forces the student to enter the numerical value of the answer (in exercises that are supposed to work that way).

If the argument of the *_Num_()* function is a decimal, either comma or period can be used as a decimal symbol. Please note that, in order for the function to work as described

before when using a text field, the text field must have its *text only* checkbox marked.

isNumber(x) is a Descartes function which receives an argument (*x* in the example) and returns a 1 value if the argument is a real number, or 0 otherwise. This enables the use of conditionals which depend on whether the argument is an *NaN*, *Infinity* or a character string (in these cases, the function returns a 0 value), or whether it is a real number (in which case the function returns a 1 value). For example, *isNumber(sqrt(-1))*, *isNumber(log(-1))*, and *isNumber(1/0)* al return a 0 value, since the argument of the first two examples corresponds to *NaN*, and the latter corresponds to *Infinity*. Contrariwise, *isNumber(e^2)* and *isNumber(sqrt(25))* both return a 1 value, since both arguments correspond to real numbers.

The function also returns a 0 value when its argument is a character string. For example, *isNumber(123)* returns a 1 value, but *isNumber('123')* returns 0, since its argument is a character string. Nonetheless, *isNumber(_Num_('123'))* returns 1 again, since the *_Num_()* function converts the string of numbers to a numeric value.

13.2.3 *HTMLIFrame* space functions

These functions are related to *HTMLIFrame* embedded spaces, are proper to *DescartesJS* and are used to pass information between a main scene (a container) and a scene opened as a subordinate HTMLIFrame space in the container, when this scene is also a *DescartesJS* scene. For more information on this type of space, see the HTMLIFrame space topic.

- ***<identifier>.set('vr',var)***. This function requires the identifier of a space (main or subordinate) before the *.set* suffix. Its arguments are a character string and a variable (*'vr'* and *var* in the example). This function is used to send information to another scene. The first argument is the name of a variable, **in the receiving scene**, whose value is to be set by the scene sending the information. The second argument is the variable, **in the scene sending the information**, whose value is to be transferred to the receiving end. In this example, the receiving scene will have a *vr* variable and the value of the *var* variable in the sending scene is to be assigned to it in the receiving scene. The <*identifier*> is that of the scene receiving the information. There are a few more things to point out regarding the identifier. However, passing on the information using this function does not ensure the receiving scene is updated. This is done with the <*identifier*>*.update()* function explained shortly.
- ***<identifier>.exec('fnc',func)***. This function sends a call to a *fnc()* function that must be already present in the *Definitions* tab **of the receiving scene** (it sends an instruction to execute such function). The first argument is a character string (*'fnc'* in the example) matching the name of the function to execute on the receiving end. The second argument (*func* in the example) is optional: it is the variable to be passed along as an argument of the function on the receiving end. If the *fnc* function requires a single argument, the function can be set as in the example). Otherwise,

instead of a single second argument, a character string (flanked by single quotes ‘’) consisting in the various variables (separated by commas) to be passed as argument is included as the *<identifier>.exec(‘fnc’,func)* function’s second argument.
For example, consider a *fnc(arg1,arg2)* function on the receiving end which uses two arguments: *arg1* and *arg2*. Then, the `<identifier>.exec(‘fnc’,‘a,b’)` call in the scene sending the data would pass the values of the *a* and *b* variables in it as arguments to be used on the *fnc()* function on the receiving scene, and execute that function.
If the function does not require arguments, a blank character string is included as second argument in the *exec* function. For example, `<identifier>.exec(‘fnc’,‘’)`.

- ***<identifier>.update()***. As already mentioned, this function forces the scene receiving the data (the one with the *<identifier>*) to update or refresh itself. It is usually used after a variable value has been set, or a function has been remotely run, on the receiving end. I. e., it is usually used after *<identificador>.set* or *<identificador>.exec* functions. This update function requires no arguments.

When a subordinate scene (one included inside a main scene via an HTMLIFrame space) is to send information to the main scene, this scene may sometimes not know the identifier of the main scene. In these cases, a *parent* identifier suffices to indicate the information is to be transferred to the main scene. For example, `parent.set(‘vr’,var)`.

When a parent scene is to pass information to a subordinate scene, the identifier is the name of the subordinate scene’s HTMLIFrame space. This is the case here, since the parent scene does know the name of its subordinate space. For example, if the container scene has a subordinate scene via an HTMLIFrame space with a *daughter* identifier, the `daughter.set(‘vr’,var)` function would pass the value of the *var* variable (present in the container scene) on to the *vr* variable (present in the subordinate scene).

All the exercises in this user manual (those also included in the *DescartesJSDocumentation.zip* file) use this functionality. The main scene, or container, is the *index.html* file in each exercise’s folder (you may open it with *DescartesJS* to see how it works). It has two subordinate scenes: *<exercise name>_scene.html* and *<exercise name>_text.html*. Both of these files are opened in an HTMLIFrame space with a *MAIN* identifier. There is also code related to exchange of information between spaces in the *INICIO* algorithm of the subordinate scenes. For instance, they pass their name on to the container, so it can print it in its top right corner.

Even though any exercise provided with this documentation uses this functionality, we nonetheless include a particular exercise. Actually, this exercise is the only one that **is not displayed inside a container**, as opposed to the rest. This is deliberately done so as to keep the scenes as simple as possible. So, the exercise has just a pdf file with the instructions and a couple of scenes: *Dad.html* and *Sp2.html* (the main container scene and its subordinate one, respectively). There is no *index.html* file containing them both, as opposed to the rest of the exercises. The instructions can be read using this link. The following links can be used to access the main and subordinate scenes: Dad.html and

Sp2.html. All these files are also included in the *DescartesJSDocumentation.zip* file, inside their *HTMLIFrame* folder stored in the *Exercises* folder.

Let us review what we achieved by doing this exercise. A *set* function passed the value of a variable from a space to another, allowing communication between a container (*Dad.html*) and a contained scene (*Sp2.html*). The *exec* function remotely calls a function in another space, allowing also the passing along of an argument's value, should the function have one. The *update* function remotely refreshes the space to which it refers to. Additionally, a *parent* prefix in any of these functions indicates the target scene is the container one. One single container may have several contained subordinate scenes. To specify to which the information flows, the subordinate space's HTMLIFrame identifier is used as a prefix in the functions in the container scene. However, a contained scene can only have one parent scene. Hence the use of the default *parent* prefix when referring to the parent. As mentioned, the HTMLIFrame functionality is used in the *index.html* container in any other exercise, and its subordinate scenes. The reader may open these in *DescartesJS* and review how they work to even better understand this.

Besides the functions used to communicate information between spaces, there is a particular function whose purpose is to reload different html content in a same HTMLIFrame space.

- ***<identifier>.changeConf(path)***. This function reloads **solely the html code** of a subordinate scene. It is typically used when a same HTMLIFrame space is to handle several HTML contents. When this function is used, the *DescartesJS* interpreter is **not** loaded. This situation is desirable in situations where quick transitions between different html scenes are required. The content is loaded from an *html* file. The argument of the function is a character string with the path of the *html* file to be loaded, relative to the container's path.
  Imagine a main scene *Dad.html* has two subordinate scenes: *A.html* and *B.html*. Both are stored in the same folder as the Dad one. *Dad.html* has an HTMLIFrame space, with a *subscene* identifier, and with `A.html` entered in its *file* parameter in the space. This way, the default opened subordinate scene is *A.html*. If the contained scene is to be changed then to *B.html*, the `subscene.changeConf('B.html')` function would have to be called (remember the *B.html* file is saved in the same folder as its containing scene). When the function is called, the HTMLIFrame space is reloaded using the <ajs>...</ajs> block code from the *B.html* file (the block related to the *DescartesJS* code).

The *changeConf* function actively switches the content of an HTMLIFrame space using different scenes. This also means it is not indispensable to have one HTMLIFrame space for each scene to be used.

13.2.4 Audio and video control functions

Some of *DescartesJS* intrinsic functions are associated to audio a video controls. These functions have the control's identifier, followed by a period (.), as a prefix. For example, <*control identifier*>.*play*() is one used to control the file's playback. The functions are:

- **<control id>.play()**: When this function is called, the audio or video file related via the control id is reproduced from the beginning. For example, *a2.play()* would be the function for a control with an *a2* identifier.
- **<control id>.stop()**: This one stops the playback of the audio or video file related to the control. For an *a2* control, this function would be *a2.stop()*.
- **<control id>.pause()**: When this function is called, the related audio or video file is paused. If it is eventually played back, it will start at the time where it was paused. For an *a2* control, this function would be *a2.pause()*.
- **<control id>.currentTime(t)**: This one does have an argument. It is a number (*t* in the example) which is interpreted as the time (in seconds) where the playhead is to be placed in the related audio or video file. If the file is subsequently played, it will start at that time. For an *a2* control, *a2.currentTime(5)* will place the playhead of the audio or video file of the *a2* control at 5 seconds after the beginning. An *a2.play()* function called afterwards will play the file back starting at 5 seconds from the beginning.
  Note that, though similar, this is a function and not related to the variable used to print the playhead's time: <*control id*>.*currentTime*, which has neither function parentheses nor arguments. For more information on this variable, review the audio and video control variables topic.

This functionality can be reviewed in the *audio control's* related exercise.

13.2.5 Menu numeric control related functions

There is a single function associated to this type of numeric control.

- **<menu control id>.setOptions(str)**: This function allows the programmer to enter the options displayed by the related menu via a character string (the *str* argument in the example). When defined only via the configuration editor, these controls have their *options* parameter where the menu's options are entered separated by commas. However, it is sometimes necessary to set their options via something a user might enter, such as a character string.
  Consider the following example. The user fills some text-only *text field* type controls: *t1* and *t2*, and then wants a menu to display what was filled as the options of an *m1* menu. Then, both text field controls actions could be to calculate. And the parameter of calculation would be:

```
str=t1+','+t2
m1.setOptions(str)
```

Note that the first assignment creates a character string which is the concatenation of the text value of *t1* with a comma, then concatenated with the text value of *t2*. This is then assigned to the *str* variable. The format emulates the way the options would be entered in the *options* parameter of the menu, if it were entered via the *Controls* tab of the configuration editor. The second line starts with the menu's *m1* identifier, followed by *.setOptions(str)* the string containing the options is the argument of the function. When this function is executed, the *m1* menu's options change automatically to whatever the user entered in *t1* and *t2*. This function is particularly useful since it allows for a more dynamic behavior of the menu numeric control.

13.2.6 Matrix and array information transfer through text variables

We check first the functions involved in transferring information between matrices and text variables. Later on, and since the information here included is complicated, an example is included so as to clear doubts.

- **_MatrixToStr_('<matrix id>')**. It stands for *matrix to string*. Its argument is a character string in which the identifier of the matrix is entered (<*matrix id*> in the example).

  The function returns, in the form of a character string, the contents of the matrix.

  The returned text uses label format. The first line consists of the identifier of the matrix flanked by the usual <> brackets. For example, <M> would be the first line for an *M* matrix. Later, in another line, the values of the entries (or columns) of the first row are listed, each separated by the following symbol ¦. Each line represents one row of the matrix. After the last line, the closing label with the identifier of the matrix is present (</M> for a matrix with an *M* identifier).

  We see now a brief example. Consider the following instruction: `mstr=_MatrixToStr_('M')`. This will dump the contents of the matrix in an *mstr* variable. If *M* is a 2×2 matrix, and its entry values are *M[0,0]=1*, *M[0,1]=2*, *M[1,0]=3*, and *M[1,1]=4*, *mstr* would end up having:

```
<M>
1 ¦ 2
3 ¦ 4
</M>
```

- **_StrToMatrix_(mstr,'<matrix id>')**. It stands for *string to matrix*. Its first argument is the variable holding the character string where the contents of the matrix are stored.

The second argument is the identifier of the matrix into which the contents are to be passed. The function, therefore, involves two character string type arguments. If the second (the identifier) is to be entered explicitly, it must therefore be flanked by single quotes (‘’).

This function does the opposite of what *_MatrixToStr_()* does. It does not return a value as such, it only internally passes information stored as a character string into a *DescartesJS* matrix.

As a brief example, suppose the information is stored in a *mstr* variable as a character string. We wish to pass the data into a matrix with an *M3* identifier. Suppose the information inside the variable is the following character string:

```
<M>
1 ¦ 2
3 ¦ 4
</M>
<M3>
4 ¦ 3
2 ¦ 1
</M3>
```

Note that *mstr* has information pertaining to an *M* matrix, and information pertaining to the *M3* matrix. If the `_StrToMatrix_(mstr,'M3')` function is called, *DescartesJS* searches the character string with all this information, but it only extracts the data related to the *M3* matrix (the second argument in the function specifies this). And it dumps that data in the *M3* matrix. The *M3* matrix ends up having the following entry values: *M3[0,0]=4*, *M3[0,1]=3*, *M3[1,0]=2*, and *M3[1,1]=1*.

The *M3* **may be previously declared or not** in the *Definitions* tab. If already declared, the data is dumped on it, and any previous entry values are overwritten. Otherwise, the matrix is internally declared and the information dumped in it. The matrix rows and columns variables (<*matrix id*>.*filas* and <*matrix id*>.*columnas*, respectively) adjust their information, taking precedence over any value that may have previously been assigned to the number of rows and columns via the definition of the matrix in the *Definitions* tab of the configuration editor.

We now do a full exercise to better understand these two functions.

This exercise's interactive scene, along with the instructions to build it, can be found at Matrix String. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

Let us summarize what we achieved by doing this exercise. First, the content of an *M* matrix is dumped, in the form of a character string, in an *mstr* variable. Afterwards, the entries of the *M* matrix are all set to zero. Then, the matrix is given its initial values

again by passing them on from the *mstr* variable in which they were stored. It is important to remember that, for the *_StrToMatrix_* function to work correctly, the name of matrix (its second argument) has to match the identifier of the matrix to which the data will be passed on. As already mentioned, the *mstr* variable can potentially have more data besides the values of the entries of *M*. However, the function will only search those inside the <*M*></*M*>, since its second argument is `'M'`. Finally, the last step shows it is possible to dump values stored in a character string on to a matrix, even if it has not been previously defined (the *mstr2* variable holds the character string, and the *M2* previously undefined matrix is created on the fly.

Array information can also be stored and retrieved, similar to how it is done with matrices.

- **_VectorToStr_('<array id>')**. Its name stems from the fact that arrays are called *vectores* in Spanish. So, the name stands for *array to string*. This function has a single argument: the array's identifier flanked by single quotes (''), since it is in the form of a character string.

  The function returns a character string corresponding to the array's contents, and is therefore usually assigned to a variable. As an example consider an *arr* variable where the *V* array's contents are to be stored. Then, `arr=_VectorToStr_('V')` achieves just that.

  The format of the stored information is similar to the one used in matrices. The label appears in the first line, and is the array's identifier flanked by brackets. For example, `<V>` for an array with a *V* identifier. Then, in a second line, the first entry of the array. And so on, each entry appearing in a different line. Once all entries are accounted for, the label closes (`</V>` in the example).

- **_StrToVector_(arr,'<array id>')**. Again, its name stems from the fact that arrays are called *vectores* in Spanish. So, the name stands for *string to array*.

  As with matrices, this function passes the values stores in a character string into an array. For instance, `_StrToVector_(arr,'V')` will pass the contents of the character string (the first argument: the *arr* variable) on to the entries of an array with a *V* identifier. The format of the character string information is the same one mentioned previously.

  As is the case with matrices, the data in the character string may contain surplus information. The function will only pass on the information pertaining to the array indicated via the function's second argument. Note that both arguments of the function are in the form of character strings.

  If the array into which the information is passed on has not yet been defined in the *Definitions* tab of the configuration editor, it will be created on the fly, just as what happens with matrices. If it had been previously defined, its data will be overwritten and its number of entries redefined according to the data it receives.

The character string format of information from an array is very similar to its matrix's counterpart. The only difference is that each line corresponds to an entry of the vector, while for matrices, each line is made up of diverse data (the rows), each separated by the ¦ symbol.

So far, the manipulation of the data from matrices and arrays, and which is stored as character strings, is done inside the scene only. It is sometimes necessary to save this data in a file external to the scene, to later be retrieved. This is handled in the external data saving and retrieving topic.

13.3 Boolean Operators and conditionals

Boolean operators are used to compare values of *DescartesJS* elements. They return either a 1 or 0 value, depending on the comparison's result. The elements involved in a comparison can be constants, variables, and even the values returned by functions.

If the result of a comparison is true, a 1 value is returned, and 0 otherwise. We now list the boolean operators available in *DescartesJS*.

13.3.1 Boolean operators and their use in conditions

The folowing list contains the operators available in *DescartesJS*, as well as examples of how they are used to compare values.

- == (identical to): This operator consists of two equal signs, the second immediately after the first (==). It compares the element before the operator with that after it. If they are **identical**, a true verdict is given and a 1 value returned. If they are not identical, a false verdict is given and a 0 value returned. For example, *2==2* returns a 1 value since the condition is true, whereas *2==1* returns a 0 value since the condition is false.
  DescartesJS is flexible, and allows the comparison to be made if only one equal sign is used (=). However, since this symbol is associated with the assignment of values to variables, it should not be used to make comparisons.

- **!=** (different from): This operator (an exclamation sign followed by an equal sign) compares the element before with the one after the operator. If the values are **different**, a 1 value is returned. If they are not different, a 0 value is returned. For example, *2!=2* returns a 0 value, while *2!=1* returns a 1 value.

- < (less than): This operator compares the element before it with the one after and, if the former is numerically **lower than** the latter, a 1 value is returned (and 0 otherwise). For example, *2<1* returns a 0 value since the condition is false, whereas *1<2* returns a 1 value since the condition is true.

- > (greater than): This operator compares the element before it with the one after and, if the former is numerically **greater than** the latter, a 1 value is returned (and 0 otherwise). For example, *2>1* returns a 1 value since the condition is true, whereas *1>2* returns a 0 value since the condition is false.

- <= (less than or equal to): This operator compares the element before it with the one after it. If the former is **lower than or equal to** the latter, a 1 value is returned (and 0 otherwise). For example, *2<=2* returns a 1 value since the condition is true, whereas *3<=2* returns a 0 value, given that this condition is false.

- >=: (greater than or equal to): This operator compares the element before it with the one after it. If the former is **greater than or equal to** the latter, a 1 value is returned (and 0 otherwise). For example, *3>=3* returns a 1 value since the condition is true, whereas *3>=4* returns a 0 value, given that this condition is false.

- **!** (*not* operator): This operator, which consists of a single exclamation mark, negates whatever condition comes after it. If the condition that follows is true, a 0 value is returned. And if it is false, a 1 value is returned. Negation basically turns a 1 value into 0 and vice versa. For example, *!(3>4)* returns a 1 (true) value, whereas *!(3<4)* returns a 0 (false) value. An even more direct example is *!1*, which returns a 0 value; or *!0*, which returns a 1 value.

- **|** (*or* operator): This operator, sometimes called *pipe*, consists of a vertical bar that is usually entered by typing the key left of the 1 key in the alphanumeric keyboard. It separates two conditions, or boolean values; and if either of them is true (or has a 1 value), the whole expression returns a 1 value. For example, *(2>3)|(3>1)* returns a 1 (true) value since 3 is effectively greater than one. So, the first condition is false, but the second is true. False *or* true returns a true (a 1 value). However, *(2>3)|(3>4)* returns a 0 since neither condition is true.

- **&** (*and* operator): This operator, also known as *and*, consists of an ampersand symbol. It separates two conditions, or boolean values; and if either of them is false (or has a 0 value), the whole expression returns a 0 value. For example, *(2>3)&(3>1)* returns a 0 (false) value since the left hand side condition is false. A similar example, *(2>3)&1*, also returns a 0 (false) value since, again, the left hand side condition is false. However, *(2<3)&(3<4)* returns 1, since neither condition is false.

Conditions are frequently used to assign certain values to variables. Say, for instance, that an *a* variable is to be assigned a 10 value if the value of a *u* variable is greater or equal to 50; and a 20.5 value otherwise. The following assignment would have just that function:

```
a=(u>=50)?10:20.5
```

Note that this code starts as a typical value assignment to an *a* variable. However, after the = symbol, a condition is entered between parentheses: *(u>=50)*. This condition is only true if the value *u* holds is 50 or higher. A *DescartesJS* condition in an assignment forcefully

has a question mark (?) after it. Immediately after it comes the value to be assigned to the variable **should the condition be true** (or have a 1 value). A colon symbol (:) follows this value, and immediately after it comes the value to be assigned to the variable **should the condition be false** (or have a 0 value). Consider the following example:

```
a=((c<2)|(d>1))?j:sqrt(k)
```

To better understand this example, let us give the *c* and *d* variables some test values. We start with 1 and 0, respectively. *c<2* returns a 1 value (since the condition is true), but *d>1* returns a 0 value (since the condition is false). Afterwards, these couple of boolean values (1 and 0) are operated via an *or* operator. So, the whole condition boils down to *1|0*. The *or* operator returns a 1 (true) value is either argument is true (or 1). So, the whole condition in the example returns a 1 value, and the value right after the question mark is assigned to the *a* variable. The *a* variable ends up having the value of a *j* variable. Nonetheless, if the *c* and *d* variables were to have 3 and 0 values, respectively, then *a* would be assigned the square root of the value of the *k* variable, since the whole condition returns a 0 value in that case (checking this result is left as a challenge for the reader).

Let us do an exercise to even better understand these concepts. This exercise's interactive scene, along with the instructions to build it, can be found at Conditions. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

The text displayed in this exercise allows us to evidence the behavior of unitary conditions (where one value is compared with another). The *showred* variable value is the result of a slightly more complicated condition (one built from two unitary conditions). Finally, a useful application of using conditionals was presented: the ability to show or hide a graphic element depending on the value of a certain variable. This can be extended to determine whether controls are displayed or active as well.

The use of a single variable in a conditional parameter such as the *draw if* one in graphic objects is also a noteworthy behavior in this exercise. The *showyellow* and *showred* variables, since they adopt the value returned from a conditional, are already binary. Though the condition in the *draw if* parameter could be *showyellow==1* and *showred==1*, they can be further simplified to *showyellow* and *showred*. When the variables used for such decisions have more than 2 values, they are no longer binary and cannot be used alone. For example, suppose an *a* variable controls which feedback text is displayed when a student answers a question. 0 is assigned to it when no feedback is to be displayed, 1 when the answer is correct, and 2 when the answer is incorrect. *a* then has 3 possible values. In this case, an explicit comparison between *a* and the possible values has to be implemented.

Remember the & and | operators allow the combination of unitary conditions. When these operators are used, the correct use of parentheses as grouping elements is very important for the scene to behave as required.

Colors were applied to some objects in this exercise. Feel free to review the color editor topic for more information on this functionality.

13.3.2 Using mute variables to condition the execution of functions

Sometimes, a decision has to be made whether to execute one particular function, or another. Since functions return values to a certain variable, this decision can be made using a condition. Consider the following instruction:

```
blah=(z>3)?GenerateData():blah
```

Here, if the z variable is greater than 3, a *GenerateData()* function is called, and its return value assigned to a *blah* variable. If $z \leq 3$, the *blah* variable is assigned its own value, thus doing absolutely nothing new. *blah* is an example of a mute variable.

The function may very well not return anything at all. I. e., it may be just an algorithmic function that returns no particular value. However, using *blah* as a mute or dummy variable allows the code to be a bit clearer.

DescartesJS actually does not require functions to explicitly return their values to a variable. The previous instruction may actually be simply:

```
(z>3)?GenerateData():0
```

In this example, the function is executed when $z > 3$. If it returns a value, that value will be dumped in a *DescartesJS* internal dummy variable and be ignored. Otherwise, a 0 value will be dumped in a *DescartesJS* variable and be ignored. So, it is not necessary to use dummy variables to conditionally execute functions. However, some developers prefer to do it since it makes the code clearer.

13.4 Generic operators

Generic operators are math symbols commonly used to assign values, add, subtract, multiply, divide, and group, among others with more specific functions. Some of them also have alternate functions, such as the concatenation of character strings.

Operators work on constants, variables, elements returned by functions, and texts.

- =: This operator is used to assign values. When used, the value of the constant, variable, or expression at the right of the symbol is taken and assigned to the variable at its left. Numeric values and character strings can be assigned to a variable. For example, `a=2` takes the value 2 at the right and assigns it to the *a* variable. `b=2*3` takes the value of the expression at the right, which is 6, and assigns it to the *b* variable. `c='hi'` takes the character string at the right and assigns it to the *c* variable. If this last variable is printed, *hi* is the printed text.
  There is an important difference between the single equal operator (this one used for assignments) and the double equal operator (==). The latter is used to compare values of expressions, and is discussed in the boolean conditions and operators topic.

- **+**: This operator is used to add the values of variables and/or constants. It is also used to concatenate character strings. For example, `a=2+3` assigns the *a* variable the value of 5. But, `b='I'+'salute'+'you'` assigns the *b* variable the *I salute you* character string.
  "Adding" a numeric value with a character string first changes the numeric value into a character string, and then concatenates them. For example, `c=2+'3'` assigns the *c* variable the *23* character string.

- *****: This operator multiplies the values of the expressions at its sides. For example, `a=2*pi` assigns the *a* variable the value of 6.283...

- **/**: This operator divides the value at its left side by that at its right, and returns the value. For example, say an *a* variable has a 3 value and a *b* one has a 1.5 value. `c=a/b` assigns a 2 value to the *c* variable.

- **∧**: This *power* operator takes the element at its left and elevates it to a power that corresponds to the value of the element at its right. For example, `q=2^3` assigns a value of 8 (2 times 2 times 2) to the *q* variable.
  This operator can be used to extract roots different than the square root. For example, `r=3^(1/3)` extracts the cubic root of 3 and assigns this value to the *r* variable.

  There is an important behavior regarding the power operator when it is used to graph power functions. Say `y1=x^0.6` and `y2=x^(3/5)`. These two functions are mathematically equivalent. They are both $x^{3/5}$. However, if both equations are plotted via an *equation* graphic object, only the part of the graph with the *x>0* domain is plotted. This happens because $y = x^p$ functions are only defined for positive values of x when p is not an integer. 0.6 and (3/5) are taken as real non integer values and the function for negative values of x is not plotted.

  So, when a $y = x^{\frac{p}{q}}$ function is plotted, with q a positive odd number, to explicitly indicate the q denominator is to be taken as the q-th root, `y3=(x∧3)∧(1/5)` would have to be entered (according to our example). This function is therefore defined for x within the real numbers.

  To summarize, in order to indicate that a function is the q-th root, it is necessary to enter it as $wedge(1/q)$ (the argument elevated to 1 over the number corresponding to the root to be extracted). The function's domain is then taken to be all real numbers.

- **%**: This *modulo* operator takes the value at its left side and divides it by the value at its right side. It returns the remainder after such a division. For example, `8%3` returns a **2** value, since 7=3*2+**2**. Another example, `3.8%1.2` returns a **0.2** value, since 3.8=3*1.2+**0.2**.

- **( )**: Parentheses, as operators, are used to group expressions in assignments and/or conditions, so as to specify the order in which other operators such as product or

sum are implemented. *DescartesJS* follows the typical precedence of operations common to all programming languages. Parentheses can be used to ignore these and specify a different order. The math operations order and hierarchy topic can be reviewed for more information on this functionality.

DescartesJS may display certain error messages when the programmer makes mistakes in assignments or operations:

- *Infinity*: This error appears typically when a division by zero is attempted. If a value is to be printed and a division by zero is involved in its calculation, the *Infinity* text is printed in its stead.
- *NaN*: This error appears when *DescartesJS* attempts to print a value whose calculation involves an invalid operation. An example is the square root of a negative argument, or the operation of character strings with numbers using operators different from +. The + operator is valid in this case since it is used to concatenate.
- Console errors: These errors appear explicitly in the DescartesJS console. It is therefore recommended to have it open while developing a complicated scene prone to human error. The errors here reported are usually explicit as to their nature. For example, "*faltan paréntesis por cerrar*" (Spanish for *closing parenthesis missing*) when the number of opening parentheses and closing ones does not match. The location where the error is detected may not be explicit. So, besides having the console open, it is also a good practice to constantly keep an eye on it so as to be able to pinpoint the error to a new part of code recently added.
  The errors reported in the console are not necessarily recent. So, another suggestion is to close and open the console so as to clear its contents. If an error is displayed in a console that was recently cleared, it means the error is still present in the code.

13.5 Math operations order and hierarchy

Just as with any other programming language, or even any calculator, *DescartesJS* follows a predetermined order or hierarchy of operations when several operations are implemented in a single instruction. The power are first dealt with, then multiplications and division, and finally additions and subtractions. And the order for operations in the same hierarchy is from left to right (the leftmost operation is handled first, and so on). Consider the following assignment:

```
a=3+2*3+4/5^2
```

First, the powers are implemented, so the expression is reduced to

```
a=3+2*3+4/25
```

Then, the products and divisions are done, from left to right, leaving

`a=3+6+4/25`, and then `a=3+6+0.16`

Afterwards, the additions and subtractions are done, also left to right, leaving first

`a=9+0.16`, and then `a=9.16`

It is frequently necessary to specify a different order of operations to *DescartesJS*. This is done using parentheses as grouping operators. For example, consider an expression to extract the 5 root of 32 ($\sqrt[5]{2}$). This is achieved using $32^{\frac{1}{5}}$. A common error in this example would be to use the following expression:

`32^1/5`, instead of `32^(1/5)`

The former expression first performs the power, thus obtaining

`32/5`, which finally leads to `6.4`

It is therefore necessary in this example to use the parentheses to indicate the order of operations. By using

```
32^(1/5)
```

DescartesJS first attends to the contents inside the parentheses. This group returns a 0.2 value, and the 32 is then powered to the 0.2 exponent, finally returning the expected 2 value result.

Parentheses thus alter the order of operations. When parentheses are found, their contents are parsed first, and their returned value is then considered in the outer expression.

Parentheses can be nested inside other parentheses. In these cases, the innermost ones are dealt with first, and so on until the outermost layer is reached. Consider the following expression

```
2*(1+3/(2+1))
```

DescartesJS first encounters the outermost parenthesis. It then searches its contents to see if there are other parentheses inside it, and finds one. Searching its contents for more parentheses inside, it finds no more and evaluates the expression in this inner parenthesis, simplifying the expression to

```
2*(1+3/3)
```

DescartesJS continues by evaluating the contents if the remaining parenthesis. In it, it solves first products and divisions left to right, obtaining

```
2*(1+1)
```

It then does the additions and subtractions in the parenthesis

`2*2`,

From which the final 4 value is obtained.

Parentheses allow for a more flexible behavior y enabling the user to dictate the order of operations. This, however, is commonly related to human error. An opening parenthesis may be missing its closing one, or vice versa. This type of error happens more frequently when using lengthy expressions. One way to avoid this is to immediately close

the parenthesis after inserting an opening one. Once done, the inner contents of that set are entered between them. Yet another way to avoid this is to use modular expressions. A lengthy expression may be replaced by more modular assignments. This has the downside of requiring more variables, but allows for an easier expression. The errors mentioned are discussed more in depth in the DescartesJS console topic already described, as shown in Figure 13.5.0.1.

Note that the error in this figure results from a *text* graphic attempting to print the value of `(3+2*(1/3)`, in which a closing parenthesis is missing. If the missing parenthesis were replaced, the expression would be `(3+2*(1/3))`, returning a 3.6666... value, which is what is finally printed at the top left corner of the scene in the figure. *DescartesJS* can sometimes "sense" what the developer meant to enter, but only to some extent. This type of error messages are only displayed in the console. That is why we recommend keeping it open and at the front, particularly when dealing with a complicated piece of code.

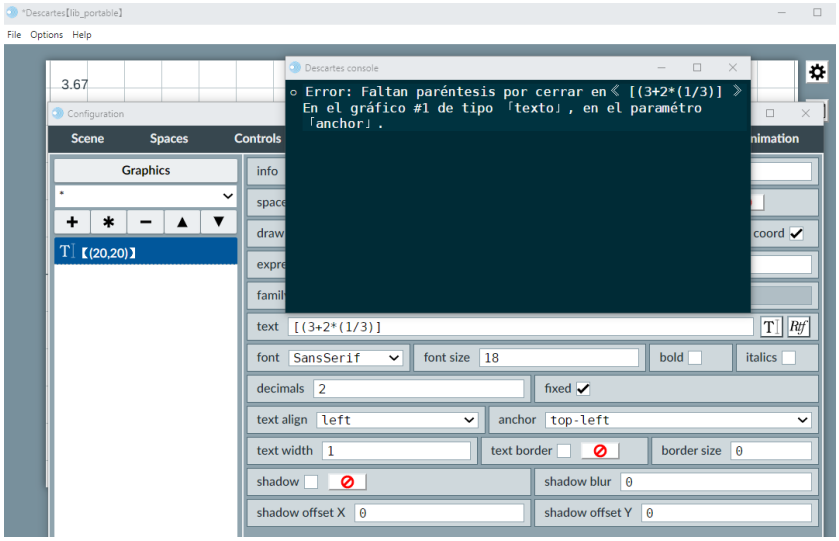


Figure 13.5.0.1: Console error message when attempting to evaluate an expression with a mismatch of opening and closing parentheses.

13.6 Update order when handling a scene

When *DescartesJS* loads a scene, and when a user interacts with it, a certain order of updates and instructions is carried out. The user needs not remember all this most of the time. However, certain weird behaviors in a scene may be explained by this functionality.

For example, consider a *DescartesJS* scene containing a spinner with a *n1* identifier. What happens when the scene is loaded if the spinner is given an initial 3 value, but there is a `n1=7` assignment in the *INICIO* algorithm? And what happens if, say, we include an event that assigns *n1* another value? What value will it end up having when the scene

finishes loading? How will it change following a user's interaction with the scene? How will it change if an animation is run?

These questions can be answered if the *DescartesJS* predetermined order of executions is understood.

13.6.1 Updates upon loading a scene

When a scene is loaded and launched, the following executions take place and in the following order:

1. **Objects' construction**. In this first moment, all the objects conforming a scene are built with the default values.
2. **Beginning phase**. Once the objects are present, this second moment consists of the following updates in this order:
   (a) Definitions present in that tab (arrays, matrices, etc.).
   (b) Algorithms in the Programs tab.
   (c) Controls defined in the tab with the same name.
   (d) Spaces. This update includes the calculation of variables related to spaces and the update of graphic objects housed in them.
3. **An update phase for all values**. This phase is included so as to ensure all objects are using their updated values instead of their predefined (default) ones, and is rather important so events can work properly.

Each element in this list can involve several executions. For example, there may be executions implemented in the arrays and matrices defined in the *Definitions* tab. These are performed in the order in which they appear in the list. This applies also for the other tabs involved.

All this defines how a scene is shown upon loading, following the clicking of the *Apply* or *Ok* button in the configuration editor. However, certain actions also take place when a user interacts with a scene, or it suffers changes from an animation, for instance.

13.6.2 Subsequent updates

Any subsequent interaction from a user, or any animation step (if there is an animation), triggers an update cycle with the following actions in the order in which they are listed.

1. **Auxiliary elements update**. The auxiliary elements in the *Definitions* are first updated.
2. **Controls update**. An initial update of all controls.
3. **Events update**. An update of any events found in the *Programs* tab.
4. **Controls update**. A second update of all controls. Its purpose is to ensure they hold their updated values.

5. **Spaces update**. All element related to spaces are updated.

The second control update is included to respond to situations where a control (say *b*) at the end of a controls' list may result in modifying one before it (say *a*) in the list. In this type of situations, *a* will not end with its updated value if the second controls update is not performed.

Say, *a* is a control with its starting value set to 0 and its minimum value (via its *min* parameter) is set to *b*. *b*'s value was recently changed to 1. A first sweep of the controls will leave *a* with its 0 starting value. However, when reaching *b*, its 1 value is implemented. The second sweep will ensure *a* ignores its 0 value and used *b*'s one instead, since *a*'s minimum value is *b*, and *b*'s starting value is greater than *a*'s one.

As mentioned before, the developer is not required to know this functionality. It is seldom necessary. However, knowing the order can help make clear certain behaviors in a scene, such as why a variable adopts one value or a different one depending on where the assignment occurs. It can also help ensure the correct assignment of values to controls.

14

Data saving and retrieving

This chapter deals with the saving and retrieving of data generated by a *DescartesJS* scene. This type of data handling is used via matrices, so this chapter also involves a considerable amount of matrices' functionality.

It is occasionally necessary to store large amounts of matrices data in a character string. When this is done, *DescartesJS* separated the entries' values with special symbols so that they can be easily read afterwards. Eventually, this data is typically assigned to a variable. And then the data can be saved in a text file. This file can later be used to retrieve the data back into a matrix.

To understand this functionality's usefulness, consider the following case. The results of a simulation done in *DescartesJS* are to be used in a scene. These results are lengthy, and they are stored in a very big matrix. However, the simulation is complicated and takes a lot of time. Since only the results are important, it is not desirable that *DescartesJS* performs the simulation every time the scene is launched. A user would grow tired of waiting, say, 5 minutes for the results every time the scene is loaded. The results of the simulation could alternatively be saved to an external text file, and *DescartesJS* would only need to retrieve this data when the scene loads, instead of running the whole simulation again. By doing this, the developer would only run the simulation once, save the data, and then configure the scene to load such data when launching.

14.1 Saving and retrieving information in files

DescartesJS can save data into files via character strings. It is also possible to load the data from the file into a character string, to then be stored in a variable.

- _**Save**_(). Used to save a content to file.

  This function has, as a first argument, a character string with the path and name of the text file in which the information is to be saved. If the file has an extension, it is also included. The second argument is the content to be saved. For example, if `_Save_('MyFile.txt','my data')` is executed, will launch a *DescartesJS* save dialog. It will already have *MyFile.txt* suggestion included by default, which the user may modify. If the file is saved, it saves the second argument's character string (the *my data* text in the example). The second argument is usually a variable which contains the character string to be saved, since it is typically used to store large amounts of data, and not a single phrase.

In order for this function to work properly, the scene's *html* file has to be already saved in some folder of choice. This is necessary since the path to the save file needs to have a reference. An example of the best way to call the function is `_Save_('./MyFile.txt',data1)` (the first period included indicates to start in the folder where the scene's file is saved). It is not absolutely necessary to include the path. The file's name suffices if the scene has accessed that path previously. However, we recommend always including the path.

- **_Open_()**. This function opens a file selected by the scene's user. It cannot be used to open files pre-selected inside the scene. It merely launches a dialog with which the user selects the loacl text file to open, and dumps its name, as well as its contents, into a couple of variables.

  Its single argument is a character string that corresponds to the name of the function (without its arguments parentheses) in which a couple of variables with fixed names can be used to pass the name of the file and its contents into the developer's own variables.

  As an example, consider a scene with a button control whose action is set to *calculate*, and its calculation parameter is `_Open_('getData')`. For this to work, the scene has to have a *getData()* function in its *Definitions* tab. In this function, two assignments should be present:

  ```
  fName=DJS.fileName
  fContent=DJS.fileContent
  ```

  In this example, an *fName* variable is assigned the name of the file (*DJS.fileName* is a fixed variable which holds the name of the file); and *fContent* is assigned the contents of the file (*DJS.fileContent* is also fixed).

- **_Load_()**. This function opens a file from a drive. The function returns the contents read from the file, and it is therefore typically assigned to a variable.

  Its single argument is a character string with the path and file name to read.

  For example, if `dt=_Load_('MyFile.txt')`, the *MyFile.txt* file's contents are dumped into a *dt* variable, which ends up holding such data as a character string. Since paths are also allowed, another example is `dt=_Load_('./data/MyFile.txt')`, which would read the data of a *MyFile.txt* file stored in a *data* folder present in the folder where the scene's file is located, and then dump its contents in the *dt* variable.

  It is **very important to remember that**, due to browsers' security preferences, this function **only works in scenes used in the *DescartesJS* editor**, and not in scenes opened in browsers.

  The main differences between the *_Open_()* and *_Load_()* functions is that, on the one hand, *_Load_()* allows the scene to access files with their names indicated inside the scene (the instruction insice the scene specifies which file to read and it is not necessary to select it via a dialog). On the other hand, *_Load_()* only works if the scene is opened in the *DescartesJS* editor, and not when it is opened in a browser.

Thanks to this last reason, *_Load_()* is commonly used only to prepare data in a scene that will eventually hold such data inside its own code; or in scenes intended to be used solely in the *DescartesJS* editor.

All this may seem confusing, and an exercise might be the best way to clear doubts. This exercise's interactive scene, along with the instructions to build it, can be found at Open Load Save. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise uses a text file previously generated in a *UTF-8 without BOM* code supporting text editor such as *Notepad++*. This coding is used so that no special characters that may prevent *DescartesJS* from correctly reading the file are present. The data is read using the *_Open_()* function, and then stored in another file (*File2.txt*) using the *_Save_()* function. They are finally read also using the *_Load_()* function.

Again, due to security preferences common to all browsers, *DescartesJS* cannot read a particular file using the *_Open_()* function if the file has not previously been selected by the user. It is therefore not possible for the scene to open the file directly. So, if the scene is opened in a browser, the *_Open_()* function is used as a means for the user to select which file to open via a dialog, and once selected, the file can be opened. Alternatively, the *_Load_()* function can be used for such a purpose, but its disadvantage is that it is only guaranteed to work properly in the *DescartesJS* editor, and not necessarily in a browser. Finally, the content of the *File.txt* file is the data of a matrix, though it is never assigned to one. Remember the *_StrToMatrix_()* and/or *_StrToVector_()* functions can be used to pass data from a character string to a matrix and an array, respectively.

14.2 Data saving and retrieving within a scene

It is frequently preferable to save data within the scene's *html* file. This guarantees there will be no problems reading the data due to access permissions.

For example, suppose there is data in a matrix that takes long to generate via a simulation due to the complexity of the calculations. The scene may freeze, or the browser starts to display dialogs indicating the user that the scene is taking too long to respond. All these are undesirable behaviors in a scene, since the user may become confused. The data stored in the matrix can be saved in a text file using the functionality described in the Matrix and array information transfer through text variables topic. Additionally, and in order to ensure the data will be retrieved, it may be desirable to embed this content as a part of the *html* code of the scene inside a *script* label block. A *DescartesJS* scene has no trouble reading data from its same *html* file.

The *script* block in question has an opening label: `<script id=''path and name of the file'' type=''descartes/archivo''>`, and a closing `</script>` one. *archivo* stands

for *file* in Spanish. For example, `<script id=''./dat.txt'' type=''descartes/archivo''>` indicates a *dat.txt* file is the one read. Its contents appear between the opening and closing script labels. This data file can be generated first, and then its data is copied into the *script* block inside the *html* code of the scene. Again, this ensures no problem will arise when attempting to read this file, since it is contained in the scene itself. The suggested location of the script block inside the *html* code is immediately after the closing label of the *div* block that contains the *DescartesJS* scene code.

There are a few pointers regarding the embedding of data as script blocks. Even though it is not indispensable, we suggest using the `./`, and not only the file's name, inside the opening *script* label. This further ensures that the file is correctly found.

We do now an exercise to better understand this functionality. This exercise's interactive scene, along with the instructions to build it, can be found at Save Script. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

It is worth remembering that the *_Load_()* function does not allow loading files foreign to the scene locally. Notice, however, that the *Data.txt* file is embedded inside this same scene via a *script* block. So, the *_Load_()* function can be freely used.

Note as well that the scene was first built in *DescartesJS* and then its code edited in a text editor. However, it was necessary to close *DescartesJS* prior to the text edition. This was done so that edition done in one does not alter the changes made in the other. Though this rarely happens, it is good practice to edit with one tool at a time.

The last step in the instructions of this exercise involves opening the scene in a browser and clicking the *Load* button. This is done so as to verify that the scene will successfully read the data in the script block when opened in a browser.

This exercise handles the data from a 2×2 matrix. It is a very simplistic example. However, consider the advantages when handling very large matrices, where a long time may be involved when calculating the values for each entry. On one hand, this *saving the script* strategy allows the data generated from a simulation to be stored inside the scene's *html* file, so the simulation will have to be run only once. On the other hand, the script block in the *html* file ensures the data is readily available, whether the scene is viewed locally in a browser, or is opened from a server. One must, however, anticipate larger scene files if the *script* block is lengthy.

15

General tools

Some tools in *DescartesJS* do not belong to a particular tab. They are distributed among several tabs and their functionality is not restricted to a certain control, or graphic, etc. This chapter is dedicated to such tools.

15.1 Color editor

This tool allows for colors, color gradients, and image patterns to be used in graphic objects as well as backgrounds for control and space elements. It consists of a window launched when the *color* button of a *DescartesJS* element is clicked. Examples of such elements are: graphic objects, the axes of a cartesian plane, texts, a space's border, and many more. This window includes three different tabs.

The color editor dialog consists of three tabs, all of which have four common buttons: a *Copy* one, by which the user copies a color definition (single, gradient or pattern) to an inner clipboard; *paste*, by which the user pastes a color definition that was previously copied (thus avoiding entering the definition by hand), *Accept* to implement the color definition, and *Cancel* to exit the dialog without implementing the definition.

15.1.1 RGB

This tab of the color editor allows the user to enter a single color. Figure 15.1.1.1 shows the color editor dialog in its *RGB* tab. Note that the color set in it is 8E44AD in hex, with a 0 transparency.

This dialog has the following elements:

- *Custom color menu*: A color menu with some custom colors, such as pure red, pure green, pure blue, and some other color combinations.
- *Copy*: A button which copies the current color in the dialog to an internal *DescartesJS* clipboard, so it can later be pasted using the color editor of some different element. It is particularly useful when many different elements are to have the same color and transparency. One only has to clic the *Paste* button instead of manually entering the color for each.
- *Paste*: A button which pastes a previously copied color definition. When this button is clicked, the color editor of the currently edited element adopts the copied color.

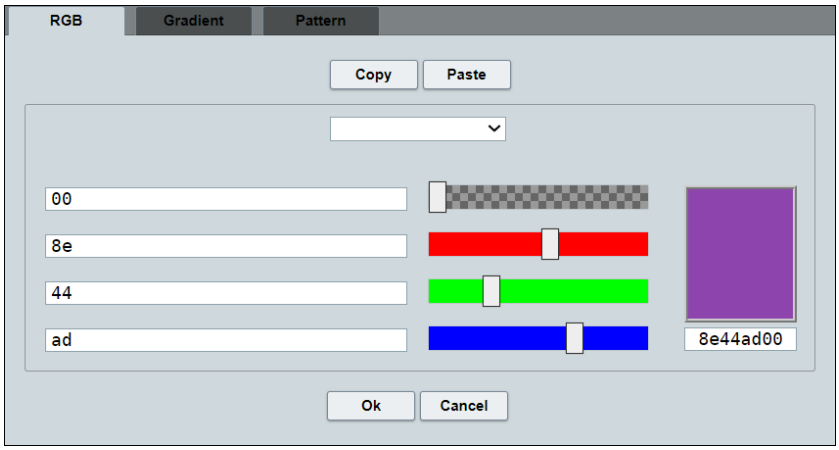


Figure 15.1.1.1: The *RGB* tab of the color editor dialog.

- *Transparency text field and scrollbar*: A text field with a horizontal scrollbar at its right. Both control the element's transparency (i. e., how much of the background it allows to pass through it). By default, it has a zero transparency (100% opaque). However, when it is desirable for an element to let the user view through it, its transparency can be set using the scrollbar. A completely transparent object will not be viewed at all. The text field allows the developer to enter the transparency via text. Variables, constants, and expressions are allowed there. We discuss this functionality soon.
- *Red color component text field and scrollbar*: consists of a text field and a **red** horizontal scrollbar, both which control the amount of the red component in a color.
- *Green color component text field and scrollbar*: consists of a text field and a **blue** horizontal scrollbar, both which control the amount of the green component in a color.
- *Blue color component text field and scrollbar*: consists of a text field and a **green** horizontal scrollbar, both which control the amount of the blue component in a color.
- *Color preview panel*: This panel is a rectangle with the resulting color in it. It also has a text field with the hexadecimal color code (with transparency) of the resulting color. The first six characters of the color code correspond to the hexadecimal color and the last two to the transparency (also in hex). This text field can also be used to define the color. I. e., it can be edited as well.

It is also possible to use an eye-dropper color picker to choose a color when in the *RGB* tab. In order to do this, simply click the color preview panel and circle with a grid inside appears. This circle follows the mouse, even outside the *DescartesJS* application. Place the central square grid of the circle over the color you wish to pick and do a left click. The color is instantly set in the color editor. This functionality is since the user no longer needs to use outside applications to pick a color and then enter it in *DescartesJS*.

If the color is to be entered via text, it can be both entered using a hexadecimal base, or a decimal one:

Hexadecimal color code

Hexadecimal (base 16) color code is an 8 digit color code. The first 6 digits are related to the color, and the last 2 to the transparency. Each color, and the transparency as well, have 256 (16^2) different possible values: 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0a, 0b, 0c, 0d, 0e, 0f, 10, 12, 13 ... 9d, 9e, 9f, a0, a1, a2 ... ff. Both upper and lower case can be used in *DescartesJS*. The lowest possible value is 00 and the highest ff.

When a valid hex value is entered in one of the text fields, the related scrollbar moves to its corresponding place. Likewise, if the bar is moved to a position, the corresponding text field updates its content upon releasing it. The color preview panel (along with the whole color code below it) also update when a color, or the transparency, is modified by using its text field or scrollbar. Finally, if a color code is entered in the whole color code text field below the preview, the single *RGB* (*red - green - blue*) color text fields update themselves, along with their accompanying text fields.

A good practice for the user not familiarized with hexadecimal numbers is to increase one by one the values of one of the scrollbars. This may help better understand how this type of numeration works.

Decimal color code

Though hex notation is the most commonly used to define colors, *DescartesJS* also allows for decimal notation. In this case, the minimum value is 0 and the maximum value is 1. This notation is the one to use when the used color is given by a variable or an expression. However, decimal numbers can also be entered directly in the single color text fields. For example, if `redColor` could be entered in the text field for the red color. The variable's value has to lie between 0 and 1 (including both extreme values). This variable's value may change due to interaction, so the red component of the element's color will change dynamically. However, note that when a decimal expression or variable is entered in a single color text field, the corresponding color bar will shift completely to the left and the preview color. Consistently, the corresponding pair of digits of the color in the 8 digit whole color code below the color preview rectangle switches to 00. In this case, this does not mean no such color component will be introduced. It is only a way for *DescartesJS* to indicate that the number is not in hex format.

It may sometimes be necessary to convert a value in hex to dec, or vice versa.

Hexadecimal to decimal conversion, and vice versa

Scientific calculators have a built-in converter so as to handle values in several different bases. Some web pages, such as this one convert between dec and hex. If, for example, the 9a value is converted to decimal, it turns out to be 154. Since *DescartesJS* handles color

decimally only between 0 and 1, this 154 has to be re-scaled by means of a rule of 3: 0 is to 0 as 154 is to 255 (the highest possible value in dec). So 9a turns out to be 154/255 = 0.603921.... This is the value that has to be entered in decimal form in the single color or transparency text fields.

Note that, when values are entered decimally in the text fields, even if they are the explicit numbers and not expressions, the corresponding scrollbar does not shift. The color preview panel may shift, by setting that color component to a zero value. Nonetheless, the expected color is visible in the interactive scene if the changes are applied. If the value entered is lower than 0, that color component is assigned its lowest possible value: 0. If the value entered is larger than 1, it is assigned its highest possible value: 1.

15.1.2 Gradient

This tab of the color editor allows the user to implement color gradients. Figure 15.1.2.1 shows this tab's elements.

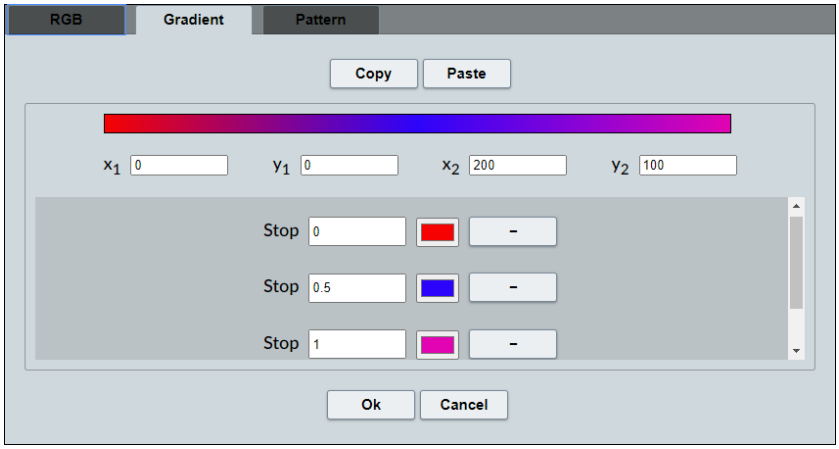


Figure 15.1.2.1: The *Gradient* tab of the color editor dialog.

This tab has the following parameters:

- x_1, y_1, x_2, and y_2: these are four textfields. x_1 corresponds to the horizontal coordinate of the start of the gradient vector. y_2 is the vertical coordinate of the start of the gradient vector. x_2 and y_2 are the coordinates of the end of the gradient vector. These coordinates are **absolute** coordinates. The gradient is drawn following the vector defined herein. And the absolute coordinates are taken regarding the container where the gradient is to be applied (be it a button's background, a space's background, etc., depending on the object housing it).
- +: a button to add a stop. Each stop can be used to define a different gradient. And each stop has the following parameters:

- *Stop*: a text field where a number between 0 and 1 is entered. This number indicates where the gradient color starts. 0 corresponds to the beginning of the gradient vector, and 1 to its end.
- color selector: a button, with a frame with the color used, which when pressed launches a color palette. To select a color, click on the desired one in the palette and it is automatically translated to a color code displayed at the bottom of the palette. The color can also be typed directly as a color code. The code used can be set to RGB, HSL, or hexadecimal.
- -: a button to remove the stop.

The horizontal bar at the top of this tab displays an example of how the gradient looks, and so it is good practice to check it regularly. The color buttons associated to each stop adopt the color selected in their respective palette. Once a gradient has been accepted, the color dialog button used (whether for a button's background, a space's background, etc.) is displayed with the chosen gradient. Figure 15.1.2.2 shows the color palette displayed when clicking on a stop color button.

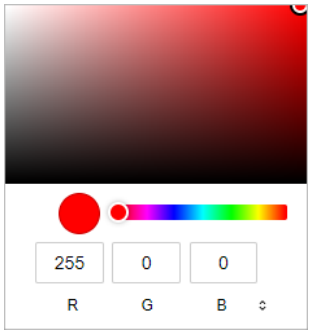


Figure 15.1.2.2: The color palette window of a stop in a gradient.

As an example, consider a *segment* graphic type object set using the following absolute coordinates: `(0,0)(200,200)`. Its *Gradient* tab parameters are then set to $x_1 = 0$, $y_1 = 0$, $x_2 = 200$ and $y_2 = 200$. In this example, the gradient vector follows the same direction as the graphic vector itself (from left to right and from top to bottom). If the stop with the 0 value is set to a yellow color, and the one with the 1 value is set to a red one, the yellow-to-red gradient will be drawn in the direction of the segment. However, if the following values are given: $x_1 = 200$, $y_1 = 200$, $x_2 = 0$ and $y_2 = 0$, the color gradient is then reversed, with the yellow near the bottom and the red near the top. Furthermore, if the following set is given: $x_1 = 0$, $y_1 = 200$, $x_2 = 200$ and $y_2 = 0$; the whole segment is drawn in the same orange color. This happens because the gradient vector is perpendicular to the segment, and the segment lies near the middle of the gradient, where the color is a 50/50 combination of yellow and red: orange.

Let us try another example. Consider a rectangle graphic object with the following absolute coordinates: `(0,0,200,100)`. If this rectangle's fill is set to be a gradient with the

following set: $x_1 = 0$, $y_1 = 0$, $x_2 = 200$ and $y_2 = 100$; a first stop (with 0 value) is set to a blue color; and a second stop (with a 1 value) is set to black; the gradient will then be from the top left corner of the rectangle to the bottom right one, since the gradient vector follows the rectangle's diagonal. However, if the following set is given: $x_1 = 0$, $y_1 = 100$, $x_2 = 200$ and $y_2 = 0$; the gradient goes from the bottom left corner to the top right one. With a gradient vector given by $x_1 = 0$, $y_1 = 0$, $x_2 = 200$ and $y_2 = 0$; the gradient is horizontal from left to right (the same color appears for any two given points with the same vertical coordinate). And if the $x_1 = 0$, $y_1 = 0$, $x_2 = 0$ and $y_2 = 100$ set is entered, the gradient is vertical, from top to bottom (the same color appears for any two points with the same horizontal coordinate).

As a final example, consider the same rectangle as before. As a minor few changes, set the first stop at a 0 value and the second stop at a 0.1 value. Use a horizontal gradient from left to right ($x_1 = 0$, $y_1 = 0$, $x_2 = 200$ and $y_2 = 0$). The blue-to-black gradient only spans the first horizontal 10% of the rectangle. The 90% right hand side remainder is completely black (the color of the last stop included). This behavior responds to the fact that the first stop is set at 0.1 (which is 10% of 1). This same effect can also be achieved if the first stop is set at 0, the second at 1, but the gradient vector is set to $x_1 = 0$, $y_1 = 0$, $x_2 = 20$ and $y_2 = 0$. The effect is the same since the vector gradient's second stop starts at 20, and 20 is 10% of 200 (the rectangle's width).

15.1.3 Pattern

The *Pattern* tab of the color editor tool allows the user to include an image (*jpg, png* or static *gif*) inside the object in which it is implemented (button's background, etc.). Figure 15.1.3.1 is an example of a configuration of this tab.

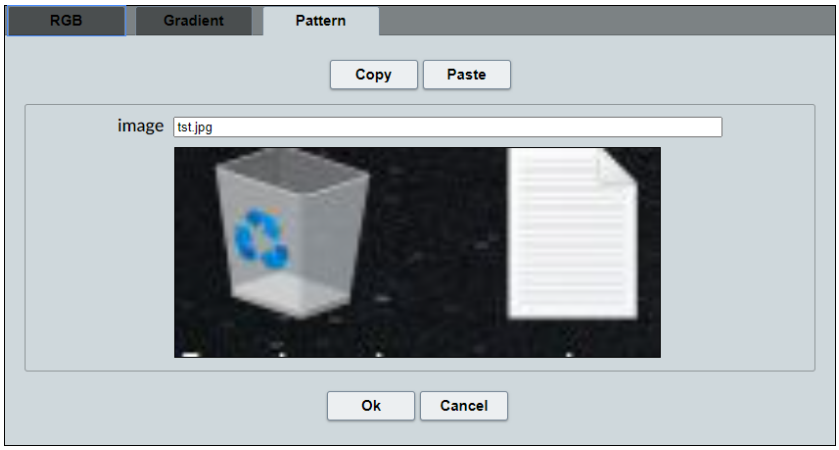


Figure 15.1.3.1: The *Pattern* tab of the color editor tool.

This tab only has one element:

- *image*: a text field in which the path to the image file is entered. If a path relative to the scene's *html* file is to be used, the scene's file should be saved first.

The image starts at the origin of the canvas (the area to be covered where the image is used). From there, it repeats itself (as a mosaic) until the whole canvas is spanned.

When a pattern is used, the related element's image tool button has a generic landscape as its inner image, thus indicating that a pattern is being used.

When a gradient is used as an inner color of a scrollbar control, the color of the bar's handle changes depending on the handle's relative position in the bar. The scrollbar's left and right buttons also adopt the gradient's extreme colors. If the scrollbar uses a pattern instead, the image used is drawn in the scrollbar's handle.

15.2 Text editing tool

The text editing tool is available for a variety of graphic elements, such as the *point*, the *segment* and, more obviously, the *text*. It is also available in elements involving text beyond graphic objects. However, its functionality is basically the same in all cases.

A text field is typically present where text can be entered on the fly. There is normally a field where the text can be entered in a single line. And, afterwards, a *T* button is present, which is used to enter plain text; and an *Rtf* button is present as well, which is used to enter text in rich text format (hence the initials).

15.2.1 Plain text editor

When the *T* button, adjacent to a *text* entry field, is clicked, a window opens where multiple lines of text can be entered comfortably. Figure 15.2.1.1 show such a window.

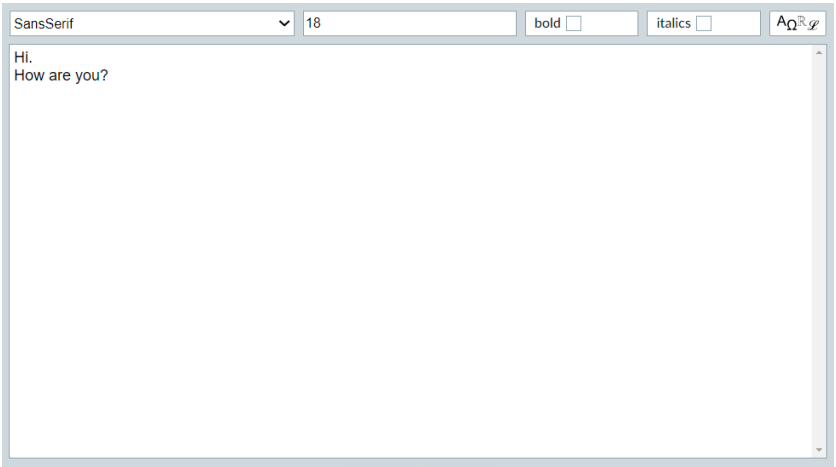


Figure 15.2.1.1: Plain text entry window.

Three different fonts are available: *SansSerif*, *Serif* y *Monospaced*. The selected font is applied to **all** the text in the window (i. e., it is not possible to have different text in

different fonts in a plain text). The plain text editor also has a menu to select the size of the font. Two checkboxes are also present with which a *bold* and *italics* format can be implemented. And, finally, there is a button with various symbols in it, which when clicked displays a panel to select special characters. Figure 15.2.1.2 shows this panel. The left hand slide of this panel has a menu which allows the developer to move more directly to a group of special characters from the same origin. This panel is particularly useful to enter mathematical operators or greek letters.

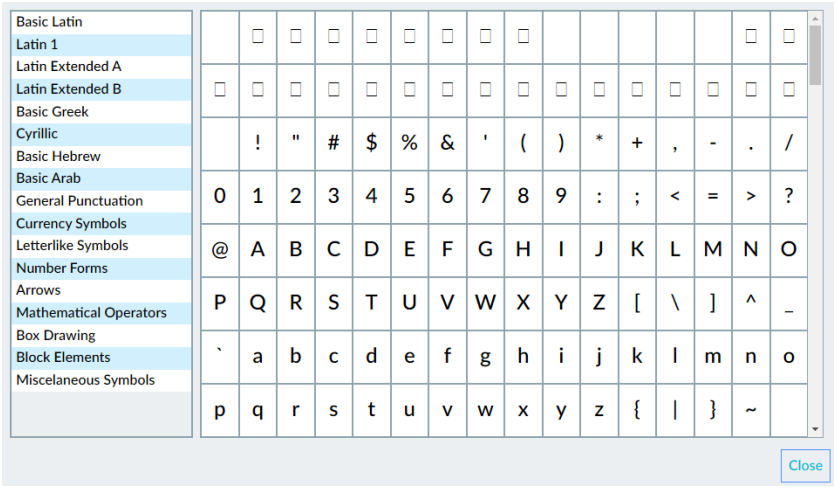


Figure 15.2.1.2: Special character selection panel, inside the plain text editor.

Line breaks in this plain text window are introduced simply by typing the *ENTER* key in the keyboard. Once text has been introduced in this window, it can be accepted via the *Ok* button, or rejected via the *Cancel* button. If accepted, the text is then also displayed in the *text* field of the element. All text appears in a single line, the line breaks represented by \n (a backslash followed by *n*). Conversely, it is also possible to enter the text in the *text* field, using \n to separate lines, and if the plain text window is afterwards opened, the entered text will appear there in multiline format.

Even though there is a *rich text format* entry window, certain instructions in the plain text window can be used to enrich the format of the text to some extent. These are instructions similar to the ones used in the $LaTeX$ language. Edition is limited to color, bold, italics, some alignment options, and a few examples of math language.

- **Word coloring**: When a part of the text is to have a different color, `\color{}{}` is used. The first set of curvy brackets contains the hex code of the color to be used, while the second set contains the text to which that color is applied.

  For example, if `Hi, \color{ff0000}{how} are you?` is entered as plain text, the changes accepted and applied, the printed text should be: *Hi, how are you?*

- **Bold**: Bold style letters are defined using `\b{}` The bold style letters go between the curvy brackets.

For example, if `Descartes\b{JS}` is entered, the following is printed: *Descartes**JS***.

- **Itálicas**: Similar to the bold formatting, italic style letters are implemented using `Descartes\i{JS} \i{}`. The italic style letters go between the curvy brackets.
- **Mathematical expressions**: Mathematical expression go inside the curvy brackets of `\${}`.
  - **Superscripts**: The ^ character is used to format whatever character (or block of characters) follows it. If more than one character is to be placed as superscript, the block of characters should be flanked by curvy brackets (`{}`). If it consists of a single character, the brackets are not required.

    For example, to get A^{BC}, it is necessary to enter `\${A^{BC}}`.
  - **Subscripts**: Their behavior is practically the same as for superscripts. However, the underscore (_) symbol is used for subscripts as the ^ is used for superscripts.

    For example, to get A_{BC}, it is necessary to enter `\${A_{BC}}`.
  - **Fractions**: The `\frac{}{}` element is used inside a mathematical expression. The numerator is placed inside the first set of curvy brackets, the denominator in the second.

    For example, in order to display the $\frac{e^x}{b_2}$ fraction, it is necessary to enter `\${\frac{e^x}{b_2}}`.
  - **Roots**: The `\sqrt{}{}` code is used inside a mathematical expression. The index of the root is placed inside the first set of curvy brackets. The radicand is placed inside the second set. It is possible to leave the first set empty, in which case no index is displayed (the summarized version of the square root).

    For example, to get $\sqrt[3]{\sin(a)}$, it is necessary to enter `\${\sqrt{3}{sin(a)}}`.

    And to get $\sqrt{\sin(a)}$, `\${\sqrt{}{sin(a)}}` would have to be entered.
  - **Integrals**: The `\int{}{}{}` code is entered inside a mathematical expression to display integrals. The integral's lower limit is placed inside the first set of curvy brackets, the upper limit in the second set, and the integrand and variable of integration in the third.

    For example, entering `\${\int{a}{b}{x^2 dx}}` results in printing $\int_a^b x^2 dx$.

    If nothing is entered in the first and second sets of curvy brackets, an indefinite integral is printed.
  - **Sums**: The `\sum{}{}{}` code is used inside a math expression to print sums. The initial value of the index of the sum is placed inside the first set of curvy brackets, the upper value of the same inside the second set, and the argument of the sum inside the third set.

    For example, entering `\${\sum{i=1}{n}{i^2}}` results in printing $\sum_{i=1}^{n} i^2$.
  - **Products**: Its `\prod{}{}{}` code is similar to that of sums and integrals. The initial value of the index of the product is placed in the first set of curvy brackets, its upper value in the second set, and the argument of the product in the third set.

For example, entering `\${\prod{i=1}{n}{A_i}}` results in printing $\prod_{i=1}^{n} A_i$.

If the graphic text object is set to use *Serif* as its font, mathematical expressions will be displayed in italics by default, since that is the font typically used to display equations and other mathematical language.

- **Centered align**: The text to center is entered inside the curvy brackets of the `\c{}` code. If the *text width* parameter has a 1 value (i. e., there is no horizontal limit to the text), then the text is centered relative to the longest line present. Otherwise (if, say, *text width* is set to `500`), the text is centered relative to the whole 500 px indicated. The text width topic has more information of this parameter.
- **Right align**: Its functionality is similar to that of the centered text. The text is entered inside the curvy brackets of an `\r{}` expression.
- **Left align**: The text is entered inside the curvy brackets of an `\l{}` expression.
- **Justified align**: The text is entered inside the curvy brackets of an `\j{}` expression. The text is justified relative to the width, in px, entered in the *text width* parameter.

Various text editions can be nested one inside the other. For example, to print a text such as ***Hola***, in which the text is blue and the *o* is in bold, it is necessary to enter: `\color{0000FF}{H\b{o}la}`.

This type of edition is recommended for texts whose formats are not to be excessively complicated. For those that are, the rich text format discussed next is the suitable option.

15.2.2 Rich text format editor

When the *Rtf* button adjacent to a *text* field is clicked, a different text edition window is displayed. It has a lot of buttons near its top margin. These allow the inclusion of different types of text, as well as mathematical symbols. This window is displayed in Figure 15.2.2.1.

This window can be used to enter text as in the plain text one. However, it is possible to apply different formats to different parts of the text by selecting them and applying the formats afterwards. For instance, bold style, italics, a specific font type or size can be applied to different selection of the text.

Apart from the first controls, with which we are already familiar, there are more sofisticated ones:

- **underline button**: It has an underlined U, and applies an underline to the selected text.
- **overline button**: It has an overlined O, and applies an overline format to the selected text.
- **color button**: It has a color inside it. When clicked, the color editor dialog is launched, and the color there selected is applied to the selected text.

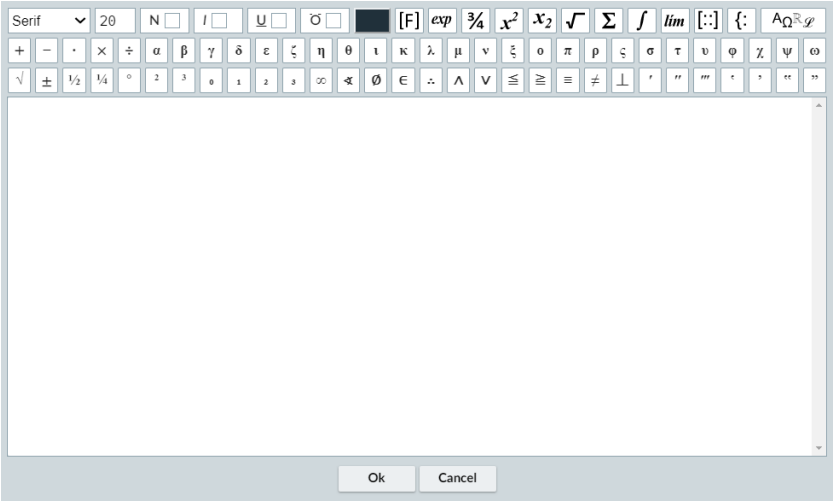


Figure 15.2.2.1: Rich text format entry window.

- **[F] button**: It inserts a formula box at the right of the cursors position.The box has the shape of a text field with a dotted border. Once added, the cursor can be placed inside it to enter mathematical expressions. The buttons following the [F] one in that line are used to enter a certain form (for instance, an expression to print a variable's value, or the $\frac{3}{4}$ to enter fractions). The form is entered inside the formula box.
  The *CTRL* + *f* keyboard shortcut in windows inserts a formula box as well (i. e., it does exactly the same as clicking the [F] button) when inside the rich text edition window.

- ***exp* button**: This button is used to enter *expressions*. Expressions can be constants, a variable's value, or a math expression calculating a value. When the text is printed in a scene, their value is printed. Using *expressions* in rich text format is analogous to flanking an expression between square brackets [] when using plain text. For example, some part of the program may have an `a=2+1` assignment. `a` could be entered as the expression's argument, and the resulting 3.00 value would be printed in the scene. To enter the expression to print, a double click is done on the yellow highlighted *expr* box entered when the *exp* button is clicked. This brings up the expression dialog. In the *valor* text field (*valor* is Spanish for *value*) field, the expression is entered. That dialog also has another field to specify how many decimals this expression is to show, and whether they are fixed or not. These specifications are particular to this expression only, and are not used generally for other values the text may print, as is the case when using the plain text editor. Expressions are housed inside a formula box. If the *exp* button is clicked while the cursor is not in a formula box, a formula box is then created where the cursor is, and an *expr* expression added inside it. Expressions may be placed in any placeholder inside a formula. For instance, an integral's upper limit may be an expression, so that this limit varies depending on certain conditions in the scene.

- **fraction button**: A button with a 3/4 text. When clicked a fraction form is entered at the cursor's position inside the formula box. If the cursor isnot in a formula box, one is created with the fraction inside it. This fraction form has a numerator and denominator placeholder.
  The keyboard shortcut to this form is *CTRL* + *7*.

- **superscript button**: This button has a x^2 text in it. It adds a base and exponent (superscript) placeholder at the cursors position. If such is outside a formula box, one is created on the fly.
  The corresponding keyboard shortcut is *CTRL* + ↑, where ↑ is the keyboard's upward pointing arrow.

- **subscript button**: This button has a x_2 text in it. It adds a placeholder for an expression and one for its subscript, at the position of the cursor. If the cursor is not inside a formula box, one is created on the fly.
  The corresponding keyboard shortcut is *CTRL* + ↓, where ↓ is the keyboard's downward pointing arrow.

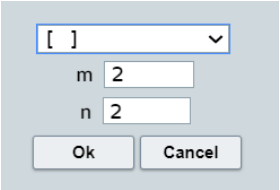
- **root button**: This button has a root's radical symbol in it. It adds the radical symbol at the cursor's position. If the cursor lies outside a formula box, one is added on the fly. The radical symbol added has a placeholder for the radicand and the index of the root.
  The corresponding keyboard shortcut is *CTRL* + *r*.

- **sum button**: This button has the greek sigma (Σ) character in it. When clicked, it adds a sum math symbol at the cursor's position. If the cursor is outside a formula box, one is created on the fly. The sum has three placeholders: one for the starting value of the sum's index, one for its upper value, and one for the sum's argument. These can be navigating using the keyboards four arrows.
  The corresponding keyboard shortcut is *CTRL* + *s*.

- **integral button**: This button has an integral calculus symbol ($\int$). It adds an integral at the cursor's position. If the cursor lies outside a formula box, one is created on the fly. Once added, the keyboard's arrow keys can be used to navigate the three placeholders in it: the integral's lower limit, its upper limit, and the integrand.

- **limit button**: This button has the word *lim* in it. When clicked a limit form is entered in the cursor's position. If the cursor lies outside a formula box, one is created on the fly. The keyboard's arrow keys can be used to navigate the placeholders: one for the limits variable, one for its tendency value, and one for the limit's argument.
  The corresponding keyboard shortcut is *CTRL* + *l*.

- **matrix button**: This button has a matrix symbol in it (square brackets housing four dots in a 2×2 disposition). When clicked, a matrix definition dialog is launched:



The menu in the dialog allows the developer to select the symbols flanking the entries of the matrix (square brackets, parentheses, curvy brackets, simple vertical bars, or double vertical bars can be selected). The *m* and *n* parameters correspond to the number of rows and columns, respectively. Once the matrix has been added, it appears at the position of the cursor in the text. If the cursor lies outside a formula box, one is created on the fly. The entries of the matrix have placeholders that can be navigated using the keyboard's arrow keys.

- **function definition by parts button**: This button has an opening curvy bracket followed by a couple of dots in it. When clicked, a dialog launches with a *parts* text field, in which the number of parts defining a function by parts is entered. Once accepted, the opening curvy bracket appears with one placeholder for each part of the defined function. The keyboard's arrow keys can be used to navigate the placeholders. The whole function definition is inserted at the cursor's position. If the cursor lies outside a formula box, one is created on the fly.
- **special characters panel**: Identical to its special characters button counterpart in the plain text editor. When clicked, a panel is displayed with special characters in it. They are grouped by categories, which are listed at the left hand side of the panel, and may be clicked to move form one category to another.

Be aware that the size control in the rich text edition window is a text field (as opposed to a menu), and it therefore allows for values between 8 and 200.

Besides the button already mentioned, there are two rows of buttons. All greek letter symbols are included in these, besides some useful math and text operators. When one of this is clicked, the symbol is added at the position of the cursor. If the *SHIFT* key is pressed down, these symbols switch to their upper case version (for the letter ones).

When text is added in a placeholder inside a math formula box, it is always possible to insert text to its left or right by moving the cursor to the desired place. Additionally, grouped placeholders (such as in the radical, fraction, or integral) can be simultaneously removed by placing the cursor at the right of the whole group and keying the backspace key; or by placing the cursor at the left of the whole group and keying the delete key. Precision is required if, instead of removing a whole group, only a single character is to be removed. The easiest way to achieve this is by placing the cursor at the right (or left) of the particular character by using the keyboard's arrow keys. Once there, the backspace (or delete key if the cursor is at the left of the character to remove) key is used and the single character is removed. Also, if a new line is to be entered after some math formula, it is first necessary to place the cursor at the right of the formula **outside** the formula box and then click the *ENTER* key to add a new line.

It is possible to combine different text forms to get complicated math expressions. For example, a continued fraction such as $\frac{1}{1+\frac{1}{1+...}}$ can be printed by first entering a fraction first, then entering the numerator and, after moving to the denominator placeholder, typing 1+ and clicking the fraction button again. Repeating this process, more and more iterations of the continued fraction can be included.

It is also possible to assign different styles inside a same box formula. This makes it possible to use different colors, or even fonts, inside a same formula.

When the *Ok* button is clicked, the rich text editor window closes and the configuration editor is displayed again. The *text* parameter in it diplays the edited text. It has some code in it, which is not necessarily what the user entered. This is the rich text code required for the text to have its given format. When the changes are applied, the printed text looks just like it did in the rich text format editor.

A brief exercise may put everything in perspective. This exercise's interactive scene, along with the instructions to build it, can be found at Text editor. The interactive scene's file as such can be found at this link. All these files are also stored in the *DescartesJSDocumentation.zip* file.

This exercise shows that more mathematically aesthetical formulas can be entered using the rich text editor, even though it may be more difficult to use than the plain text one. Additionally, it enables the developer to enter different expressions with different decimal conditions inside a same text graphic object.

15.3 The virtual keyboard

The native keyboard of most mobile devices takes up a considerable proportion of the screen, thus reducing the area reserved for the running application. *DescartesJS* has the option to use a custom made keyboard instead, which is displayed **inside** the scene itself, so as not to reduce the scene's size. This is the *DescartesJS virtual keyboard*.

The virtual keyboard is housed inside the scene, and the developer may choose its coordinates. It is only implemented when the keyboard checkbox of a text field supporting control is marked. Text field supporting controls are those whose values can be entered by means of a text field, such as the *textfield*, the *spinner*, the *scrollbar* and the *menu* controls.

There are many different keyboards available for selection. The choice should be made depending on which one best adheres to the control's desired functionality. For example if the user is to enter a purely numerical answer, the 7×2 or the 14×1 keyboards would be the ideal choices, since they have no operation symbol, thus discouraging the use of the *DescartesJS* internal calculator. The 4×4 keyboard includes basic operators such as the + and -. The 5×4 one includes also the $\times$ and / operators. If the user is to provide alphanumeric answers, the $10 \times 4\_alfa$ might be the best choice.

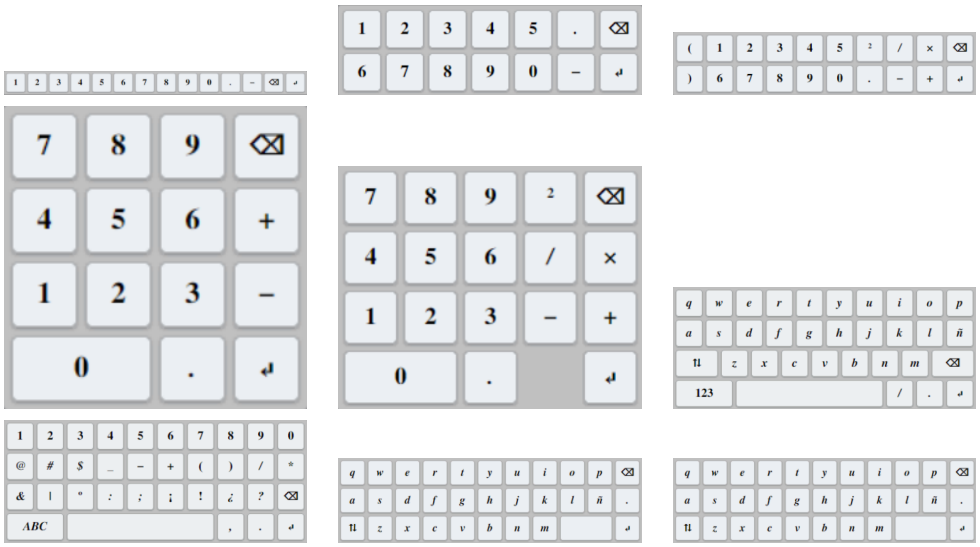


Figure 15.3.0.1: The various virtual keyboards available.

When a character or number key is selected in a virtual keyboard, its character is entered in the cursor's position inside the keyboard's text field. To exit the keyboard and accept the changes, the virtual keyboard's enter key has to be selected.

The virtual keyboard is only displayed when a user clicks on the text field of the control for which the keyboard is to be used. The keyboard is closed upon leaving the text field. When the keyboard is displayed, the remaining area of the scene not occupied by it turns slightly grayer. This is done so the user knows the only possible interaction is with the keyboard only (all other elements in the scene become inactive).

The keyboards have some hidden characters. For example, when a vowel remains pressed, the accentuated versions of the vowel are displayed on top of the keyboard. If the decimal point is pressed down and held, several symbols are displayed at the top as well. Since these extra characters are displayed above the keyboard, the keyboard should be placed at least 50 px below the space's top margin so the extra characters may be visualized. For example, `(100,50)` could be entered in the keyboard position parameter, so as to leave enough space for these characters to be displayed.

15.4 Keyboard shortcuts

15.4.1 Shortcuts to the configuration editor and its tabs

DescartesJS has several keyboard shortcuts to access its most frequently used elements. These are listed in the following list. The *CONTROL* key is abbreviated as *CTRL* in their description. The + symbol indicates two keys are pressed simultaneously.

Shortcuts used when in the *DescartesJS* main editor.

- **CTRL+E**: If the *CONTROL* key and the *E* one are clicked simultaneously, the configuration editor is launched.

Shortcuts used when inside the *DescartesJS* configuration editor.

- **CTRL+1**: This combination displays the Scene tab.
- **CTRL+2**: This combination displays the Spaces tab.
- **CTRL+3**: This combination displays the Controls tab.
- **CTRL+4**: This combination displays the Definitions tab.
- **CTRL+5**: This combination displays the Programs tab.
- **CTRL+6**: This combination displays the 2D Graphics tab.
- **CTRL+7**: This combination displays the 3D graphics tab.
- **CTRL+8**: This combination displays the Animation tab.
- **CTRL+ENTER**: This combination applies the changes, just like when the Apply button of the scene's configuration editor is clicked.
- **CTRL+ALT+INTRO**: This combination accepts the changes, just like when the Ok button of the configuration editor is clicked.
- **CTRL+BACKSPACE**: This combination closes the configuration editor, just like when the Close button of the configuration editor is clicked, ignoring any changes.

15.4.2 Listed elements navigation shortcuts

The *Spaces*, *Controls*, *Definitions*, *Programs*, *Graphics* y *3D Graphics* tabs of the configuration editor each have a panel at their left hand side, where the various elements of that tab are listed. When this panel is selected in one of the tabs, it displays a dark blue border around it, **and only then** do certain keyboard shortcuts become available.

When the panel is selected and there is more than one element listed in it, the keyboard's up and down arrows (↑ and ↓) can be used to move the selection to the element above or below, respectively.

If an element is selected in a list with multiple elements, the *CTRL* + ↑ moves the selected element one place upwards in the list (if it is not already at the top), and the *CTRL* + ↓ moves it one place downwards (if it is not at the bottom).

These shortcuts allow for a better and more agile organization and visualization of the elements in a list, and are especially useful for developers who favor the keyboard over the mouse.