CONSTRUCT 2 JAVASCRIPT SDK DOCUMENTATION

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk

---

Developers can extend Construct 2 with their own plugins and behaviors using the Javascript SDK. This manual documents how to use the SDK and the features Construct 2 exposes through the plugin interface.

This is a technical manual for javascript programmers. If you're looking for help on how to use Construct 2, please see the Construct 2 manual.

Familiarity with Construct 2 is recommended before developing with the SDK. The terminology and functions may be hard to understand otherwise. The beginner's guide is a good place for developers to start learning how Construct 2 works.

Download the SDK template to get started. The download includes a template plugin, behavior and effect which serves as a useful starting point for developing your own Construct 2 addons.

Uses for the Javascript SDK

The Javascript SDK allows you to integrate your own Javascript code in to your Construct 2 games. This is especially useful for integrating Construct 2 games with your own or third-party web-based services or backends, such as your own login and high-score systems, or to integrate a third-party advertising or payment solution. In addition to that, you can create your own new features in Construct 2 tailored to your specific game by writing some of the logic in Javascript, or expose brand new or platform-specific features to the Construct 2 event system.

Developer mode for previewing

By default Construct 2 only loads runtime scripts when previewing a project for the first time. Closing and reopening a project will cause Construct 2 to re-load the runtime scripts for all plugins. However, you can also set Construct 2 in to 'developer mode' which causes it to re-load plugin runtime scripts every time you press preview. This can save time during development since you can edit scripts while keeping a project open. To set developer mode, run regedit and open the following registry key (create it if it doesn't exist):

HKEY_CURRENT_USER\Software\Scirra\Construct2\html5

and add the key devmode and set it to 1 (DWORD value). Note this does not affect edittime scripts - these are only ever loaded the first time the editor starts up, so to reload them you must still close and reopen Construct 2.

JAVASCRIPT SDK FOR CONSTRUCT 2

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation

---

This section covers what you need to know about using the Javascript SDK for Construct 2.

OVERVIEW OF THE CONSTRUCT 2 SDK

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/overview

---

Third party developers can write their own plugins and behaviors for Construct 2 in javascript. Plugins have two parts: the editor side (defining the plugin settings, actions and conditions, etc) and the runtime side. The editor side is interpreted by Google's V8 javascript engine built in to the HTML5 exporter. The runtime side runs in the browser. Note the implications: you cannot use browser features in the edittime, and you should not use browser-specific features in the runtime.

Javascript is not Java! Java is an application programming language developed by Sun (which, confusingly, can also run in browsers via a plugin). Javascript is the native programming language for web pages in browsers. Make sure you're clear on the difference.

You do not need any special tools to develop plugins or behaviors. All you need is a text editor and a little javascript knowledge. A good text editor with syntax highlighting for javascript is Notepad++, which is favoured by Scirra developers.

This guide will not teach you javascript. Generally, we ask that you do not post questions about the javascript language itself to the Scirra forums. There are many other better places to ask on the web. Questions about the SDK are always welcome, though. Some useful resources for javascript are:

- Mozilla's Javascript Guide A complete guide to javascript. This might be a good starting point if you are new to programming.
- StackOverflow An excellent Q&A website. Also a good place to search to see if your question has already been asked and answered.
- Javascript Garden Guide to the unusual parts of javascript. Very useful if you have experience with a different programming language but are new to javascript.
- Mozilla Developer Network (MDN) An excellent reference for HTML, javascript, and more. A very useful place to look up features for the browser side of the plugin.

Overview of plugins and behaviors

Before developing a plugin or behavior, you should be familiar with the usage of Construct 2, or the terminology and functionality will be difficult to understand. The Construct 2 beginner's guide is a good place to start.

As you probably know, plugins define new objects in the editor, and behaviors add functionality other objects. Plugins and behaviors are surprisingly similar. Behaviors are essentially also plugins, but with the aim of affecting another object. The SDKs for both are very similar, so this guide will simply describe how plugins are made, and note where behaviors are different.

Plugin scripts

Plugin and behavior scripts are located at

<install path>\exporters\html5\plugins
<install path>\exporters\html5\behaviors

Each plugin has its own folder. Plugins consist of four files:

- common.js - this is prepended to both edittime.js and runtime.js in case you have code common to both.
- edittime.js - defines the plugin for the editor, including all its actions, conditions and expressions.
- runtime.js - defines the plugin functionality in the browser.
- PluginIcon.ico - the editor loads this icon to represent the plugin.

A template for both a plugin and behavior can be downloaded here. You can copy these to a folder in the above directories to provide a skeleton starting point for your plugin or behavior.

Plugins work identically in the 32-bit and 64-bit versions of Construct 2.

All the "built-in" plugins and behaviors are also written this way. It would be useful to read their scripts - also in the above directories - to see how their features are implemented. You can learn a lot from this.

All the scripts for the entire javascript runtime are also located in <install path>\exporters\html5. They are perfectly readable and commented (not minified). This is a great way to learn even more, but may make for some heavy reading. You may find these three scripts particularly relevant, though:

- common_prelude.js - prepended to both edittime scripts and runtime scripts.
- edittime_prelude.js - prepended only to edittime scripts.
- preview_prelude.js - prepended only to runtime scripts.

Remember that javascript has no way of limiting access to objects. This means you can alter any part of the runtime at any time. You should assume this is a bad idea (with undefined consequences) unless the object properties have been explicitly documented with valid ways of accessing the property. The runtime is complex, and any undocumented changes you make may break projects in subtle ways or have other unintended consequences. Keep the reference handy!

Good luck!

Developing your own plugins and behaviors is relatively straightforward and can be fun! The following guide pages describe plugin development in more detail. If you have any questions about the SDK feel free to ask on the forum, but please remember we ask that you look to other resources for questions on the javascript language itself.

PLUGIN SETTINGS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/plugin-settings

---

At the top of edittime.js is the function GetPluginSettings() which tells Construct 2 some important information about the plugin or behavior. Here is the settings function for Sprite:

```
function GetPluginSettings()
{
 return {
  "name":   "Sprite",
  "id":   "Sprite",
  "version":  "1.0",
  "description": "An animated object that is the building block of most projects.",
  "author":  "Scirra",
  "help url":  "http://www.scirra.com",
  "category":  "General",
  "type":   "world",
  "rotatable": true,
  "flags":  pf_animations | pf_position_aces | pf_size_aces | pf_angle_aces | pf_appearar
 };
};
```

Each field is as follows:

name

This is the name of your plugin as it appears in the dialogs in Construct 2. Note it is separate to the id.

id

This is a string identifying your plugin. All plugins must have a unique id. The id, not the name, is saved in the project XML to identify a plugin. This means you can safely change the plugin's name without breaking existing projects. However, if you change the id, Construct 2 will consider it to be a different kind of plugin, and all existing projects using the plugin will no longer load. Therefore, you should choose an appropriate id when starting development of a new plugin, and never change it.

version

This is a float in the format x.y which identifies the version of your plugin. You should keep this updated whenever you make a new release. Construct 2 uses it

to verify projects are compatible when opening. For example, Construct 2 will show a warning if a project was saved with version 2 of a plugin, but is opened with version 1 of the plugin installed.

description

Some text describing the purpose of the plugin. This is displayed in the dialog when choosing a plugin.

author

You or your organisation.

help url

When the user clicks 'help' in the editor for your plugin, this is the URL they are sent to.

category

In the editor dialogs, all plugins and behaviors are grouped in to categories. This specifies which category your plugin belongs to. It is advisable to use an existing category whenever relevant, but you can set this to anything you like and Construct 2 will put it in its own category. The category is case sensitive.

type *(not used in behaviors)*

This can take one of the following values, depending on what kind of plugin you want to make:

- "world" The plugin appears in the layout, and therefore draws something to the screen (e.g. Sprite, Tiled Background, Text).
- "object" The plugin does not appear in the layout, and therefore does not draw anything (e.g. Array). The draw methods will not be called, and the user cannot place the object in the layout - they must access it via the Object Bar or Project Bar.

rotatable *(not used in behaviors)*

When the type is *"object"*, this setting is ignored. When the type is *"world"*, this specifies if the object has an angle. The user may also rotate the object in the layout editor. For example, Sprite is rotatable, but Text is not.

flags

Flags describing additional settings. These can be combined with bitwise OR (e.g. pf_position_aces | pf_size_aces), or set to 0 for no flags. The following flags are available for plugins:

- pf_singleglobal Specifies a *single global* type of plugin. When inserted, these are available project-wide, and there is only ever one instance of the object

(additional instances cannot be created). This is ideal for input objects or other non-object based features, e.g. Mouse, Keyboard, Audio. pf_singleglobal cannot be used with *"world"* type plugins.

- pf_texture The plugin uses a single texture. Tiled Background uses this flag. Construct 2 will open the image editor when inserting the plugin.
- pf_animations The plugin uses Construct 2's animation system. Sprite uses this flag. Construct 2 will open the animations editor when inserting the plugin.
- pf_tiling Only valid when pf_texture or pf_animations is also used. Specifies that the plugin will tile its texture. This alters the image editor's functionality to better suit tiled textures. Tiled Background uses this flag.
- pf_position_aces Only valid with *"world"* type plugins. Automatically inherit actions, conditions and expressions for the object position (such as Set X and Set Y).
- pf_size_aces Only valid with *"world"* type plugins. Automatically inherit actions, conditions and expressions for the object size (such as Set Width and Set Height).
- pf_appearance_aces Only valid with *"world"* type plugins. Automatically inherit actions, conditions and expressions for the object appearance (such as Set Visible and Set Opacity).
- pf_zorder_aces Only valid with *"world"* type plugins. Automatically inherit actions, conditions and expressions for the object Z order (such as Set Layer and Move To Front).

The following flags are available for behaviors:

- bf_onlyone The behavior can only be added to an object once. Normally the user can add a behavior as many times as they like, but this flag prevents them from adding it again. For example, the Solid behavior uses this flag, because it does not make sense for an object to have two Solid behaviors.

dependency

This one isn't listed above, but if you need external files bundled with your plugin (e.g. a javascript library) you can specify one or more dependency files with:

dependency": "file1.js;file2.js;file3.html

You must provide these files in the plugin's folder. Construct 2 will then copy them out when exporting the project, and make them available on the preview server for testing. Construct 2 will also automatically insert a script tag in to the HTML page before the runtime for any files ending in .js, so you do not need to worry about loading them yourself.

Changes after publishing

You should not change the id, type, rotatable or flags settings after releasing your plugin or behavior (other than to add new 'aces' flags), as this will break all existing projects using it. All the other settings can be changed at any time.

ACTIONS, CONDITIONS AND EXPRESSIONS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/actions-conditions-expressi

---

These are often called ACEs for short, or ACE to refer to 'an action, condition or expression'. In your edittime.js, you can specify any ACEs your plugin or behavior uses. Remember behavior ACEs are merged with the object's ACEs in the editor dialog.

Parameters

If your ACE takes parameters, you must first add them using the following functions. The *name* and *description* parameters are required for all parameters. The *name* appears as a label to the left of the parameter. The *description* appears at the top of the dialog when the parameter has focus. The description is actualy very important: you can save the user a trip to the manual by including some vital information, like the units being used (e.g. pixels vs. texture coordinates). Try to include anything the user might wonder about the parameter in the description. Hopefully you'll have found descriptions useful in using Construct 2 - try to do the same with your parameter descriptions!

Some parameters also take an *initial_str*. This is an optional string (always a string, even for number parameters) that is pasted in to the parameter as a default. Try to set a common or useful default, e.g. "100" for a percentage. This can also help save the user time and also indicate what a reasonable entry is. Never let the default give a syntax error! If not provided, it is 0 for number/any type parameters, and "" (an empty string) for strings.

All parameters also take an optional *id* number parameter at the end. This should not normally be used, but can allow you to modify parameters after releasing your plugin. All parameter ids are by default their index, and you can maintain compatibility with existing projects by explicitly assigning ids to match old parameters to new ones.

Parameters are all passed to the corresponding runtime function as ordinary javascript values. They are passed in the same order as they are added with these functions.

AddNumberParam(name, description, initial_str);

A typed number parameter, which accepts integers or floats.

AddStringParam(name, description, initial_str);

A typed string parameter.

AddAnyTypeParam(name, description, initial_str);

A typed parameter that accepts either integers, floats or strings. Wherever possible, prefer number or string parameters, since they force the correct type to be entered, reducing the chance of a mistake. The runtime function must check the actual type passed using javascript's *typeof* operator.

AddCmpParam(name, description);

A combo box parameter with the following standard items: Equal, Not equal, Less, Less or equal, Greater, Greater or equal.

AddComboParamOption(text);
AddComboParam(name, description, initial);

A combo box parameter with custom items. Call *AddComboParamOption* once per item before *AddComboParam* to set the items. Note *initial* specifies the integer index of the default item, rather than a string like other parameters. The runtime function receives the integer index of the chosen item.

AddObjectParam(name, description);

A button that allows the user to pick an object type in the project. The runtime function receives a reference to the object type in the runtime. Construct 2 may also pass *null* for this parameter, so always check the parameter is not null before using it.

AddLayerParam(name, description);

A typed parameter where the user can enter a layer's name (string), or number (0-based index). The runtime receives a reference to the layer in the runtime. Construct 2 may also pass *null* for this parameter so always check the parameter is valid before using it.

AddLayoutParam(name, description);

A combo box with all the layouts in the project. The runtime receives a reference to the layout in the runtime. Construct 2 may also pass *null* for this parameter so always check the parameter is valid before using it.

AddKeybParam(name, description);

A button the user can click and hit a key. The runtime receives the virtual key (VK) code.

AddAnimationParam(name, description, initial_str);

A typed string parameter where the user can enter the name of one of the object's animations. Only valid when the plugin specifies *pf_animations*.

Construct 2 may also pass null for this parameter so always check the parameter is not null before using it.

AddAudioFileParam(name, description);

A combo box with all the audio files in the project. The runtime receives a string of the filename without the extension. For example, if *MyFile.ogg* is chosen, then *"MyFile"* is passed. If the browser supports Vorbis, you should append .ogg; if not, append .m4a. The .m4a file is not guaranteed to exist: Construct 2 considers .ogg files to be the project's audio files, and .m4a are only used as a backup for .ogg files if the browser does not support Vorbis. However, the user may not have created a .m4a backup file.

AddCondition(id, flags, list_name, category, display_string, description, script_name);

id

A number uniquely identifying this condition. This is saved to the project XML, so you may change the rest of the parameters after releasing your plugin, but not the id.

flags

This may be zero for no flags, or a combination of the following values combined with bitwise OR (|):

cf_trigger

A triggered condition. Events with triggers are not evaluated every tick; instead, they are run by the runtime.trigger() function. There are some limitations on triggers: an event cannot contain two triggered conditions, and a trigger is always the first condition.

cf_fake_trigger

Appears and works exactly like a trigger in the editor, but is passively evaluated every tick in the runtime like an ordinary condition. You cannot specify both *cf_trigger* and *cf_fake_trigger*. Fake triggers are useful for conditions like "Every X milliseconds", where the condition should work like a trigger in the editor, but it is most convenient to implement it as an ordinary condition in the runtime. Fake triggers cannot be triggered by runtime.trigger(): they are identical to ordinary events in the runtime. This flag only affects the editor.

cf_static

Normally, the condition function at runtime is called once per instance, to 'filter' the picked objects. Static conditions (those specifying this flag) only have their function called once, no matter how many instances there are. You must then

perform any picking yourself in the function. For example, 'Pick random instance' would be most conveniently implemented as a static condition.

cf_not_invertible

Prevents the user inverting the condition. This is useful if the inverted condition does not make sense, e.g. 'Pick random instance'.

cf_deprecated

Hides the condition from the dialog. If you wish to replace a condition in your plugin with different features, you must not remove it entirely: this will break all existing projects using it. Instead, marking it deprecated prevents any new projects from using it, while letting old projects load correctly and continue using it.

cf_incompatible_with_triggers

Prevents the user adding the condition to an event with a trigger in it. This is not generally useful for plugins - the runtime uses it for certain special conditions like 'trigger once'.

cf_looping

Shows a looping icon next to the condition. This does not affect the functionality of the condition in any way - it is purely cosmetic. Construct 2 will assume your condition function is implemented as a looping condition.

list_name

This is the name of the condition in the *Add condition* dialog, e.g. "Compare X".

category

The category the condition belongs to in the dialog. This may be empty for behavior conditions, where the category name will be the name of the behavior. It must not be empty for plugin conditions.

display_string

The string displayed in the event sheet view. {0}, {1}, {2} etc. are substituted with the corresponding parameter. These can each only appear once in the display string. You can also use bold and italic HTML tags (but no other HTML is valid). Try to follow conventions in the rest of Construct 2's plugins and behaviors when using bold and italic.

description

A string appearing at the top of the *Add condition* dialog when the user has selected the condition. As with parameter descriptions, try to be as helpful as possible for the user with this.

script_name

The name of the condition function in the runtime script. For example, if it is "MyCondition", the runtime function must be cnds.MyCondition. Since dot syntax must be used for properties (see *Google Closure Compiler compatibility*), make sure this does not contain any spaces etc.

Adding conditions

By convention, the plugin's conditions are listed first. Once any parameters have been added using the above functions, the following function adds a condition:

AddCondition(id, flags, list_name, category, display_string, description, script_name);

id

A number uniquely identifying this condition. This is saved to the project XML, so you may change the rest of the parameters after releasing your plugin, but not the id.

flags

This may be zero for no flags, or a combination of the following values combined with bitwise OR (|):

cf_trigger

A triggered condition. Events with triggers are not evaluated every tick; instead, they are run by the runtime.trigger() function. There are some limitations on triggers: an event cannot contain two triggered conditions, and a trigger is always the first condition.

cf_fake_trigger

Appears and works exactly like a trigger in the editor, but is passively evaluated every tick in the runtime like an ordinary condition. You cannot specify both *cf_trigger* and *cf_fake_trigger*. Fake triggers are useful for conditions like "Every X milliseconds", where the condition should work like a trigger in the editor, but it is most convenient to implement it as an ordinary condition in the runtime. Fake triggers cannot be triggered by runtime.trigger(): they are identical to ordinary events in the runtime. This flag only affects the editor.

cf_static

Normally, the condition function at runtime is called once per instance, to 'filter' the picked objects. Static conditions (those specifying this flag) only have their function called once, no matter how many instances there are. You must then perform any picking yourself in the function. For example, 'Pick random instance' would be most conveniently implemented as a static condition.

cf_not_invertible

Prevents the user inverting the condition. This is useful if the inverted condition does not make sense, e.g. 'Pick random instance'.

cf_deprecated

Hides the condition from the dialog. If you wish to replace a condition in your plugin with different features, you must not remove it entirely: this will break all existing projects using it. Instead, marking it deprecated prevents any new projects from using it, while letting old projects load correctly and continue using it.

cf_incompatible_with_triggers

Prevents the user adding the condition to an event with a trigger in it. This is not generally useful for plugins - the runtime uses it for certain special conditions like 'trigger once'.

cf_looping

Shows a looping icon next to the condition. This does not affect the functionality of the condition in any way - it is purely cosmetic. Construct 2 will assume your condition function is implemented as a looping condition.

list_name

This is the name of the condition in the *Add condition* dialog, e.g. "Compare X".

category

The category the condition belongs to in the dialog. This may be empty for behavior conditions, where the category name will be the name of the behavior. It must not be empty for plugin conditions.

display_string

The string displayed in the event sheet view. {0}, {1}, {2} etc. are substituted with

the corresponding parameter. These can each only appear once in the display string. You can also use bold and italic HTML tags (but no other HTML is valid). Try to follow conventions in the rest of Construct 2's plugins and behaviors when using bold and italic.

description

A string appearing at the top of the *Add condition* dialog when the user has selected the condition. As with parameter descriptions, try to be as helpful as possible for the user with this.

script_name

The name of the condition function in the runtime script. For example, if it is "MyCondition", the runtime function must be cnds.MyCondition. Since dot syntax must be used for properties (see *Google Closure Compiler compatibility*), make sure this does not contain any spaces etc.

Adding actions

By convention, the plugin's actions are listed second. Once any parameters have been added, the following function adds an action:

AddAction(id, flags, list_name, category, display_string, description, script_name);

id

A number uniquely identifying this action. This is saved to the project XML, so you may change the rest of the parameters after releasing your plugin, but not the id.

flags

This may be zero for no flags, or:

af_deprecated

Hides the action from the dialog. If you wish to replace an action in your plugin with different features, you must not remove it entirely: this will break all existing projects using it. Instead, marking it deprecated prevents any new projects from using it, while letting old projects load correctly and continue using it.

list_name

This is the name of the action in the *Add action* dialog, e.g. "Set X".

category

The category the action belongs to in the dialog. This may be empty for behavior actions, where the category name will be the name of the behavior. It must not

be empty for plugin actions.

display_string

The string displayed in the event sheet view. {0}, {1}, {2} etc. are substituted with the corresponding parameter. These can each only appear once in the display string. You can also use bold and italic HTML tags (but no other HTML is valid). Try to follow conventions in the rest of Construct 2's plugins and behaviors when using bold and italic.

description

A string appearing at the top of the *Add action* dialog when the user has selected the condition. As with parameter descriptions, try to be as helpful as possible for the user with this.

script_name

The name of the action function in the runtime script. For example, if it is "MyAction", the runtime function must be acts.MyAction. Since dot syntax must be used for properties (see *Google Closure Compiler compatibility*), make sure this does not contain any spaces etc.

Adding expressions

By convention, the plugin's expressions are listed third. Expressions can only use number, string or 'any type' parameters, since expression parameters are all entered as text. Once any parameters have been added, the following function adds an expression:

AddExpression(id, flags, list_name, category, expression_name, description);

id

A number uniquely identifying this action. This is saved to the project XML. After releasing your plugin you cannot change the id or expression_name, but the other parameters can be changed.

flags

This may not be zero - at least one flag specifying the return type must be set. The available flags are:

ef_return_number

The expression returns either an integer or float, using the runtime ret.set_int() or ret.set_float() functions.

ef_return_string

The expression returns a string, using the runtime ret.set_string() function.

ef_return_any

The expression can return either an integer, float or string. Either *ef_return_number* or *ef_return_string* should be preferred where possible, since Construct 2's expression parser cannot verify the correct types are used when *ef_return_any* expressions are involved.

ef_variadic_parameters

Construct 2 will allow the expression to be used with additional 'any type' parameters past the end of those specified. Any parameters that are specified will still be required and type checked. If no parameters are specified, the expression can be used with any number of parameters at all.

ef_deprecated

Hides the expression from the object panel and autocomplete, and raises a syntax error if the user tries to type it in. If you wish to replace an expression in your plugin with different features, you must not remove it entirely: this will break all existing projects using it. Instead, marking it deprecated prevents any new projects from using it, while letting old projects load correctly and continue using it.

list_name

This is intended to be equivalent to the condition and action list_name. However, in the current release of Construct 2, it is not used. It may be displayed in a future release though, so it should still be set appropriately.

category

The category the expression belongs to in the dialog. This may be empty for behavior expressions, where the category name will be the name of the behavior. It must not be empty for plugin expressions.

expression_name

The name of the expression. This follows the dot in the expression syntax. For plugins, the expression is used as MyObject.expression_name. For behaviors, it is used as MyObject.MyBehavior.expression_name. It also serves as the script name: if it is "MyExpression", the runtime function must be exps.MyExpression. Since dot syntax must be used for properties (see *Google Closure Compiler compatibility*), make sure this does not contain any spaces etc.

description

The description that appears next to the expression in the Expressions Panel. As with parameter descriptions, try to be as helpful as possible for the user with this.

Finishing up

Once all your ACEs are added, call the following function:

ACESDone();

You cannot add any more ACEs after this line.

Implementing the runtime functions

Each ACE must have a corresponding function in runtime.js. These functions must be added to the prototypes of the Cnds, Acts and Exps objects. Despite the fact the functions are added to an empty object's prototype, they are invoked with this referencing an instance of your plugin.

Conditions are declared like so:

```
Cnds.prototype.ScriptName = function (params)
{
 return true;
};
```

params is a list of function parameters corresponding to any parameters to added to the condition. It can be empty (i.e. *function ()*) if no parameters were added. The function must return true if the condition was true for this instance, or false if not.

Actions are declared like so:

```
Acts.prototype.ScriptName = function (params)
{
 // do something
};
```

params is the same as with conditions. Actions do not need to return anything.

Expressions are declared like so:

```
Exps.prototype.ExpressionName = function(ret[, params])
{
 ret.set_int(0);
 // or:
 // ret.set_float
 // ret.set_string
 // ret.set_any
```

```
};
```

Note that the expression name is used here rather than a script name. *ret* must always be the first parameter, and is required even if the expression has no parameters. Any parameters the expression uses must follow *ret*. The expression's return value must not be returned by returning from the javascript function (e.g. *return 0;*) - values returned this way are ignored. Instead, call ret.set_int, ret.set_float, ret.set_string or ret.set_any to return a value from the expression. ret.set_any determines the type of the javascript value passed to it, and sets the return type based on that.

Remember the *ret* parameter is required and must be used to return values from the expression! This is different to writing ordinary javascript functions.

PROPERTIES

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/properties

---

Your plugin properties specify what appears in the properties bar when your plugin is selected. Construct 2 adds its own properties for most plugins. However, you can specify custom properties which are shown at the bottom of the properties bar. For behaviors, properties appear in a sub-category of the behaviors category.

Properties are specified in edittime.js. By convention, they follow the ACE definitions.

```
var property_list = [
 // a list of cr.Property objects
 ];
```

Each property should be a new cr.Property object:

new cr.Property(type, name, initial_value, description[, param, readonly])
type

One of the following property types:

ept_integer

An integer number. Floating point numbers cannot be entered to integer properties.

ept_float

A floating point number.

ept_text

A text property.

ept_color

A color selector. The runtime receives an rgb string, e.g. "rgb(255,255,255)".

ept_font

A font selector. The runtime receives a string formatted "facename,size,weight,italic".

ept_combo

A combo box property. Items are specified in the *param* parameter of cr.Property as a pipe-separated string, e.g. "One|Two|Three".

ept_link

A link property. Links do not have any associated value nor are they passed to the runtime; they simply allow you to do something in OnPropertyChanged() when it is clicked.

ept_section

Creates a new header in the properties bar. Useful for splitting up long property lists in to groups, like with the Particles object.

name

The name of the property.

initial_value

The initial value of the property. This must be a javascript number for integer and float properties; a javascript string for text properties; an RGB value for color properties (e.g. cr.RGB(255, 255, 255)); a string in the format "Facename,size" for font properties; the string of the default item to select for combo properties; and the link text for link properties.

description

The text that appears as a tip at the bottom of the properties bar. Try to keep it brief, but as helpful as possible to the user. Any opportunity to save the user a trip to the manual is worth taking.

param *(optional)*

For combo properties, a pipe-separated string specifying the combo box items, e.g. "One|Two|Three". For link parameters, this can be one of the following values:

"firstonly"

By default, clicking a link calls OnPropertyChanged() once for each of the selected instances. If you are performing an action on the object type, such as

invoking the image editor, specifying *"firstonly"* calls OnPropertyChanged() once only for the first selected instance rather than repeatedly.

"worldundo"

Create a 'world' undo point before calling OnPropertyChanged(). This allows undoing any change in position, size or angle. Sprite uses this so 'Make 1:1' can be undone.

readonly *(optional)*

Set to true to make the property read-only (uneditable).

Getting property values at runtime

In your instance's onCreate() function in runtime.js, properties are available via the array this.properties[]. This is an array of the property values. The values are in the same order as the properties were added, excluding link properties. For example, if you have two link properties followed by three integer properties, this.properties only has three elements (the three integer properties in the order they were added).

THE EDIT-TIME

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/edit-time

---

You cannot use browser features in the edit-time, since the edittime.js script is interpreted by Google's V8 javascript engine rather than a real browser. However, Construct 2 exposes some editor features in the edittime script. The functions and objects you can use are documented here.

Global functions

alert(msg)

Bring up a message box with the message. Generally only useful for testing or errors.

assert2(cnd, msg)

Assert that cnd is true, else cause a check failure with msg. This is only used in checked builds. However, it is useful for testing and diagnostics.

#Edittime callbacks#

CreateIDEObjectType()

Called whenever Construct 2 needs to create a new object type from your plugin.

IDEObjectType()

The class representing an object type in the editor.

IDEObjectType.CreateInstance()

Called whenever Construct 2 needs to create a new instance from your plugin.

IDEInstance()

The class representing an instance in the editor.

IDEInstance.OnInserted()

Called whenever the user manually inserts a new instance in to the project. Typically this is done via the Insert Object dialog.

IDEInstance.OnDoubleClicked()

Called when an instance is double-clicked in the layout. You may want to invoke

the texture or animation editor here.

IDEInstance.OnPropertyChanged(name)

Called just after one of the plugin properties has been changed for this instance. For link properties, this is where you can perform your link click action. You can also clamp or adjust the property value here (via this.properties[name]), and the property will update accordingly.

IDEInstance.OnRendererInit(renderer)

Called when a layout is opened containing instances of your plugin. You should load any textures or fonts here.

IDEInstance.OnRendererReleased(renderer)

Called when a layout containing instances of your plugin is closed. You should release any textures or fonts here.

IDEInstance.Draw(renderer)

Draw the object in the editor, if it is a *"world"* plugin type. Otherwise, Draw() is not called.

The edittime instance

The IDEInstance object has an *instance* member through which you can access information about the instance in the editor. It supports these methods:

instance.SetSize(size)

Set the object's width and height according to the cr.vector2 passed to size.

instance.GetSize()

Return a cr.vector2 with the width and height of the object.

instance.SetHotspot(p)

Set the hotspot according to the cr.vector2 point *p*. Hotspots are specified in texture coordinates, where (0,0) is the top left corner of the object, and (1,1) the bottom right, and (0.5, 0.5) in the middle.

instance.GetBoundingRect()

Return a cr.rect object specifying the object's axis-aligned bounding rectangle.

instance.GetBoundingQuad()

Return a cr.quad object specifying the object's bounding quad.

instance.EditTexture()

Either *pf_texture* or *pf_animations* must be specified. Invokes the image editor or animation editor respectively.

instance.GetTexture()

Return an object representing the object's current texture. If *pf_animations* is specified, this is a texture representing the first frame in the first animation.

instance.GetOpacity()

Return the object's current opacity, if it has one.

Edittime rendering

The OnRendererInit, Draw and OnRendererReleased functions pass a renderer object as a parameter. It supports these methods:

renderer.Quad(q [, opacity, uv])

Render the quad *q* (a cr.quad) with the current texture. *opacity* is optional, and *uv* can be a cr.rect specifying the texture coordinates to draw.

renderer.Line(a, b, color)

Render a line with the current texture from *a* to *b* (both cr.vector2 objects) with the specified color (use *cr.RGB(r, g, b)* to generate a color).

renderer.Fill(q, color)

Fill the quad *q* (a cr.quad) with solid color (use *cr.RGB(r, g, b)* to generate a color).

renderer.Outline(q, color)

Draw four lines outlining the quad *q* (a cr.quad) with the given color (use *cr.RGB(r, g, b)* to generate a color).

renderer.CreateFont(face_name, face_size, bold, italic)

Create a font object with the given parameters. Font objects support the method:

font.DrawText(text, rc, color, halign)

where *text* is the text to draw, *rc* is a cr.rect of the rectangle to draw the text inside, *color* is the text color, and *halign* is one of ha_left, ha_center or ha_right.

renderer.ReleaseFont(font)

Release a previously created font.

renderer.LoadTexture(texture)

Load the object's texture. Use this in OnRendererInit(). Pass the object's texture

from instance.GetTexture().

renderer.SetTexture(texture)

Set the texture currently drawing with.

renderer.ReleaseTexture(texture)

Release a previously loaded texture. Use this in OnRendererReleased().

renderer.EnableTiling(enable)

Pass *true* to enable tiling textures, or *false* to disable. Tiling allows uv coordinates greater than 1, causing the texture to repeat.

Texture objects also support the following methods:

texture.GetImageSize()

Return a cr.vector2 representing the texture's width and height in pixels (regardless of the object's current size).

texture.GetFilename()

Return the current active filename of the texture. This may be in a temporary folder if the texture has been edited but not saved.

texture.GetID()

Return an arbitrary string identifying this texture.

RUNTIME OVERVIEW

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/runtime-overview

---

Unlike the edittime, the runtime runs in a browser. This means you have access to all browser technologies, ranging from WebSockets and AJAX to the Web Audio API. Exciting stuff!

Strict mode

Runtime scripts must conform to the ECMAScript 5 "strict" mode. This helps reduce bugs and allow javascript engines to run the script faster. To find out more about Strict mode, see this blog post by John Resig.

Object recycling

To reduce garbage collector overhead, Construct 2 recycles the javascript objects for instances when objects are created and destroyed at runtime. For more information, see onDestroy() in the next section.

jQuery

Construct 2's javascript runtime includes jQuery. All plugins and behaviors may assume the presence of jQuery 1.6.3 (or higher, if it is updated in future builds) when they are running on a browser-based platform. Note jQuery is not included with non-browser platforms, which include CocoonJS and directCanvas.

It is preferable to access jQuery via the full name (jQuery.foo()) rather than the short name ($.foo()) in order to maintain compatibility with other scripts that may be running in the page. For examples of using jQuery, the Mouse and Keyboard plugins use jQuery to detect input events.

You can find out more about jQuery at http://jquery.org/.

Debugging

Most browsers report javascript errors silently. If your plugin script contains an error, the browser will probably ignore the entire script. This will then prompt an assertion failure during preview similar to "Plugin 'FooBar' is not available". To see the error that caused this, most browsers provide a javascript console. You can find it in the browser menus, or one of the keyboard shortcuts Ctrl+Shift+J, Ctrl+Shift+K, or F12 may work. Most major browsers also implement full javascript debuggers (with breakpoints, watch, call stack etc.) via the same console.

Google Closure Compiler compatibility

When exporting, Construct 2 gives the user the option to 'Minify script'. This runs the common and runtime scripts through Google Closure Compiler's ADVANCED_OPTIMIZATIONS mode. This imposes some limitations on what scripts can do. You must obey these limitations when writing your plugins, otherwise your plugin will be broken on export. More details can be found on the Closure Compiler website.

The main thing is to always use dot syntax (Object.property) rather than bracket syntax (Object["property"]) in your own code. All properties using dot syntax are changed by Closure Compiler, but none of the properties in bracket syntax are changed. Therefore, if you use Object.property in one place and Object["property"] in another to access the same property, the plugin will be broken on export. You may still use bracket syntax (e.g. for a dictionary of user-inputted strings) - just be aware of how Closure Compiler will transform the code.

If you refer to external libraries, you must always use bracket syntax (i.e. Object["property"]). If you use dot syntax, Closure Compiler will rename the property and it will access the wrong property of the external library after export.

Remember the edittime scripts are not passed through Google Closure Compiler, so you can write them how you like.

The Document Object Model (DOM)

As with any javascript running in a browser, you can modify and update the DOM. However, this is not recommended in plugins and behaviors for three reasons:

1. Construct 2's exported projects are intended to be totally self-contained. Ideally the canvas is the only page element affected by the script.
2. Any DOM elements you change may not be present on some pages, or may be present but intended for a different purpose on other pages. Therefore modifying DOM elements can break compatibility with some pages.
3. Non-browser platforms like CocoonJS do not have a DOM, so your plugin will most likely not work on these platforms.

With careful consideration to the above three points, you could still try experimenting with DOM features in plugins.

RUNTIME FUNCTIONS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/runtime-functions

---

First of all, in runtime.js there are two places you must change to insert your plugin ID. Assuming the script is unmodified, these are on lines 9 and 16:

```
// line 9
cr.plugins_.MyPluginID = function(runtime)
// ...
// line 16
var pluginProto = cr.plugins_.MyPluginID.prototype;
```

In both cases you must replace MyPluginID with the plugin ID you specified in edittime.js. Remember to note that the ID must be unique to your plugin, and must not change after release (else all existing projects will break).

Classes

The first part of the script fairly straightforwardly defines three classes: the Plugin class, the Type class, and the Instance class. These represent the plugin, an object type, and an instance of an object type respectively. For example, "Sprite" is a kind of plugin. "Ogre", "Monster" and "Troll" may be three object types in a game, based on the Sprite plugin. In the layout there may be one Ogre, three Monsters and five Trolls - these are the instances of the object types. There is only ever one Plugin class instantiated for a project, and if your plugin flags specify *pf_singleglobal*, there is also only ever one object type and one instance.

You can store whatever you like in each class as necessary, but note you must not remove the runtime required members (e.g. references back to the runtime or plugin). You may also extend the prototypes of each with your own functions.

onCreate()

Each plugin, type and instance class has an onCreate method. These are called after Construct 2 has added any of its own properties to the objects. The object creation always goes:

constructor -> add Construct 2's properties -> onCreate()

For example, the runtime adds the *x*, *y*, *width* and *height* properties to world instances. These are not yet added in the constructor, but can be accessed in onCreate(). After onCreate(), your object is sealed by the runtime (see *Object sealing* in Runtime overview).

onDestroy() and object recycling

To reduce garbage collector overhead, Construct 2 often recycles instance objects (for both plugins and behaviors). This means the instance constructor may be re-called on a previous instance of your object, rather than a new empty javascript object. You should be aware of this - especially the fact the previous instance has already been sealed - when designing plugins and behaviors. You can also take advantage of it to help reduce garbage collector overhead, for example by testing for the existence of members in the constructor and re-using them where possible.

When your instance is destroyed at runtime, Construct 2 calls the instance's onDestroy() method if it has one. However, the object is likely still referenced in the cache for object recycling. This creates a possible problem in that even after being destroyed, the instance still exists. To save memory, you should release references to any large arrays or objects in onDestroy(). Further, you must release references to other objects in the runtime in onDestroy(), otherwise you may have dangling reference bugs. For example, the Platform behavior stores a reference to the last floor object the player landed on. This reference is set to *null* in onDestroy().

draw(ctx) and drawGL(glw)

If your instances appear in the layout, they need to draw themselves. The draw method has a canvas 2D context as a parameter. This context has already been translated and scaled as necessary - you simply need to draw your instance according to its *x*, *y*, *width*, *height* etc. members. The drawGL method is used when WebGL is enabled. Instead of passing the low-level WebGL context itself, a wrapper class is passed - see GLWrap.js in the install directory for a list of methods, or see how the built-in plugins draw in WebGL mode. You should make sure your plugin draws identically in both Canvas 2D and WebGL mode.

If any of your plugins actions or other functions cause a change in the rendering, you must set

```
this.runtime.redraw = true;
```

The canvas does not automatically draw every tick. Do not set the flag if the change has no effect on rendering, since this will trigger a wasteful redraw.

Bounding boxes

If you change an object's size, position or angle, its axis-aligned bounding box changes. You *must* indicate this to Construct 2 by following with a call to

```
instance.set_bbox_changed();
```

If you do not call this, you will cause bugs where objects do not collide properly, disappear near the edge of the screen, and so on. Try not to forget this. It must be called immediately following any changes. Example:

```
instance.x += 1;
instance.y += 1;
instance.set_bbox_changed();
```

ACE IMPLEMENTATIONS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/ace-implementations

---

Actions, conditions and expressions (*ACE* or *ACEs*) defined in edittime.js must each have a corresponding runtime method. In the runtime script you'll find objects named Cnds, Acts and Exps, which you must add your methods to like so:

```
Cnds.prototype.MyCondition = function(...
```

where MyCondition is the script name from edittime.js. See the edittime documentation on ACEs for more - it also covers how to write the runtime functions. You must remember to use dot syntax - see the section on *Google Closure Compiler compatibility* for more.

Implementing conditions

There are four kinds of condition:

Ordinary conditions

These are the default. They are evaluated once per tick (or whenever the event is run). The method is called once for each instance, returning true or false, thus filtering instances meeting the condition. Conditions specifying *cf_faketrigger* are still ordinary conditions.

Trigger conditions

These specify *cf_trigger* (not *cf_faketrigger*). These are not run every tick: they only ever run when explicitly called by runtime.trigger(). This is useful for input events and other on-the-spot events.

Static conditions

These specify *cf_static*. Ordinary conditions are called once for each instance. Static conditions are only ever called once no matter how many instances there are. You must pick the necessary instances in the condition method. For example, 'Pick random object' is most easily implemented as a static condition.

Looping conditions

These are actually ordinary conditions, but are implemented in such a way that they repeat. The *cf_looping* flag should be set, but this does not affect any functionality. Instead, the method must 'retrigger' the event (calling event.retrigger()) which runs the remaining conditions then actions and subevents, then returns to your condition again. The event should be retriggered

once per loop iteration, then the actual condition method must return false - otherwise Construct 2 will carry on and run the event one more time. A good example of a looping condition is Array's *For each element* condition.

Implementing actions

Actions are usually the easiest to implement. They do not need to return anything. Just perform the action!

Note you may not affect any of the picked objects in an action (via the SOL methods), except for any objects passed in object parameters.

Implementing expressions

The most common thing to forget in expressions is the first parameter must be *ret*! The expression returns its value through this rather than by returning from the javascript function itself. Any actual expression parameters follow after *ret*.

Other than that, expressions are also straightforward to implement. The following ret methods are used to return a value:

```
ret.set_int(0);  // return an integer
ret.set_float(0); // return a float
ret.set_string(""); // return a string
ret.set_any(0);  // return a float or string, depending on the javascript type of the paramete
```

Reference

You may use any browser APIs you like in a plugin. That's generally what makes plugins useful! It is best to keep plugins focused, and expose a single API or feature through a single plugin. As described in the overview, the Mozilla Developer Network is a good place to get an overview of browser features and APIs.

The rest of the documentation covers the reference for Construct 2's javascript runtime. These methods may be interesting to you so your plugin can integrate seamlessly with the way Construct 2 works.

Some parts of the runtime are undocumented. This is usually for a reason: it is not advised that plugins use them, since it will cause bugs. Therefore, you should only use the documented parts, in the manner that the documentation states is valid. The fact javascript does not provide encapsulation (*public* and *private* like other languages) is not permission to use the "internal" parts of the runtime.

CREATING A .C2ADDON PACKAGE

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-documentation/c2addon-packages

---

The .c2addon file format allows plugin, behavior and effect developers to easily package their addon in to a single file. Users can then drag and drop the file in to the Construct 2 window to install it.

A .c2addon file is actually a zip file with a renamed extension. The zip file always contains *info.xml* in the root, and a subfolder called *files*. What goes in the *files* folder depends on whether you're publishing a plugin, behavior or effect.

info.xml

The info.xml file specifies metadata about your addon. It states whether the type is a plugin, behavior or effect, and has information like the name, version and author of the addon. Use the info.xml provided in the Javascript SDK as a starting point, and from there it should be straightforward to fill out. Be sure to write documentation and add the link to the documentation to info.xml. You can also find a template of info.xml here.

Plugin and behavior files

When distributing a plugin or behavior, the *files* subfolder needs to contain another subfolder. For example your folder structure in the zip would be:

info.xml
files\myplugin\common.js
files\myplugin\edittime.js
files\myplugin\PluginIcon.ico
files\myplugin\runtime.js

myplugin is the name of the plugin or behavior folder, as it should appear in the install directory. Construct 2 will simply copy and paste this entire folder, preserving the folder name.

Effect files

When distributing an effect, simply place the .fx file and .xml file for the effect in the *files* subfolder in the zip, for example:

info.xml

files\myeffect.fx
files\myeffect.xml

Packaging

Add all the files to a zip (right-click and 'Send to compressed (zipped) folder' in Windows Explorer). Be sure not to accidentally create a root level subfolder in the zip which itself contains info.xml - that will be rejected by Construct 2. Once zipped, rename the file so the extension is .c2addon rather than .zip. Test it by dragging and dropping the file in to the Construct 2 window. If you can install after the prompt successfully, your file is ready to distribute.

Samples

Here are some sample .c2addon packages for an example plugin, behavior and effect. To inspect the contents, rename them to .zip and open them.

- myplugin.c2addon
- mybehavior.c2addon
- myeffect.c2addon

SDK REFERENCE

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-reference

---

This section provides a reference of some of the functions available to the SDK.

RUNTIME REFERENCE

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-reference/runtime

---

The runtime object represents a single instance of a user's project running in the browser. It is usually accessed via this.runtime from an object instance. Remember only documented properties and methods should be used.

Runtime properties

runtime.canvas ***(read-only)***

The canvas element in the page the project is running on.

runtime.width ***(read-only)***
runtime.height ***(read-only)***

The size of the canvas element in the page. This can change at runtime if *Fullscreen in browser* is enabled and the user resizes the browser window.

runtime.redraw ***(write-only)***

This must be set to true whenever anything is done that affects how the project is rendered. If *redraw* remains false, the canvas is assumed not to have changed and will not be redrawn.

runtime.plugins[]

An array of all the plugins used in the project.

runtime.types_by_index[]

An array of all the object types in the project.

runtime.layouts_by_index[]

An array of all the layouts in the project.

runtime.eventsheets_by_index[]

An array of all the event sheets in the project.

runtime.wait_for_textures[] ***(write-only)***

Add a HTML Image() class to this array during loading, and the loader will wait for it to finish downloading before starting the runtime. See Tiled Background for an example.

runtime.timescale

The current time scale.

runtime.kahanTime.sum ***(read-only)***

The current in-game time, in seconds, with timescaling applied.

runtime.tickcount

The number of ticks elapsed since the start of the game.

runtime.changelayout ***(write-only)***

Set to a reference to a layout object and the following tick the runtime will execute *Go to layout* on that layout.

runtime.running_layout ***(read-only)***

A reference to the current layout that is running.

runtime.files_subfolder ***(read-only)***

The subfolder where project files are held.

runtime.extra

The runtime is sealed. You may store any additional properties you need in this object.

runtime.start_time ***(read-only)***

Set to date.getTime() on starting the first layout.

Runtime functions

runtime.tickMe(this)

Call to have the runtime call tick() on your plugin instance. You must define a tick() method in the instance's prototype. Note: behaviors are automatically ticked.

runtime.getDt(this)

Return delta-time, in seconds, with time scaling applied. You must pass an instance to get *dt* for, to maintain compatibility with the *Set object timescale* feature.

runtime.addDestroyCallback(f)

Calls f(inst) whenever an instance is destroyed.

runtime.DestroyInstance(inst)

Destroys the given instance.

runtime.createInstance(type, layer)

Create a new instance of the object *type* on the given layer. Returns a reference to the created instance.

runtime.getLayerByName(name)

Get a layer by name, case insensitive, or *null* if not found.

runtime.getLayerByNumber(index)

Get a layer by zero-based index, or *null* if out of bounds.

runtime.testAndSelectCanvasPointOverlap(type, x, y, inverted)

Pick any instances of *type* that overlap the point (x, y) in canvas co-ordinates. Pass the condition's inverted state for correct behavior. See the Mouse object for an example.

runtime.testOverlap(a, b)

Return true if the two given instances are overlapping.

runtime.testOverlapSolid(inst)

Returns an instance with the Solid attribute if *inst* is overlapping a solid, else *null*.

runtime.pushOutSolid(inst, xdir, ydir, dist)

Pushes *inst* the distance given by *xdir* and *ydir* up to *dist*, until *inst* is not overlapping any solid. If *inst* finishes overlapping a solid, it is put back to its original position and *false* is returned. Otherwise, *true* is returned.

runtime.pushOutSolidNearest(inst, max_dist)

Pushes *inst* in an 8-direction spiral pattern up to *max_dist* pixels away until it is no longer overlapping a solid. If *inst* is still overlapping a solid by *max_dist*, it is returned to its original position and *false* is returned. Otherwise, *true* is returned.

runtime.trigger(method, inst)

Trigger the condition specified by *method*, with the object *inst* triggering. The condition must specify *cf_trigger* and not *cf_faketrigger*. If triggering your own plugin's trigger, pass this for *inst*. *method* must specify the plugin routine in the form:

cr.plugins_.MyPluginID.prototype.cnds.MyTrigger

e.g.:

cr.plugins_.Mouse.prototype.cnds.OnClick

runtime.getCurrentCondition()

Returns the current condition. Only valid in condition methods.

runtime.getCurrentAction()

Returns the current action. Only valid in action methods.

runtime.getCurrentEventStack()

Return the current event "stack frame". This can be used to determine the current event via runtime.getCurrentEventStack().current_event.

LAYOUT FUNCTIONS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-reference/layout

---

A layout represents a layout in the project at runtime. Only one layout can be running at a time.

Layout properties

layout.runtime

A reference back to the runtime.

layout.event_sheet

A reference to the layout's event sheet, or null if it does not have one.

layout.name

The layout's name.

layout.width
layout.height

The size of the layout, in pixels.

layout.unbounded_scrolling

A boolean indicating the 'unbounded scrolling' setting.

layout.layers[]

An array of layers on the layout.

Layout functions

layout.scrollToX(x)
layout.scrollToY(y)

Scroll the layout to the given coordinates.

LAYER FUNCTIONS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-reference/layer

---

Layouts consist of multiple layers. All layout objects belong to a layer.

Layer properties

layer.layout

A reference back to the layout the layer is on.

layer.runtime

A reference back to the runtime.

layer.scale

The current layer scale.

layer.viewLeft
layer.viewRight
layer.viewTop
layer.viewBottom

Defines the rectangle of the currently visible viewport. This may be larger or smaller than the canvas size if the scale is not 1.0.

layer.name

The layer name.

layer.index

The zero-based layer index.

layer.visible

A boolean indicating if the layer is currently visible.

layer.background_color

The layer's background color, as an array in the format [r, g, b, a]. Ignored if the layer is transparent.

layer.transparent

A boolean indicating if the layer is transparent.

layer.parallaxX
layer.parallaxY

The layer's parallax rate for the X and Y axes.

layer.opacity

The layer's opacity, from 0 (transparent) to 1 (opaque).

layer.forceOwnTexture

A boolean indicating the *Force own texture* setting.

layer.instances[]

An array of all the object instances (of any object type) currently on this layer.

Layer functions

layer.canvasToLayerX(x)
layer.canvasToLayerY(y)

Convert from canvas coordinates to layer coordinates. Useful for converting e.g. mouse co-ordinates to layer co-ordinates. See Mouse for an example.

OBJECT TYPE FUNCTIONS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-reference/object-type

---

See *Classes* in *Runtime functions* for a description of plugins vs. object types vs. instances. Since SOLs (selected object lists) are based on object types, these are also documented below.

Object type properties

type.plugin

A reference to the plugin the object type is from.

type.texture_file

Only valid when *pf_texture* is specified. The filename of the texture PNG file.

type.texture_filesize

Only valid when *pf_texture* is specified. The file size of the texture PNG file, generally used to aid the progress bar accuracy.

type.animations

Only valid when *pf_animations* is specified. Stores the object animations. See Sprite for an example.

type.index

The zero-based index of the object type in the runtime's types_by_index array.

type.instances[]

Array of all the currently created instances of this object type.

type.behaviors[]

Array of all the behaviors added to this object type.

#Object type functions#

type.getFirstPicked()

Return the first instance of this type. If in an event, this returns the first picked instance, otherwise the first instance in the instances array. Returns *null* if no instances exist or are picked.

type.getPairedInstance(inst)

inst must be from another object type. Return the instance of this object type that corresponds to *inst*.

type.getCurrentSol()

Returns the current SOL object. See below.

The SOL

The SOL (selected object list) is the list of all instances currently matching the event. Conditions filter instances matching the condition from the full instance list. In an action, condition or expression, you may wish to access this list (e.g. for static conditions). getCurrentSol() on the object type returns its SOL object, which is documented here.

Remember you must not modify the SOL for any object types other than:

- the current object type in a condition, or
- any object type passed in an object parameter.

You may have read-only access to all other SOLs, though.

SOL properties

sol.type

Reference to the object type the SOL is for.

sol.instances[]

Array of the instances currently matching the event. If sol.select_all is *true*, this is ignored and may have undefined contents (e.g. arbitrary instances left over from a previous event). Therefore, only use this when sol.select_all is *false*.

sol.select_all

If *true*, this specifies that sol.instances must be ignored and sol.type.instances used (the list of all instances). If *false*, this indicates the contents of sol.instances is the current selection. If setting from *true* to *false*, you must clear sol.instances and fill it with your intended content.

SOL functions

sol.hasObjects()

Returns *true* if the object type has at least one instance and select_all is *true*, or if the SOL instances array is not empty and select_all is *false*, else *false*.

sol.getObjects()

Returns a reference to the current array of instances. If select_all is *true*, this is the type's array of all instances. If *false*, this is the SOL's current instances list.

sol.pick(inst)

If select_all is *true*, sets it to *false* and picks *inst*. Otherwise, ensures *inst* is in the SOL instances array. Note: avoid this function where possible, since it runs in *O(n)* time and will be slow for large projects.

INSTANCE FUNCTIONS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-reference/instances

---

Instances are generally plugin-defined. You should read the sources of a plugin to see how it is implemented. However, Construct 2 guarantees the presence of certain properties and functions. These common properties and functions are documented here. Note any properties or functions relating to objects in a layout (e.g. *x*, *y*) are not present in non-layout objects.

Common instance properties

inst.type

Reference to the instance's object type.

inst.uid *(read-only)*

The instance's unique ID. Construct 2 issues the first instance a UID of 0 and increments by 1 for each new instance created. UIDs must never change through the life of an object.

inst.instance_vars[]

Array of instance variables. Each element is an ordinary javascript value corresponding to the current value of the instance variable at that index.

inst.x
inst.y

The current position of the instance in the layer. If changed, you must call inst.set_bbox_changed().

inst.width
inst.height

The current size of the instance, in pixels. If changed, you must call inst.set_bbox_changed().

inst.angle

The current object angle, in radians. Not all objects support an angle, e.g. Text object. If changed, you must call inst.set_bbox_changed().

inst.opacity

The current object opacity, from 0 (transparent) to 1 (opaque). Not all objects

use the opacity.

inst.hotspotX
inst.hotspotY

Current position of the hotspot, in texture co-ordinates (e.g. (0.5, 0.5) will be a centered hotspot).

inst.bbox

A cr.rect representing the instance's axis aligned bounding box. You must call inst.update_bbox() before using this property.

inst.bquad

A cr.quad representing the instance's bounding quad. You must call inst.update_bbox() before using this property.

inst.visible

A boolean indicating if the object is currently visible.

inst.layer

A reference to the layer the object is on, if a world object.

inst.behavior_insts[]

Array of instances of each behavior added to the object type.

inst.inst

In behavior instances only: this is the reference to the object instance your behavior should modify.

Common instance functions

inst.set_bbox_changed()

You must call this to indicate to Construct 2 that the object's bounding box has changed, after modifying the *x*, *y*, *width*, *height* or *angle* properties.

inst.update_bbox()

You must call this before accessing the bbox or bquad members of an instance. Otherwise, their values will be invalid.

inst.add_bbox_changed_callback(f)

Calls f(inst) whenever set_bbox_changed() is called on *inst*. Warning: this can cause a large performance overhead, so use with care.

inst.get_iid()

Get the Instance ID (IID). This is the zero-based index in the object type's instances array where this instance is located. Note: you must use this function to get the IID, since IIDs are lazily assigned.

inst.toString()

Overridden to return a string in the format "inst:Type[#]uid" e.g. "inst:Player[#]0".

CR FUNCTIONS

View online: https://www.construct.net/en/construct-2/manuals/construct-2-javascript-sdk/sdk-reference/cr-functions

---

In the rest of the documentation, you may have noticed references to cr.vector2 or cr.rect. These, and some other common functions, are declared in common_prelude.js. Like the rest of the runtime, they are in the cr namespace (for Construct Runtime). This section documents these classes and functions.

cr.vector2

cr.vector2 is a simple x,y position (also usable as a size). Create with: new cr.vector2(x, y)

Since r52, it is not recommended to use cr.vector2 in runtime scripts. The use of many temporary vector2s creates a lot of garbage collector overhead which can cause poor performance in some browsers. Prefer to directly manipulate separate *x* and *y* javascript numbers instead.

cr.vector2.x
cr.vector2.y

The x and y co-ordinates of the vector./dd]

cr.vector2.offset(x, y)

Vector addition. Modifies the vector2 offset() is called on.

cr.vector2.mul(x, y)

Vector multiplication. Modifies the vector2 mul() is called on.

cr.rect

cr.rect is a simple 2D rectangle. It is implicitly axis-aligned. Create with: new cr.rect(left, top, right, bottom)

Creating many rect objects can result in garbage collector overhead causing poor performance. Re-use existing rect objects wherever possible.

cr.rect.left
cr.rect.top
cr.rect.right
cr.rect.bottom

Defines the position of the rectangle.

cr.rect.set(left, top, right, bottom)

Convenience function to set each member in one call.

cr.rect.width()
cr.rect.height()

Return the size of the rectangle.

cr.rect.offset(x, y)

Offset the rectangle by the given x and y co-ordinates.

cr.rect.intersects_rect(rc)

Test if the rectangle intersects another cr.rect.

cr.rect.contains_pt(x, y)

Test if the point lies inside or on the border of the rect.

cr.quad

A quad is simply four 2D points that form a four-sided shape. Unlike cr.rect, quads can represent rectangles at any angle (non-axis-aligned). Quads do not have to store a rectangle - each of the four points can have any position at all - but in the runtime, they are always used for the purpose of rotated rectangles.

Create a quad with: new cr.quad()

cr.quad.tlx
cr.quad.tly
cr.quad.trx
cr.quad.try_ ***(note underscore to avoid "try" keyword)***
cr.quad.brx
cr.quad.bry
cr.quad.blx
cr.quad.bly

The co-ordinates of the four points making up the quad. The terms tl, tr, br and bl refer to "top left", "top right", "bottom right" and "bottom left" respectively (which correspond to their positions when representing an unrotated rectangle).

cr.quad.set_from_rect(rc)

Set the quad points to represent a shape that is identical to the given

cr.rect

object.

cr.quad.set_from_rotated_rect(rc, a)

Set the quad points to represent a cr.rect rotated by *a* radians.

cr.quad.offset(x, y)

Offset the quad by a point.

cr.quad.bounding_box(rc)

Set the cr.rect *rc* to a rect representing the axis-aligned bounding box of the quad.

cr.quad.contains_pt(x, y)

Test if the point lies inside the quad.

cr.quad.intersects_quad(q)

Test if the quad intersects another cr.quad.

cr.ObjectSet

An object set represents a mathematical set of javascript objects, i.e. each object can only be stored once or not at all, never twice. Adding an object to an ObjectSet that already contains it has no effect. The ObjectSet uses the .toString() method to distinguish objects, so any objects stord in the ObjectSet must have an appropriate overload that uniquely identifies it. Since the underlying container is a javascript object, performance is better than using an array and lookups to store unique objects.

Create a new ObjectSet with: new cr.ObjectSet()

cr.ObjectSet.contains(x)

Test if the set contains object *x*.

cr.ObjectSet.add(x)

Add the object *x* to the set if not already present, else no effect.

cr.ObjectSet.remove(x)

Remove the object *x* from the set if present, else no effect.

cr.ObjectSet.clear()

Remove all objects from the set, returning it to an empty state.

cr.ObjectSet.isEmpty()

Returns true if the object set is in an empty state.

cr.ObjectSet.count()

Return the number of objects stored in the set.

cr.ObjectSet.valuesRef()

Return a read-only reference to the javascript array containing all the objects in the set. This is faster than values().

cr.ObjectSet.values()

Return an array containing a copy of all the objects in the set. Since it returns a copy, the result can be modified.

Utility functions

The following functions are not all related, but are often useful.

cr.RGB(red, green, blue)

Generate a color value. Useful for color parameters.

cr.arrayRemove(arr, index)

Modify the array parameter *arr* to remove the element at *index*.

cr.arrayFindRemove(arr, item)

Modify th array parameter *arr* to remove the element equal to *item*.

cr.clamp(x, a, b)

Return *x*, or *a* if *x* is lower than *a*, or *b* if *x* is greater than *b*.

cr.to_radians(a)

Return *a* converted from degrees to radians.

cr.to_degrees(a)

Return *a* converted from radians to degrees.

cr.clamp_angle_degrees(a)

Return *a* wrapped in to the range [0, 360)

cr.to_clamped_degrees(a)

Return *a* converted from radians to degrees and wrapped in to the range [0, 360).

cr.to_clamped_radians(a)

Return *a* converted from degrees to radians and wrapped in to the range [0, 2pi).

cr.angleDiff(a1, a2)

Return the smallest angle, in radians, between *a1* and *a2*.

cr.angleRotate(start, end, step)

Return *start* rotated towards *end* by *step* radians.

cr.angleClockwise(a1, a2)

Return true if *a2* is clockwise of *a1* (in radians) by the smallest angle.

cr.xor(x, y)

Return logical XOR of *x* and *y* (javascript has no native operator for this).

cr.lerp(a, b, x)

Linearly interpolate *a* to *b* by *x*.

cr.segments_intersect(a1x, a1y, a2x, a2y, b1x, b1y, b2x, b2y)

Test if the line *a1* to *a2* intersects the line *b1* to *b2*.

cr.seal(o)

Seal the object *o*. If ECMAScript 5 is not supported, has no effect.

cr.freeze(o)

Freeze the object *o*. If EXMAScript 5 is not supported, has no effect.