C3 ADDON SDK DOCUMENTATION

View online: https://www.construct.net/en/make-games/manuals/addon-sdk

Welcome to the Construct 3 Addon SDK documentation. The Addon SDK allows third-party developers to create new addons for Construct 3 using JavaScript or TypeScript. This includes:

- **Plugins:** new kinds of object that appear in the *Create new object* dialog, with their own actions, conditions and expressions. Plugins are ideal for integrating third-party services. There are also two ways plugins can provide enhanced platform integration, accessing features not normally available in browsers:
    - Plugins can specify a **Cordova plugin** dependency for enhanced integration on mobile. See Specifying dependencies for more details.
    - Plugins can bundle a **wrapper extension** for enhanced integration on desktop. See the guide on wrapper extensions for more details.
- **Behaviors:** new kinds of behaviors that appear in the *Add behavior* dialog, with their own behavior actions, conditions and expressions that get added to the object the behavior is added to. This can be used for creating new rapid prototyping features, or advanced gameplay logic that integrates with event sheets.
- **Effects:** new kinds of visual effects that appear in the *Add effect* dialog. These are custom fragment shaders written in GLSL for WebGL and WGSL for WebGPU.
- **Themes:** allow custom appearances for the Construct 3 editor, using additional CSS stylesheets to change the default editor appearance.

Download

The Addon SDK files are hosted on the Construct Addon SDK GitHub repository. Follow the link, click the green *Code* button, and select *Download ZIP* to download a copy of the files.

There are sample files for example custom plugins, behaviors, effects and themes. The files for an addon can be zipped and renamed .c3addon to directly test it in the Construct 3 editor, via the Addon Manager.

While developing addons, be sure to use Developer Mode with a local HTTP server. It makes it much quicker to test since you don't need to keep creating .c3addon files, and much easier to fix problems, which otherwise can prevent Construct 3 from starting up.

Custom importer API sample data

The plugin SDK includes a sample plugin using the Custom Importer API. The included file *customImporterSampleData.zip* can be drag-and-dropped in to Construct 3 to demonstrate reading a custom format.

TypeScript support

The addon SDK has optional support for TypeScript, allowing for static typing, better autocomplete, and a range of other benefits for addon development. To learn more see the section on TypeScript support.

Learning web technologies

The Addon SDK documentation assumes you have a basic knowledge of JavaScript. A basic knowledge of HTML and CSS may also be useful. This documentation does not attempt to teach you these technologies. If you're just starting out, we recommend the MDN web docs as a good place to start. It provides thorough documentation on all aspects of the web platform, and also includes guides for learning web development.

THE .C3ADDON FILE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/c3addon-file

Construct 3 addons are distributed as a .c3addon file. This is simply a .zip file renamed to end with .c3addon. You can rename a .c3addon file to .zip and inspect its contents, or create a .c3addon file by zipping some files and renaming it.

Make sure all the files are at the root level of the zip. If you accidentally place all files in a subfolder inside the zip, you'll get the error "missing addon.json" when trying to install it in Construct, because it expects to find that file at the root level.

Structure of a .c3addon file

The files in a .c3addon for a plugin are arranged as follows. Note some features allow extra files to be added, but this is the minimal file structure for an empty plugin/behavior.

- **c3runtime/** — subfolder for Construct 3 runtime files.
- **lang/en-US.json** — language file containing strings shown in the user interface. This is kept in a separate file to facilitate translation.
- **aces.json** — JSON data file that defines actions, conditions and expressions.
- **addon.json** — JSON data file with metadata about the addon.
- **icon.svg** — addon icon.
- **plugin.js** or **behavior.js** — class representing the plugin or behavior.
- **type.js** — class representing an object type of the plugin, or behavior type of the behavior, in the editor.
- **instance.js** — class representing an instance of the plugin, or behavior instance of the behavior, in the editor.

Specifying metadata

The basic metadata about your addon, such as its ID and type, is set in **addon.json**. See Addon metadata for more information.

Setting plugin information

For plugins, the detailed information about the plugin and its capabilities is set in **plugin.js**. See Configuring plugins for more information.

Setting behavior information

For behaviors, the detailed information about the behavior and its capabilities is set in **behavior.js**. See Configuring behaviors for more information.

Setting effect information

For effects, the detailed information about the effect and its capabilities is set using extra properties in **addon.json**. The effect itself is written in a **.fx** file. See Configuring effects for more information.

Defining actions, conditions and expressions

To define your plugin or behavior's actions, conditions and expressions (ACEs), they must be specified in **aces.json** and the corresponding language strings added in **en-US.json**. (Currently the language file must be in US English, but the fact it is in a separate file will help facilitate translation in future.)

For more information see Defining actions, conditions and expressions.

ADDON METADATA

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/addon-metadata

The metadata for your addon, specifying details like its ID and type, is defined by **addon.json**. An example is shown below.

```
{
	"is-c3-addon": true,
	"sdk-version": 2,
	"type": "plugin",
	"name": "My custom plugin",
	"id": "MyCompany_MyAddon",
	"version": "1.0.0.0",
	"author": "Scirra",
	"website": "https://www.construct.net",
	"documentation": "https://www.construct.net",
	"description": "Example custom Construct 3 plugin.",
	"editor-scripts": [
								"plugin.js",
								"type.js",
								"instance.js"
							],
	"file-list": [
		"c3runtime/plugin.js",
		"c3runtime/type.js",
		"c3runtime/instance.js",
		"c3runtime/conditions.js",
		"c3runtime/actions.js",
		"c3runtime/expressions.js",
		"lang/en-US.json",
		"aces.json",
		"addon.json",
		"icon.svg",
		"instance.js",
		"plugin.js",
		"type.js"
	]
}
```

Note some of the information is duplicated elsewhere in the addon's files. This is because the editor reads this file before it loads any other files when asking the user if they want to install the addon. Note information specified here, such as the ID, must exactly match everywhere else it is used.

The addon SDK provides a JSON schema to help you write addon.json files, as it provides autocomplete and validation in compatible editors.

Each field and its possible values are described below.

---

is-c3-addon

Boolean set to `true`. This is used by Construct 3 to identify valid addons.

---

sdk-version

The addon SDK version the addon has been built with. Currently only SDK version 2 is supported. Effects and themes are not affected by the addon SDK version and can omit this field.

---

type

One of `"plugin"`, `"behavior"`, `"effect"` or `"theme"`, indicating the kind of addon this is.

---

name

The displayed name of the addon, in English.

---

id

The unique ID of the addon. This is not displayed and is only used internally. This must not be used by any other addon ever published for Construct 3, and must never change after you first publish your addon. (The name is the only visible identifier of the addon in the Construct 3 editor, so that can be changed any time, but the ID must always be the same.) To ensure it is unique, it is recommended to use a vendor-specific prefix, e.g. `MyCompany_MyAddon`. It must match the ID set in plugin.js.

---

version

A string specifying the addon version in four parts (major, minor, patch, revision). Be sure to update this when releasing updates to your addon. It must match the version set in plugin.js/behavior.js.

---

author

A string identifying the author of the addon.

---

website

A string of a URL to the author's website. It is recommended to provide updates to the addon at this URL if any become available. The website should use HTTPS.

documentation

A string of a URL to the online documentation for the addon. It is important to provide documentation for your addon to be useful to users.

description

A string of a brief description of what the addon does, displayed when prompting the user to install the addon.

min-construct-version

The minimum Construct version required to load your addon, e.g. "r399". If not specified, the addon will be allowed to be installed with any version of Construct. If specified and the user attempts to install the addon with a version lower than the minimum, then Construct will prevent installation and show a message indicating that a newer version of Construct must be used. If the user installs the addon with a newer version of Construct and then rolls back to an older version of Construct lower than the minimum, then Construct will refuse to load the addon (a message will be logged to the console) and the editor will act as if the addon is not installed.

supports-worker-mode

A boolean indicating whether the addon supports Construct's worker mode, where the entire runtime is hosted in a Web Worker instead of the main thread. This defaults to `true`. Providing the addon only uses APIs available in a Web Worker, then it is compatible; where access to the DOM is necessary, then a DOM script can be used to still access those features in worker mode - see Runtime scripts for more details. Therefore it should be possible for every addon to support worker mode, and supporting it is strongly recommended as worker mode can bring performance benefits. This can be set to `false` to indicate that the addon does not yet support worker mode, which may be useful to expedite addon development or if the addon makes use of extremely complex DOM operations. This will cause worker mode "auto" to switch to DOM mode which may degrade the performance of the project. If the user attempts to switch worker mode to "Yes" in project using the addon, then Construct will show an error message highlighting the addon that does not support the mode, and prevent changing the setting.

editor-scripts

For plugins and behaviors only. An array of script files in the addon package to load in the editor. It is recommended to leave this at the default unless you have large editor dependency scripts, or if you want to minify your addon in to a single script. Note themes do not use editor scripts.

stylesheets

For themes only. An array of CSS files in the addon package to apply to the document. These are the CSS files that define the theme's appearance.

file-list

For developer mode addons only. A list of all files used by the addon. This is required for Developer Mode addons since there is no other mechanism for Construct to determine the list of files when serving files from a web server. Be sure to update this property if you add, rename or remove any files in your addon.

Additional properties for effects

When developing an effect addon, additional information about the effect is included in the addon.json file. For more information see Configuring effects.

TYPESCRIPT SUPPORT

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/typescript-support

The Addon SDK supports using TypeScript to develop plugins and behaviors. All the addon SDK samples have TypeScript support, with a .ts equivalent of every .js file, to help make it easy to get going with TypeScript. However using TypeScript is optional - if you prefer to stick to just JavaScript, then ignore or delete any .ts files in the addon SDK samples and just keep working with .js files.

Note this process is similar to using TypeScript for Construct projects, but with some altered steps due to the fact that addons are not associated with a project.

Installation

To use TypeScript with the Addon SDK, you will need a TypeScript-compatible code editor. This guide uses Visual Studio Code, or VS Code for short.

Install VS Code using the link above if you don't already have it. You'll also need to install TypeScript support, which you can do by following these steps:

1. Install Node.js if you don't already have it
2. In a terminal, run the command `npm install -g typescript`

You can check the TypeScript compiler, or `tsc` for short, is installed by running `tsc --version` in the terminal. It should print the version installed.

Modern versions of Windows use PowerShell for the terminal, and running some of the above commands could return an error like tsc.ps1 cannot be loaded because running scripts is disabled on this system due to the security restrictions set by default. To fix this, run the command `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser` to allow permission to run the command, and then try the original command again.

For more details see TypeScript in Visual Studio Code.

Setting up the addon files

The SDK sample addons come with TypeScript support, having a .ts file for every .js file, so you don't need to do anything in this step if you're starting with one of those samples.

If you want to add TypeScript support to an existing JavaScript-only addon, then create a copy of every .js file with the file extension renamed to .ts. You'll then need to go through every .ts file and add type annotations to it. The SDK samples should provide a guide of what kinds of type annotations and adjustments to make. Note that in some cases you may need to import and export types between files to ensure all types are checked correctly.

Set up TypeScript

For proper type checking, you'll need to export Construct's type definitions to your addon folder. To do that, follow these steps.

1. In Construct, enable developer mode and then reload Construct.
2. In the main menu, a new *Developer mode* submenu should have appeared. In that submenu, select *Set up TypeScript for addon*. Note this requires a browser that supports the File System Access API - try using Chrome or Edge if the option does not appear.
3. A folder picker appears. Select the folder that contains your addon (where addon.json is located).

This will make two changes to your addon folder:

1. A subfolder *ts-defs* is created, with a range of TypeScript definition files for Construct's APIs.
2. The file tsconfig.json (TypeScript configuration file) is created in the addon folder with default TypeScript settings, but only if the file does not exist. If tsconfig.json already exists then it does not alter it.

Note that Construct's TypeScript definition files will be added to and possibly revised over time. To update the TypeScript definition files to the latest version, choose *Set up TypeScript for addon* again in a future release of Construct, and it will update all the TypeScript definition files to the latest versions. Remember that if tsconfig.json already exists then this does not change it, so any alterations you've made to the TypeScript configuration will be persisted.

Workflow

Once you are up and running, you will likely want to make repeated changes to your TypeScript code, and easily be able to test your addon in Construct.

In VS Code, press `Ctrl` + `Shift` + `B` and then select *tsc: watch*. This enables a mode where VS Code will automatically compile your .ts files to .js whenever you save the file. Note this must be done once per session.

Assuming you are using a developer mode addon, then your workflow can go like this:

1. You make a change to a TypeScript file and save the change
2. TypeScript then automatically compiles the .ts file to .js (or reports errors if you made a mistake)
3. When you next preview a project in Construct, it will reload the latest .js files from your local web server.
4. Then the preview starts up running your latest code changes.

Summary

TypeScript support takes a couple of steps to set up - in particular using the *Set up TypeScript for addon* option - but brings many benefits to addon development, such as type checking and better auto-complete in compatible editors like VS Code. Once set up the workflow is largely automatic and you can go from changing a .ts file to seeing it in Construct on your next preview. Construct's type definitions will change over time, but you can just repeat the *Set up TypeScript for addon* step to update the type definitions to the latest version.

CONFIGURING PLUGINS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/configuring-plugins

The main configuration for a plugin is set in **plugin.js**.

Plugin constants

The following constants are defined in the file-level scope:

```
const PLUGIN_ID = "MyCompany_MyAddon";
const PLUGIN_CATEGORY = "general";
```

The ID and version constants must match the values specified in addon.json.

PLUGIN_ID

This is a unique ID that identifies your plugin from all other addons. This must not be used by any other addon ever published for Construct 3. **It must never change** after you first publish your addon. (The name is the only visible identifier of the addon in the Construct 3 editor, so that can be changed any time, but the ID must always be the same.) To ensure it is unique, it is recommended to use a vendor-specific prefix, e.g. `MyCompany_MyAddon`.

PLUGIN_CATEGORY

The category for the plugin when displaying it in the *Create New Object Type* dialog. This must be one of `"3d"`, `"data-and-storage"`, `"form-controls"`, `"general"`, `"input"`, `"media"`, `"monetisation"`, `"platform-specific"`, `"web"`, `"other"`.

Updating plugin identifiers

The main class declaration of the plugin looks like this:

```
const PLUGIN_CLASS = SDK.Plugins.MyCompany_MyAddon = class MyCustomPlugin extends SDK.IPluginBas
```

Be sure to update the identifiers to describe your own plugin, in both the SDK namespace and the `class` name.

Updating in type.js and instance.js

Likewise in both type.js and instance.js, you must update the following:

- `PLUGIN_CLASS` to refer to your plugin's name

- The `class` name suffixed with `Type` or `Instance`. (For example the Audio plugin uses `AudioPlugin`, `AudioType` and `AudioInstance` as the three names.)

Optional plugin scripts

With the addon SDK, you may omit the editor script files type.js and instance.js, as well as the runtime script files plugin.js and type.js. If these files are omitted, it uses the default base class with no modifications. To remove these files, be sure to do the following:

1. Delete any unused script files
2. Remove the files from the `"file-list"` array in addon.json
3. Remove any unused editor script files from the `"editor-scripts"` array in addon.json
4. In the editor plugin script, call `this._info.SetC3RuntimeScripts()` with an array of the runtime script files in use, as the default list includes c3runtime/plugin.js and c3runtime/type.js.

The plugin constructor

The main function of plugin.js is to define a class representing your plugin. In the class constructor, the configuration for the plugin is set via the `this._info` member, which is an IPluginInfo interface. The constructor also reads potentially translated strings from the language subsystem.

For more information about the possible plugin configurations, see the `IPluginInfo` reference.

Specifying plugin properties

The plugin properties appear in the Properties Bar when instances of the plugin are selected. To set which properties appear, pass an array of PluginProperty to `this._info.SetProperties`. An example is shown below. For more details see the `PluginProperty` reference.

```
	this._info.SetProperties([
		new SDK.PluginProperty("integer", "test-property", 0)
	]);
```

CONFIGURING BEHAVIORS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/configuring-behaviors

The main configuration for a behavior is set in **behavior.js**.

Behavior constants

The following constants are defined in the file-level scope:

```
const BEHAVIOR_ID = "MyCompany_MyAddon";
const BEHAVIOR_CATEGORY = "general";
```

The ID and version constants must match the values specified in addon.json.

BEHAVIOR_ID

This is a unique ID that identifies your behavior from all other addons. This must not be used by any other addon ever published for Construct 3. **It must never change** after you first publish your addon. (The name is the only visible identifier of the addon in the Construct 3 editor, so that can be changed any time, but the ID must always be the same.) To ensure it is unique, it is recommended to use a vendor-specific prefix, e.g. `MyCompany_MyAddon`.

BEHAVIOR_CATEGORY

The category for the behavior when displaying it in the *Add behavior* dialog. This must be one of `"attributes"`, `"general"`, `"movements"`, `"other"`.

Updating behavior identifiers

The main class declaration of the behavior looks like this:

```
const BEHAVIOR_CLASS = SDK.Behaviors.MyCompany_MyAddon = class MyCustomBehavior extends SDK.IBeh
```

Be sure to update the identifiers to describe your own behavior, in both the SDK namespace and the class name.

Updating in type.js and instance.js

Likewise in both type.js and instance.js, you must update the following:

- `BEHAVIOR_CLASS` to refer to your behavior's name
- The `class` name suffixed with `Type` or `Instance`. (For example the Bullet behavior uses `BulletBehavior`, `BulletType` and `BulletInstance` as the three names.)

Optional behavior scripts

With the addon SDK, you may omit the editor script files type.js and instance.js, as well as the runtime script files plugin.js and type.js. If these files are omitted, it uses the default base class with no modifications. To remove these files, be sure to do the following:

1. Delete any unused script files
2. Remove the files from the `"file-list"` array in addon.json
3. Remove any unused editor script files from the `"editor-scripts"` array in addon.json
4. In the editor behavior script, call `this._info.SetC3RuntimeScripts()` with an array of the runtime script files in use, as the default list includes c3runtime/behavior.js and c3runtime/type.js.

The behavior constructor

The main function of behavior.js is to define a class representing your behavior. In the class constructor, the configuration for the behavior is set via the `this._info` member, which is an IBehaviorInfo interface. The constructor also reads potentially translated strings from the language subsystem.

For more information about the possible behavior configurations, see the `IBehaviorInfo` reference.

Specifying behavior properties

The behavior properties appear in the Properties Bar when instances using the behavior are selected. To set which properties appear, pass an array of PluginProperty to `this._info.SetProperties`. An example is shown below. For more details see the `PluginProperty` reference. (Note that behaviors use the same property class as plugins, hence re-using the PluginProperty class for behaviors.)

```
        this._info.SetProperties([
                new SDK.PluginProperty("integer", "test-property", 0)
        ]);
```

CONFIGURING EFFECTS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/configuring-effects

The main configuration for an effect is set by additional effect-specific properties in addon.json. The additional properties used by effects are listed below.

category

The category the effect should appear in. This must be one of `"3d"`, `"blend"`, `"color"`, `"distortion"`, `"mask"`, `"normal-mapping"`, `"tiling"`, `"other"`.

supported-renderers

An array of strings indicating the supported renderers for this effect. By default (if omitted) it is `["webgl"]`. The string `"webgl2"` can be added to support a WebGL 2 variant of the effect - see the section on WebGL shaders for more details. The string `"webgpu"` can be added to support the WebGPU renderer with a shader written in WGSL - see the section on WebGPU shaders for more details.

blends-background

Boolean indicating whether the effect blends with the background. Objects and layers can use effects that blend with the background, but layouts cannot.

uses-depth

Boolean indicating whether the effect samples the depth buffer with the `samplerDepth` uniform. This is used for depth-based effects like fog.

cross-sampling

Boolean indicating whether a background-blending effect has inconsistent sampling of the background and foreground. A normal blending shader like Multiply will sample the background and foreground 1:1, so each foreground pixel samples only the background pixel it is rendered to. This is consistent sampling so `cross-sampling` should be `false`. However an effect that distorts the background, like Glass or a masking Warp effect, can sample different background pixels to the foreground pixel being rendered, so should set `cross-sampling` to `true`. This must be specified so the effect compositor can ensure the correct result is rendered when this happens.

preserves-opaqueness

Boolean indicating whether the effect preserves opaque pixels, i.e. every input pixel with an alpha of 1 is also output with an alpha of 1. This is true for most color-altering effects, but not for most distorting effects, since in some cases a previously opaque pixel will be distorted in to a transparent area of the texture. This information is not currently used, but is important for front-to-back rendering algorithms.

animated

Boolean indicating whether the effect is animated, i.e. changes over time using the `seconds` uniform. This is used to ensure Construct keeps redrawing the screen if an animated effect is visible.

must-predraw

Boolean indicating whether to force the pre-draw step. Sometimes Construct tries to optimise effect rendering by directly rendering an object with the shader applied. Setting this flag forces Construct to first render the object to an intermediate surface, which is necessary for some kinds of effect.

supports-3d-direct-rendering

Boolean indicating whether 3D objects can render directly with this effect. This defaults to `false`, causing all 3D objects with the effect to first perform a pre-draw step, and then processing the effect on a 2D surface. If set to `true`, then 3D objects with the effect are allowed to render directly to the display with the effect being processed for each triangle in the geometry. This is usually more efficient and can be more appropriate for processing 3D effects. Note however that the effect compositor may still decide to add a pre-draw step in some circumstances, so this setting is not a guarantee that it will always use direct rendering.

extend-box

Amount to extend the rendered box horizontally and vertically. Normally the effect is clipped to the object's bounding box, but some effects like Warp need to be able to render a short distance outside of that for the correct result. This property lets you extend the rendered box by a number of pixels. This property uses `"horizontal"` and `"vertical"` sub-properties, e.g. `"extend-box": { "horizontal": 30, "vertical": 30 }`

is-deprecated

Boolean to indicate a deprecated effect. This hides the effect from the *Add effect* dialog, but allows existing projects to continue using it. This allows an effect to be phased out without breaking projects.

parameters

An array of parameters that the effect uses. See the next section for more information.

Effect parameters

The `parameters` array in addon.json specifies a list of parameters that are passed to the shader as uniforms. These can be used to customise the appearance of the effect, and can also be changed at runtime. Each parameter is specified as an object with the following properties.

id

A string identifying this parameter.

c2id

Optional The corresponding ID used in a compatible legacy Construct 2 effect if this is not the same as the `id`. Note for color parameters, this can be a comma-separated list of the three parameter IDs previously used for the red, green and blue components, e.g. `"red,green,blue"`.

type

The type of the effect parameter. This can be one of `"float"`, `"percent"` or `"color"`. Floats pass a simple number. Percent displays a percentage in the 0-100 range but passes a float in the 0-1 range to the shader. Color shows a color picker and passes a vec3 with components in the 0-1 range to the shader.

initial-value

The initial value of the shader uniform, in the format the shader uses it (i.e. 0-1 range for percent parameters). For color parameters, use a 3-element array, e.g. [1, 0, 0] for red.

uniform

WebGL only The name of the corresponding uniform in the shader. The uniform must be declared in GLSL with this name. It can use whichever precision you want, but the uniform type must be `vec3` for color parameters, otherwise `float`.

This only applies to WebGL shaders written in GLSL. The WebGPU renderer ignores this setting.

interpolatable

Set to `true` so the property can be supported by timelines.

Writing shaders

Construct supports rendering with both WebGL and the newer WebGPU. However these technologies use different shader languages: WebGL uses GLSL, and WebGPU uses WGSL. To support both renderers your effect will need to provide both a GLSL and WGSL shader which both render equivalently.

For more details on writing shaders, see WebGL shaders for GLSL-specific information, and see WebGPU shaders for WGSL-specific information.

You should test your shader works with both renderers. Change the *Enable WebGPU* setting in the *Advanced* section of Project Properties to test both renderers. You can also change the *Enable WebGPU in editor* setting in Construct's Settings dialog to test both renderers with the editor's rendering in the Layout View.

WEBGL SHADERS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/configuring-effects/webgl-shaders

Effect addons that support WebGL must provide a shader written in WebGL's shading language GLSL. This section provides information specific to WebGL shaders.

Adding a WebGL 2 shader variant

All shaders written for WebGL 1 (using GLSL ES 1.0) are compatible with both WebGL 1 and WebGL 2. There is no need to write a WebGL 2 variant of a shader unless you need specific features only available with WebGL 2 (using GLSL ES 3.0).

If you do write a WebGL 2 shader, we strongly recommend still providing a WebGL 1 shader. Do whatever you can to support WebGL 1, perhaps by using WebGL 1 extensions (see the following section for more details), or use a fallback like a low quality version, a glitchy version, or even just output transparency so it doesn't render. If you don't provide a WebGL 1 shader at all, then any project using your shader will cause an error on devices that still only support WebGL 1, with the project failing to load and just displaying a blank screen.

Providing a WebGL 2 shader variant

To provide a WebGL 2 shader variant, ensure `"webgl2"` is listed in the `"supported-renderers"` property of addon.json, e.g.:

```
"supported-renderers": ["webgl", "webgl2"]
```

This tells Construct to look for both a WebGL 1 and WebGL 2 shader for your addon.

The WebGL 1 shader is still in the file `effect.fx` as before. If enabled then the file `effect.webgl2.fx` specifies the shader to load for WebGL 2. A sample of an effect using both WebGL 1 and WebGL 2 shaders is provided in the effect SDK download.

Writing WebGL 2 shaders

WebGL 2 shaders are written using GLSL ES 3.0, as opposed to GLSL ES 1.0 for WebGL 1 shaders. This documentation does not cover the full details of how to write WebGL shaders - there are lots of other resources across the web covering that. However some key points to note when writing a WebGL 2 shader are:

- A WebGL 2 shader MUST start with the line `#version 300 es`. This must be the first line - no comments or other lines are allowed before it.
- Change `varying` to `in` for the `vTex` declaration.

- `gl_FragColor` is not used in WebGL 2 shaders. Instead declare `out lowp vec4 outColor;` at the top level and assign the result color to that.
- The `texture2D()` function for sampling a texture is now just `texture()` with WebGL 2.

Once adapted you can then make use of WebGL 2 shader features, such as `dFdx()`, `dFdy()` and `textureGrad()`.

Using WebGL 1 extensions

When only WebGL 1 is supported, Construct unconditionally activates the following extensions if supported:

- EXT_frag_depth
- OES_standard_derivatives
- EXT_shader_texture_lod

If your WebGL 2 shader uses equivalent features, this means you can sometimes support WebGL 1 too by activating them in your WebGL 1 shader, e.g.:

```
#extension GL_EXT_frag_depth : enable
#extension GL_EXT_shader_texture_lod : enable
#extension GL_OES_standard_derivatives : enable

// now you can use gl_FragDepthEXT, dFdx, dFdy, texture2DGradEXT etc.
```

Note Construct currently doesn't support any way to provide an alternative WebGL 1 shader when these extensions are not supported. However this approach lets you support more devices as instead of requiring WebGL 2, your shader can work with WebGL 1 as well when the necessary extensions are available.

WebGL 2 does not support these extensions as they are built-in features with WebGL 2. You cannot write just a WebGL 1 shader using those extensions, as it won't work with WebGL 2. Instead you must write a WebGL 2 shader variant.

Testing

The Construct editor provides a setting to force the editor and preview to run with WebGL 1. This can help you test your shader variants with both WebGL 1 and WebGL 2 (assuming your device supports WebGL 2). Note this option exists for shader testing only - exported projects will continue to use WebGL 2 when available regardless of the editor setting.

Shader uniforms

Shaders are written in a GLSL (OpenGL Shading Language) ES 1.0 fragment shader and interpreted by the browser's WebGL implementation. As with normal fragment shaders, the output is written to the special `gl_FragColor` variable. A WebGL 2 shader variant can be provided which must be written in GLSL ES 3.0 which has a number of differences; see the previous section on adding a WebGL 2 shader variant for more details.

The current foreground texture co-ordinate is provided in the special variable `vTex`. This is normally used to read the foreground texture, but it is actually optional (in case you want to write a shader that generates all of its output without reference to the foreground texture at all). All other uniforms are optional, and are documented below. The full uniform declaration is included with the recommended precision.

uniform lowp sampler2D samplerFront;

The foreground texture sampler, to be sampled at `vTex`.

uniform mediump vec2 srcStart;

uniform mediump vec2 srcEnd;

The current foreground rectangle being rendered, in texture co-ordinates. Note this is clamped as the object reaches the edge of the viewport. These are mainly useful for calculating the background sampling position.

uniform mediump vec2 srcOriginStart;

uniform mediump vec2 srcOriginEnd;

The current foreground source rectangle being rendered, in texture co-ordinates. This is not clamped, so can cover a rectangle that leaves the viewport. These are mainly useful for calculating the current sampling position relative to the object being rendered, without changing as the object clips against the viewport.

uniform mediump vec2 layoutStart;

uniform mediump vec2 layoutEnd;

The current foreground source rectangle being rendered, in layout co-ordinates. This allows the current fragment's position in the layout to be calculated.

uniform lowp sampler2D samplerBack;

The background texture sampler used for background-blending effects. The `blends-background` property in addon.json should also be set to `true` before using this. For the correct way to sample the background, see the next section.

uniform lowp sampler2D samplerDepth;

The depth texture sampler used for depth-based effects. The `uses-depth` property in addon.json should also be set to `true` before using this. The depth texture is the same size as the background texture, so this is sampled similarly to `samplerBack`. See the next section for more details.

---

uniform mediump vec2 destStart;

uniform mediump vec2 destEnd;

The current background rectangle being rendered to, in texture co-ordinates, for background-blending effects. For the correct way to sample the background, see the next section.

---

uniform highp float seconds;

The time in seconds since the runtime started. This can be used for animated effects. The `animated` property in addon.json should be set to `true`.

> *Note `highp` can only be used on certain platforms. To work across all systems, check `#ifdef GL_FRAGMENT_PRECISION_HIGH` to see if `highp` is supported, else fall back to using `mediump`.*

---

uniform mediump vec2 pixelSize;

The size of a texel in the foreground texture in texture co-ordinates. This allows calculating distances in pixels rather than texture co-ordinates.

---

uniform mediump float layerScale;

The current layer scale as a factor (i.e. 1 is unscaled). This is useful to ensure effects scale according to zoom.

---

uniform mediump float layerAngle;

The current layer angle in radians.

---

uniform mediump float devicePixelRatio;

The value of devicePixelRatio in the browser, which is the number of device pixels per CSS pixel. This may be necessary in some effects to handle high-DPI displays.

---

uniform mediump float zNear;

uniform mediump float zFar;

The values of the project properties *Near distance* and *Far distance*, which represent the distance of the near and far planes from the camera position.

Useful shader calculations

Some common calculations done with the available uniforms are listed below.

To sample the foreground pixel:

```
lowp vec4 front = texture2D(samplerFront, vTex);
```

To sample an adjacent pixel, offset by the pixel size:

```
// sample next pixel to the right
lowp vec4 next = texture2D(samplerFront, vTex + vec2(pixelSize.x, 0.0));
```

To calculate the position to sample the background, find the normalised position `n` of `vTex` in the foreground rectangle, and apply that to the background rectangle:

```
mediump vec2 n = (vTex - srcStart) / (srcEnd - srcStart);
lowp vec4 back = texture2D(samplerBack, mix(destStart, destEnd, n));
```

Sampling the depth buffer works similarly to sampling the background, but only provides one component, so just read the `r` value. Note that the value in the depth buffer is normalized (0-1 range) and does not linearly correspond to distance. To get a linearized Z value for a depth sample, use the calculation below, which uses the `zNear` and `zFar` uniforms.

```
mediump vec2 n = (vTex - srcStart) / (srcEnd - srcStart);
mediump float depthSample = texture2D(samplerDepth, mix(destStart, destEnd, n)).r;
mediump float zLinear = zNear * zFar / (zFar + depthSample * (zNear - zFar));
```

To calculate the current texture co-ordinate relative to the object being rendered, without being affected by clipping at the edge of the viewport, use the original source rectangle:

```
mediump vec2 srcOriginSize = srcOriginEnd - srcOriginStart;
mediump vec2 n = ((vTex - srcOriginStart) / srcOriginSize);
```

To calculate the current layout co-ordinates being rendered, add an extra step to interpolate `n` across the layout rectangle:

```
mediump vec2 srcOriginSize = srcOriginEnd - srcOriginStart;
mediump vec2 n = ((vTex - srcOriginStart) / srcOriginSize);
mediump vec2 l = mix(layoutStart, layoutEnd, n);
```

Construct renders using premultiplied alpha. Often it is convenient to modify the RGB components without premultiplication. To do this, divide by alpha to unpremultiply the color, but be sure not to divide by zero.

```
lowp vec4 front = texture2D(samplerFront, vTex);
lowp float a = front.a;

// unpremultiply
if (a != 0.0)
        front.rgb /= a;

// ...modify unpremultiplied front color...

// premultiply again
front.rgb *= a;
```

WEBGPU SHADERS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/configuring-effects/webgpu-shaders

Effect addons that support WebGPU must provide a shader written in WebGPU's shading language WGSL. This section provides information specific to WebGPU shaders.

Providing a WGSL shader variant

To provide a WGSL shader variant for WebGPU, the `"supported-renderers"` property in addon.json must specify `"webgpu"`, e.g.:

```
"supported-renderers": ["webgl", "webgpu"]
```

Note that the supported renderers can also include `"webgl2"` if your shader also uses a WebGL 2 shader variant.

This tells Construct that the effect also supports WebGPU, and it will look for a WGSL shader file with the name *effect.wgsl*.

Writing WGSL shaders

WGSL is a substantially different shader language to GLSL. This documentation does not cover the full details of the shader language. However there are two significant differences to GLSL shaders to note when writing WGSL shaders in Construct:

1. As WebGPU is a lower-level API than WebGL, WGSL shaders tend to be more verbose than GLSL, and also need to explicitly specify engine-specific details like binding and group numbers. To avoid hard-coding details that may change in future in to WGSL shaders, Construct provides a simple preprocessor based on tokens of the form `%%NAME%%`. These are not part of the WGSL language but are Construct-specific placeholders that Construct will replace with WGSL attributes and code. A full list of the supported placeholders is included below.
2. Effect parameters are stored in a struct and have no name associated with them in WGSL (they are referenced by a byte offset). Therefore they ignore the *uniform* property for the parameter set in addon.json. Instead just list all effect parameters in the `ShaderParams` struct in the order they are defined and with the appropriate type, and Construct will automatically calculate their byte offsets and update them accordingly.

Construct-specific placeholders

Here is a list of placeholders of the form `%%NAME%%` that Construct will replace in WGSL shaders.

%%SAMPLERFRONT_BINDING%%

%%TEXTUREFRONT_BINDING%%

Replaced with the `@binding` and `@group` attributes for the foreground sampler and texture. Example usage:
`%%SAMPLERFRONT_BINDING%% var samplerFront : sampler;`
`%%TEXTUREFRONT_BINDING%% var textureFront : texture_2d<f32>;`

%%SAMPLERBACK_BINDING%%

%%TEXTUREBACK_BINDING%%

Replaced with the `@binding` and `@group` attributes for the background sampler and texture. Example usage:
`%%SAMPLERBACK_BINDING%% var samplerBack : sampler;`
`%%TEXTUREBACK_BINDING%% var textureBack : texture_2d<f32>;`

%%SAMPLERDEPTH_BINDING%%

%%TEXTUREDEPTH_BINDING%%

Replaced with the `@binding` and `@group` attributes for the depth sampler and texture for depth effects like fog. Example usage:
`%%SAMPLERDEPTH_BINDING%% var samplerDepth : sampler;`
`%%TEXTUREDEPTH_BINDING%% var textureDepth : texture_depth_2d;`

%%FRAGMENTINPUT_STRUCT%%

Defines the `FragmentInput` structure used as input to the fragment shader method. This structure defines `fragUV : vec2<f32>` as the current fragment texture co-ordinates (equivalent to `vTex` in GLSL shaders). It also defines `@builtin(position) fragPos : vec4<f32>` and two utility methods that use it (see below).

%%FRAGMENTOUTPUT_STRUCT%%

Defines the `FragmentOutput` structure returned from the fragment shader method. This structure defines `color : vec4<f32>` which is used to write the output color from the shader (equivalent to writing to `gl_FragColor` in WebGL 1 shaders).

%%SHADERPARAMS_BINDING%%

Replaced with the `@binding` and `@group` attributes for the structure containing custom effect parameters. This structure must be defined by your shader matching the effect parameters in the same order. It can be omitted if the shader does not use any custom parameters. Example usage from the 'Set color' sample shader:

```
struct ShaderParams {
	setColor : vec3<f32>
};
%%SHADERPARAMS_BINDING%% var<uniform> shaderParams : ShaderParams;
```

%%C3PARAMS_STRUCT%%

Defines a structure named `c3Params` which contains members that correspond to the Construct-provided uniforms for WebGL shaders, as well as a set of utility methods. The members of the structure currently include:

```
        srcStart                        : vec2<f32>,
        srcEnd                          : vec2<f32>,
        srcOriginStart          : vec2<f32>,
        srcOriginEnd            : vec2<f32>,
        layoutStart                     : vec2<f32>,
        layoutEnd                       : vec2<f32>,
        destStart                       : vec2<f32>,
        destEnd                         : vec2<f32>,
        devicePixelRatio        : f32,
        layerScale                      : f32,
        layerAngle                      : f32,
        seconds                         : f32,
        zNear                           : f32,
        zFar                            : f32,
        isSrcTexRotated         : u32
```

%%C3_UTILITY_FUNCTIONS%%

Defines a set of utility functions that are useful for many kinds of effects (see below)

Utility functions

Some placeholders also include definitions for useful helper functions that perform common tasks in shaders. The available functions are documented below.

Provided by %%FRAGMENTINPUT_STRUCT%%

fn c3_getBackUV(fragPos : vec2<f32>, texBack : texture_2d<f32>) -> vec2<f32>

Helper function to calculate the texture co-ordinates to sample the background texture at for background blending effects. Example: `c3_getBackUV(input.fragPos.xy, textureBack)`

fn c3_getDepthUV(fragPos : vec2<f32>, texDepth : texture_depth_2d) -> vec2<f32>

Helper function to calculate the texture co-ordinates to sample the depth texture at for depth-processing effects. Example: `c3_getDepthUV(input.fragPos.xy, textureDepth)`

Provided by %%C3PARAMS_STRUCT%%

fn c3_srcToNorm(p : vec2<f32>) -> vec2<f32>

fn c3_normToSrc(p : vec2<f32>) -> vec2<f32>

fn c3_srcOriginToNorm(p : vec2<f32>) -> vec2<f32>

fn c3_normToSrcOrigin(p : vec2<f32>) -> vec2<f32>

Pass `input.fragUV` to `c3_srcToNorm()` to return a position normalized in the range [0, 1] relative to the box `srcStart` to `srcEnd`. The `c3_normToSrc()` function performs the reverse calculation. The `srcOrigin` variants work relative to the box `srcOriginStart` to `srcOriginEnd` instead.

fn c3_clampToSrc(p : vec2<f32>) -> vec2<f32>

fn c3_clampToSrcOrigin(p : vec2<f32>) -> vec2<f32>

Clamps a given position to the box `srcStart` to `srcEnd` or `srcOriginStart` to `srcOriginEnd`.

fn c3_getLayoutPos(p : vec2<f32>) -> vec2<f32>

Pass `input.fragUV` to calculate the current corresponding position in layout co-ordinates.

fn c3_srcToDest(p : vec2<f32>) -> vec2<f32>

Maps a texture co-ordinate in the `srcStart` to `srcEnd` rectangle to the corresponding position in the `destStart` to `destEnd` rectangle.

fn c3_clampToDest(p : vec2<f32>) -> vec2<f32>

Clamps a texture co-ordinate to the `destStart` to `destEnd` rectangle.

fn c3_linearizeDepth(depthSample : f32) -> f32

Linearize a sample from the depth texture to a Z distance. Depth texture samples are usually in a normalized range [0, 1]; this method returns a Z distance based on the near and far planes, which is a more useful number for things like fog effects.

Provided by %%C3_UTILITY_FUNCTIONS%%

fn c3_premultiply(c : vec4<f32>) -> vec4<f32>

fn c3_unpremultiply(c : vec4<f32>) -> vec4<f32>

Premultiplies the RGB components by the A component in a color, and the reverse operation.

fn c3_grayscale(rgb : vec3<f32>) -> f32

Convert RGB colors to a corresponding grayscale component.

fn c3_getPixelSize(t : texture_2d<f32>) -> vec2<f32>

Returns the size of a pixel in texture co-ordinates on the given texture.

> *This uses the `textureDimensions()` WGSL built-in, and can be used as a replacement for the `pixelSize` uniform in the WebGL renderer. It is further also capable of determining the pixel size for any given texture.*

fn c3_RGBtoHSL(color : vec3<f32>) -> vec3<f32>

fn c3_HSLtoRGB(hsl : vec3<f32>) -> vec3<f32>

Converts RGB values to the equivalent in HSL, and the reverse operation.

Useful shader calculations

Some common calculations done in WGSL shaders are listed below.

To sample the foreground pixel:

```
var front : vec4<f32> = textureSample(textureFront, samplerFront, input.fragUV);
```

To sample an adjacent pixel, offset by the pixel size:

```
// get width of a pixel in texture co-ordinates
var pixelWidth : f32 = c3_getPixelSize(textureFront).x;

// sample next pixel to the right
var next : vec4<f32> = textureSample(textureFront, samplerFront, input.fragUV + vec2<f32>(pixelW
```

To calculate the position to sample the background, use the `c3_getBackUV()` helper function:

```
var back : vec4<f32> = textureSample(textureBack, samplerBack, c3_getBackUV(input.fragPos.xy, te
```

Sampling the depth buffer works similarly to sampling the background, but using the `c3_getDepthUV()` helper function on the depth texture and sampler. It's commonly useful to then linearize the resulting depth sample to a Z distance based on the near and far planes, which the `c3_linearizeDepth()` helper function does.

```
// sample depth buffer
var depthSample : f32 = textureSample(textureDepth, samplerDepth, c3_getDepthUV(input.fragPos.xy

// linearize depth sample to Z distance
var zLinear : f32 = c3_linearizeDepth(depthSample);
```

To calculate the current texture co-ordinate relative to the object being rendered, without being affected by clipping at the edge of the viewport, use the `c3_srcOriginToNorm()` helper method:

```
var n : vec2<f32> = c3_srcOriginToNorm(input.fragUV);
```

To calculate the current layout co-ordinates being rendered, use the `c3_getLayoutPos()` helper method:

```
var l : vec2<f32> = c3_getLayoutPos(input.fragUV);
```

Construct renders using premultiplied alpha. Often it is convenient to modify the RGB components without premultiplication. To do this, use the `c3_unpremultiply()` and `c3_premultiply()` helper methods:

```
// sample front texture
var front : vec4<f32> = textureSample(textureFront, samplerFront, input.fragUV);

// unpremultiply
front = c3_unpremultiply(front);

// ...modify unpremultiplied front color...

// premultiply again
front = c3_premultiply(front);
```

Precision in WebGPU shaders

WebGL shaders allow the use of shader precision qualifiers such as `lowp` and `mediump`. WebGPU uses a different approach with explicit types such as `f32`. Some devices support a lower-precision `f16` type if they support the `shader-f16` feature. To help make it easy to use the `f16` type, Construct requests to use the `shader-f16` feature where supported, and defines a type in WebGPU shaders named `f16or32`, which is `f16` when `shader-f16` is supported, otherwise it is `f32`.

Currently all of Construct's built-in inputs, outputs and library functions use the `f32` type exclusively for broadest compatibility. However shaders can make use of the `f16or32` type for their internal calculations, converting to `f32` where necessary, to help improve shader performance, especially as `f32` is high precision with a higher performance cost compared to `lowp` or `mediump` precision in GLSL.

Compatibility differences with WebGL shaders

Due to API differences between WebGL and WebGPU, the WebGL src/srcOrigin/dest uniforms use an inverted Y direction. This means instead of ranging from 0-1 for top-to-bottom, they range from 1-0.

Sometimes this does not have any impact on the effect. However in some cases it does, depending on the kinds of calculation done in the shader. When porting a GLSL shader to WGSL, you may need to emulate the inverted Y direction in WGSL to achieve the same effect. For example the Lens2 effect uses the following code pattern in WGSL to emulate the inverted Y direction:

```
// At start of shader: get normalized source co-ordinates
// and then invert Y direction to match WebGL
var tex : vec2<f32> = c3_srcToNorm(input.fragUV);
tex.y = 1.0 - tex.y;

// ... rest of effect ...

// At end of shader: invert Y direction again and then
// calculate background sampling position
p.y = 1.0 - p.y;

var output : FragmentOutput;
output.color = textureSample(textureBack, samplerBack, mix(c3Params.destStart, c3Params.destEnd
```

If you are writing a new effect, consider writing the WebGPU shader first, and then if necessary applying the Y inversion in the WebGL shader instead. As WebGPU is the newer technology, in the long term the WebGL renderer may eventually be retired, in which case it is better to have a natural code style in the WGSL shader.

DEFINING ACTIONS, CONDITIONS AND EXPRESSIONS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/defining-aces

The set of actions, conditions and expressions which are available in your addon is defined in **aces.json**. The term "ACE" is used to refer to an action, condition or expression in general. ACEs are grouped by category. The overall structure of the file format is as follows.

```
{
	"category1": {
		"conditions": [
			condition1,
			condition2,
			...
		],
		"actions": [
			action1,
			action2,
			...
		],
		"expressions": [
			expression1,
			expression2,
			...
		]
	},
	"category2": {
		"conditions": [
			...
		],
		"actions": [
			...
		],
		"expressions": [
			...
		]
	}
}
```

Note that Construct 2 uses numbers for IDs, whereas Construct 3 uses strings. These strings are then also used to identify related language strings in the language file.

JSON schema

The addon SDK provides a JSON schema to help you write aces.json files, as it provides autocomplete and validation in compatible editors. Construct will ignore a `"$schema"` property at the top level instead of interpreting it as a category to help make it easy to use the schema.

Never delete ACES after release

> *Once you have released your addon,* ***never delete any actions, conditions or expressions from it****. This will corrupt everyone's projects that use your addon, because Construct will no longer be able to find the deleted action, condition or expression in your addon. Instead mark the features deprecated so they are hidden.*

Categories

Each category key is the category ID. This is not displayed in the editor; the string to display is looked up in the language file.

For behaviors only, a default category of an empty string may be used. This category will use the behavior name. Other categories may still be used, in which case Construct 3 will append the category name after the behavior name, e.g. "MyBehaviorName: My category".

Common properties of ACE definitions

Each entry in the `"conditions"`, `"actions"` and `"expressions"` arrays is a JSON object which defines a single condition, action or expression. An example minimal condition definition for the System *Every Tick* condition is shown below.

```
{
        "id": "every-tick",
        "scriptName": "EveryTick"
}
```

The `id` and `scriptName` are the only required properties for conditions and actions. Expressions require `id`, `expressionName` and `returnType`. All other properties are optional.

The definitions for conditions, actions and expressions all share a few common properties. These are detailed below. Then the properties specific to each kind is documented after that.

---

id

A string specifying a unique ID for the ACE. This is used in the language file. By convention this is lowercase with dashes for separators, e.g. "my-condition".

---

c2id

If you are porting a Construct 2 addon to Construct 3, put the corresponding numerical ID that the Construct 2 addon used here. This allows Construct 3 to import Construct 2 projects using your addon.

scriptName / expressionName

The name of the function in the runtime script for this ACE. Note for expressions, use `expressionName` instead, which also defines the name typed by the user in expressions.

isDeprecated

Set to true to deprecate the ACE. This hides it in the editor, but allows existing projects to continue using it.

highlight

Set to true to highlight the ACE in the condition/action/expression picker dialogs. This should only be used for the most regularly used ACEs, to help users pick them out from the list easily.

params

An array of parameter definitions. See the section below on parameters. This can be omitted if the ACE does not use any parameters.

Condition definitions

Condition definitions can also use the following properties.

isTrigger

Specifies a trigger condition. This appears with an arrow in the event sheet. Instead of being evaluated every tick, triggers only run when they are explicity triggered by a runtime call.

isFakeTrigger

Specifies a fake trigger. This appears identical to a trigger in the event sheet, but is actually evaluated every tick. This is useful for conditions which are true for a single tick, such as for APIs which must poll a value every tick.

isStatic

Normally, the condition runtime method is executed once per picked instance. If the condition is marked static, the runtime method is executed once only, on the object type class. This means the runtime method must also implement the instance picking entirely itself, including respecting negation and OR blocks.

isLooping

Display an icon in the event sheet to indicate the condition loops. The condition method should use ILoopingConditionContext to implement its loop.

isInvertible

Allow the condition to be inverted in the event sheet. Set to `false` to disable invert.

isCompatibleWithTriggers

Allow the condition to be used in the same branch as a trigger. Set to `false` if the condition does not make sense when used in a trigger, such as the *Trigger once* condition.

Action definitions

Action definitions can also use the following properties.

isAsync

Set to `true` to mark the action as asynchronous. Make the action method an `async` function, and the system *Wait for previous actions to complete* action will be able to wait for the action as well.

Expression definitions

Expressions work slightly differently to conditions and actions: they must specify a `returnType`, and instead of using a `scriptName` they specify an `expressionName` which doubles as both what is typed for the expression as well as the runtime script function name.

returnType

One of `"number"`, `"string"`, `"any"`. The runtime function must return the corresponding type, and `"any"` must still return either a number or a string.

isVariadicParameters

If `true`, Construct 3 will allow the user to enter any number of parameters beyond those defined. In other words the parameters (if any) listed in `"params"` are required, but this flag enables adding further `"any"` type parameters beyond the end.

Parameter definitions

ACEs can all define which parameters they use with the `"params"` property. This property should be set to an array of parameter definition objects. Below shows an example for the System *Compare two values* condition.

```
{
	"id": "compare-two-values",
	"scriptName": "Compare",
	"params": [
		{	"id": "first-value",	"type": "any" },
		{	"id": "comparison",		"type": "cmp" },
		{	"id": "second-value",	"type": "any" }
	]
}
```

Note that expressions can only use `"number"`, `"string"` or `"any"` parameter types.

id

A string with a unique identifier for this parameter. This is used to refer to the parameter in the language file.

c2id

In some circumstances, it is necessary to specify which Construct 2 parameter ID a parameter corresponds to. However normally it can be inferred by the parameter index.

type

The parameter type. Expressions can only use `"number"`, `"string"` or `"any"`. However conditions and actions have the following options available:

- `"number"` — a number parameter
- `"string"` — a string parameter
- `"any"` — either a number or a string
- `"boolean"` — a boolean parameter, displayed as a checkbox
- `"combo"` — a dropdown list. Items must be specified with the `"items"` property.
- `"combo-grouped"` — a dropdown list with grouped items (using optgroup elements). Item groups must be specified with the `"itemGroups"` property.
- `"cmp"` — a dropdown list with comparison options like "equal to", "less than" etc.

- `"object"` — an object picker. The types of plugin to show can be filtered using an optional `"allowedPluginIds"` property.
- `"objectname"` — a string parameter which is interpreted as an object name
- `"projectfile"` — a dropdown list from which any project file in the project can be chosen. The parameter value at runtime is a relative path to fetch the project file from. The `"filter"` option can also be specified to filter the list by a file extension, e.g. `".txt"` to only list .txt files.
- `"layer"` — a string parameter which is interpreted as a layer name
- `"layout"` — a dropdown list with every layout in the project
- `"keyb"` — a keyboard key picker
- `"instancevar"` — a dropdown list with the non-boolean instance variables the object has
- `"instancevarbool"` — a dropdown list with the boolean instance variables the object has
- `"eventvar"` — a dropdown list with non-boolean event variables in scope
- `"eventvarbool"` — a dropdown list with boolean event variables in scope
- `"animation"` — a string parameter which is interpreted as an animation name in the object
- `"objinstancevar"` — a dropdown list with non-boolean instance variables available in a prior `"object"` parameter. Only valid when preceded by an `"object"` parameter.

initialValue

A string which is used as the initial expression for expression-based parameters. Note this is still a string for `"number"` type parameters. It can contain any valid expression for the parameter, such as "1 + 1". For `"boolean"` parameters, use a string of either `"true"` or `"false"`. For `"combo"` parameters, this is the initial item ID.

items

Only valid with the `"combo"` type. Set to an array of item IDs available in the dropdown list. The actual displayed text for the items is defined in the language file.

> *If you remove a combo item after publishing your addon, existing projects using that combo item will revert to the default item.*

---

itemGroups

Only valid with the `"combo-grouped"` type. Set to an array of item groups available in the dropdown list, with each group being represented by an object with properties `"id"` for the group ID, and `"items"` being an array of strings of item IDs in the group. The actual displayed text for the groups and items is defined in the language file.

```
// example "combo-grouped" parameter definition
{
	"id": "dinosaur",
	"type": "combo-grouped",
	"itemGroups": [{
		"id": "theropods",
		"items": ["tyrannosaurus", "velociraptor", "deinonychus"]
	}, {
		"id": "sauropods",
		"items": ["diplodocus", "saltasaurus", "apatosaurus"]
	}
}
```

---

allowedPluginIds

Optional and only valid with the `"object"` type. Set to an array of plugin IDs allowed to be shown by the object picker. For example, use `["Sprite"]` to only allow the object parameter to select a Sprite.

---

filter

Optional and only valid with the `"projectfile"` type. Set to a file extension including the dot to filter the list of provided project files to only those with the given file extension, e.g. `".txt"`.

---

autocompleteId

Optional and only valid with the `"string"` type. Set to a globally unique ID and string constants with the same ID will offer autocomplete in the editor. This is useful for "tag" parameters. Note the ID must be unique to all other plugins and behaviors in Construct, so it is a good idea to include your plugin or behavior name in the string, e.g. "myplugin-tag".

Language strings

The aces.json file does not include any strings displayed in the editor UI. These are all kept in a separate language file to facilitate translation. Therefore to finish adding ACEs, the relevant UI strings like the list name and description must be added to the language file. See The language file for more information.

THE LANGUAGE FILE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/language-file

In the .c3addon file, **en-US.json** is the default language file containing all the addon's strings that are shown in the editor UI in a JSON format. Moving these strings to a separate file makes it possible for the software to be fully translated.

All strings must be provided in US English (hence the filename en-US.json) since this is the default language of Construct 3, and the common language from which all other languages are translated.

Overall structure

The overall structure of the language file is as follows, taken from the plugin SDK template. Note that in general, the language file uses an ID as a key on the left, and the string to display as the value on the right.

```
{
	"languageTag": "en-US",
	"fileDescription": "Strings for MyCustomPlugin.",
	"text": {
		"plugins": {
			"mycompany_myaddon": {
				"name": "My Custom Plugin",
				"description": "Description for my custom plugin.",
				"help-url": "https://www.scirra.com",
				"properties": {
					"test-property": {
						"name": "Test property",
						"desc": "A test number property. Displayed by 'Al
					}
				},
				"aceCategories": {
					"custom": "Custom"
				},
				"conditions": {
					"is-large-number": {
						"list-name": "Is large number",
						"display-text": "[i]{0}[/i] is a large number",
						"description": "Test if a number is greater than
						"params": {
							"number": {
								"name": "Number",
								"desc": "Number to test if greate
							}
						}
					}
				},
				"actions": {
					"do-alert": {
						"list-name": "Do alert",
						"display-text": "Do alert",
						"description": "Do a dummy alert."
					}
				},
				"expressions": {
					"double": {
						"description": "Double a number.",
						"translated-name": "Double",
						"params": {
							"number": {
								"name": "Number",
								"desc": "The number to double."
							}
						}
					}
				}
			}
		}
	}
}
```

The addon SDK provides a JSON schema to help you write language files, as it provides autocomplete and validation in compatible editors.

Required fields

`"languageTag"` must be "en-US".

`"fileDescription"` is not used in the editor. It provides a hint to translators. It can simply say "Strings for [addon name]".

`"text"` represents the root node of the string tree, and "plugins" represents strings for all plugins. These should be left as they are.

`"mycompany_myaddon"` should be the **lowercase addon ID.** Note if your addon ID contains any uppercase characters, they should be lowercased for this key in the language file. The remaining strings all belong inside this key, since it represents your plugin.

`"name"` is the name of your plugin as it appears in the editor. This can be changed at any time, but the plugin ID should not be changed after release.

`"description"` is a short sentence or two describing what your plugin does.

`"help-url"` is a URL to documentation or support for your plugin.

Note: themes only need to use the `"name"`, `"description"` *and* `"help-url"` *fields.*

Strings for properties/parameters

For each PluginProperty your plugin uses, there must be a key with the property ID under `"properties"`. For effects, there must be a key with the parameter ID under a `"parameters"` instead, but it otherwise works the same. The required strings for each property are:

- `"name"` — the name of the property, which appears to the left of the field
- `"desc"` — the property description, which appears in the footer of the Properties Bar

For example given the following property:

```
new SDK.PluginProperty("integer", "test-property", 0)
```

The following language strings can be used under the `"properties"` key:

```
"test-property": {
	"name": "Test property",
	"desc": "A test number property. Displayed by 'Alert' action."
}
```

Some properties require additional keys.

Combo properties

The `"combo"` property type needs an extra `"items"` key to set the visible name of each item. Each key underneath this should be the ID of the combo item, and its value the name to use. Here is an example from the Audio plugin. The property is created as:

```
new SDK.PluginProperty("combo", "timescale-audio", {
	initialValue: "off",
	items: ["off", "sounds-only", "sounds-and-music"]
})
```

Note the items defined here are IDs rather than displayed strings. The strings to display are set in the language file like this:

```
"timescale-audio": {
	"name": "Timescale audio",
		"desc": "Choose whether the audio playback rate changes with the time scale.",
		"items": {
			"off": "Off",
			"sounds-only": "On (sounds only)",
			"sounds-and-music": "On (sounds and music)"
		}
	}
```

Link properties

The `"link"` property type needs an extra `"link-text"` key to set the text of the clickable link. An example is below.

```
"make-original-size": {
	"name": "Size",
	"desc": "Click to set the object to the same size as its image.",
	"link-text": "Make 1:1"
}
```

Category names

When defining ACEs, category IDs are used rather than category names. The `"aceCategories"` key defines the displayed name of each category. The following example displays all ACEs in the category ID `"customCategory"` as being in a section labelled `"My custom category"`.

```
"aceCategories": {
	"customCategory": "My custom category"
}
```

Category names are shared across actions, conditions and expressions.

ACE strings

Strings for conditions, actions and expressions are listed in the keys `"conditions"`, `"actions"` and `"expressions"` respectively. Note they are not sorted by category here; each section lists all ACEs of that type.

Similar to properties, each key under each section is the ID of the action, condition or expression. As with the definitions themselves, actions and conditions work slightly differently to expressions.

Action and condition strings

The required keys are:

- `"list-name"` — the name that appears in the condition/action picker dialog.

- `"display-text"` — the text that appears in the event sheet. You can use simple BBCode tags like `[b]` and `[i]`, and use `{0}`, `{1}` etc. as parameter placeholders. (There must be one parameter placeholder per parameter.) For behaviors only, the placeholder `{my}` is substituted for the behavior name and icon.
- `"description"` — a description of the action or condition, which appears as a tip at the top of the condition/action picker dialog.

Expression strings

The required keys are:

- `"description"` — the description that appears in the expressions dictionary, which lists all available expressions.
- `"translated-name"` — the translated name of the expression name. In the en-US file, this should simply match the expression name from the expression definition. This key mainly exists so it can be changed in other languages, making it possible to translate expressions in some contexts. Note when actually typing an expression the non-translated expression name must always be used.

Parameters

Actions, conditions and expressions can omit the `"params"` key if they have no parameters. However if they have any parameters this key must be present, and each parameter must have its own key inside with the ID of the parameter. Similar to the plugin properties, each key must have a `"name"` and `"desc"`. For `"combo"` parameters there must also be a property `"items"` (which work the same as `"combo"` property types: each item ID maps to its display text). For `"combo-grouped"` parameters, there must also be a property `"itemGroups"` (see below for an example).

Example

The *Is large number* condition in the plugin SDK uses the following definition in aces.json:

```
{
	"id": "is-large-number",
	"scriptName": "IsLargeNumber",
	"highlight": true,
	"params": [
		{
			"id": "number",
			"type": "number"
		}
	]
}
```

Its corresponding language strings are defined in en-US.json as follows:

```
"is-large-number": {
	"list-name": "Is large number",
	"display-text": "[i]{0}[/i] is a large number",
	"description": "Test if a number is greater than 100.",
	"params": {
		"number": {
			"name": "Number",
			"desc": "Number to test if greater than 100."
		}
	}
}
```

"combo-grouped" example

When using the `"combo-grouped"` parameter type, the definition in aces.json might look like this:

```
{
	"id": "dinosaur",
	"type": "combo-grouped",
	"itemGroups": [{
		"id": "theropods",
		"items": ["tyrannosaurus", "velociraptor", "deinonychus"]
	}, {
		"id": "sauropods",
		"items": ["diplodocus", "saltasaurus", "apatosaurus"]
	}
}
```

Its corresponding language strings can be defined in en-US.json inside the `"params"` key as follows:

```
"dinosaur": {
	"name": "Dinosaur",
	"desc": "Choose a dinosaur",
	"itemGroups": {
		"theropods": {
			"name": "Theropods",
			"items": {
				"tyrannosaurus": "Tyrannosaurus",
				"velociraptor": "Velociraptor",
				"deinonychus": "Deinonychus"
			}
		},
		"sauropods": {
			"name": "Sauropods",
                        "items": {
                                "diplodocus": "Diplodocus",
                                "saltasaurus": "Saltasaurus",
                                "apatosaurus": "Apatosaurus"
                        }
                }
        }
}
```

EDITOR SCRIPTS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/editor-scripts

Plugin and behavior addons have separate scripts that run in the context of the editor rather than the runtime (the Construct game engine).

Effects don't use editor scripts. They only provide shader code.

Most addons do not need complex editor scripts. However some editor features are available for things like specifying dependencies and importing assets. These are documented in the Editor API reference section of the Addon SDK manual.

Do not access the DOM in editor scripts

The editor DOM, including all HTML, CSS styles, and event handlers, are considered internal details. Do not develop addons that access or modify these in any way. Such addons risk breaking at any time, including permanently breaking with no workaround, and in this event Scirra will not provide support. In future editor addons are likely to be sandboxed, in which case all unsupported features will become unavailable anyway.

RUNTIME SCRIPTS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/runtime-scripts

Plugin and behavior addons have separate scripts that run in the context of the runtime (the Construct game engine) rather than the editor.

Effects don't use runtime scripts. They only provide shader code.

Runtime documentation

In Construct's addon SDK, runtime scripts are based on the same APIs as used by Construct's scripting feature. The APIs specific to the Addon SDK can be found in the Addon SDK interfaces section of the scripting reference in the Construct manual. Further, all the APIs in the rest of the scripting reference section of Construct's manual are also accessible to addons.

Using modules

A major feature of the Addon SDK is first-class support for modules. These allow you to use `import` and `export` in your addon's runtime scripts.

Configuring use of modules

By default the Addon SDK samples are all already configured to use modules. However for clarity, to use modules an addon must be configured as follows:

1. The editor plugin must call `this._info.SetRuntimeModuleMainScript("c3runtime/main.js")` to set the runtime script file *main.js* as the main script. Then this is the only script that Construct loads.
2. All other runtime scripts must then be imported in main.js, e.g. `import "./instance.js";`
3. Add main.js to the `file-list` in addon.json so it works in developer mode

If an addon does not call `SetRuntimeModuleMainScript()`, then Construct automatically generates a main script that imports every runtime file. However this is not the right way to use modules if you want to do something like `import` a module in one of your existing runtime files.

Adding a module script

Assuming the addon is already configured to use modules - which all the Addon SDK samples are - then if you want to import a new module script named *mymodule.js*, these are the steps to follow.

1. Create the file *c3runtime/mymodule.js* and write an `export` in it.

2. In the editor plugin/behavior script, call `this._info.AddC3RuntimeScript("c3runtime/mymodule.js");` to add it as a runtime script, so the editor knows the file exists.
3. In addon.json add *c3runtime/mymodule.js* to `"file-list"` so the file is available in developer mode.
4. Import the new module in an existing runtime script. For example at the top of instance.js you could write: `import * as MyModule from "./mymodule.js";`

In short, when an addon is configured to use modules, all you need to do to add a new module script is to add it as a runtime file and then `import` it somewhere.

DOM calls in the C3 runtime

A major architectural feature of the runtime is the ability to host the runtime in a dedicated worker, off the main thread. In this mode it renders using OffscreenCanvas. With the modern web platform, many functions and classes are available in dedicated workers, and more are being added over time. Refer to this MDN guide on available APIs in workers.

Providing your addon's runtime calls only use APIs available in a worker, such as `fetch()` or IndexedDB, then it will not need any changes to support a worker. However if it does use APIs not normally available in a worker, then it will need some changes.

The principle behind making calls to the main thread in the C3 runtime is to split the runtime scripts in to two halves: the runtime side (that runs in the worker), and the DOM side (that runs on the main thread where the document is). The DOM side has full access to all browser APIs. The runtime side can issue DOM calls using a specially-designed messaging API built in to the runtime. Essentially instead of making a call, your addon can post a message with parameters for the call to the script on the DOM side, where the API call is really made. The DOM side can then send a message back with a result, or send messages to the runtime on its own, such as in response to events. The messaging APIs make this relatively straightforward. However one consequence to note is that a synchronous API call will become asynchronous, since the process of messaging to or from a worker is asynchronous.

Once this approach is used, there is no need to change anything to support the normal (non-Worker) mode. In this case both scripts will run in the same context and the messaging API will just forward messages within the same context too. Therefore this one approach covers both cases, and ensures code works identically regardless of whether the runtime is hosted in the main thread or a worker.

Using a DOM script

By default Construct 3 assumes no DOM scripts are used. If you want to use one, use the following call on IPluginInfo to enable one:

```
this._info.SetDOMSideScripts(["c3runtime/domSide.js"]);
```

Since an array of script paths is used, if you have a lot of DOM code, you can split it across different files. Don't forget to add these files to the file list in `addon.json`.

For documentation on the DOM messaging APIs, refer to DOMElementHandler (used in domSide.js), ISDKDOMPluginBase (used in plugin.js), and ISDKDOMInstanceBase (used in instance.js).

For an example demonstrating how to get started, see the **domElementPlugin** template in the C3 plugin SDK download. This demonstrates using the above APIs to create a simple `<button>` element in the DOM with a custom button text, and firing an *On clicked* trigger, with support for running in a Web Worker.

Supporting the debugger

Plugins and behaviors can display custom properties in the debugger by overriding the `_getDebuggerProperties()` method of the instance class. It should return an array of property sections of the form `{ title, properties }`, where `title` is a string of the section title and `properties` is an array of property objects. Each property object is of the form `{ name, value }` with an optional `onedit` callback. The name must be a string, and the value can be a string, number or boolean.

Editing properties

If an `onedit` callback is omitted, the debugger displays the property as read-only. If it is provided, the debugger allows the property to be edited. If it is changed, the callback is run with the new value as a parameter.

In many cases, editing a property does the equivalent of an action. To conveniently manage your code, you can implement actions as methods on your instance class, and call the same method from both the action and the debugger edit handler. As a public method is also accessible from Construct's scripting feature, it also makes the feature accessible from JavaScript code in projects.

Translation

By default, property section titles and property names are interpreted as language string keys. This allows them to be translated by looking them up in your addon's language file. Note property values do not have any special treatment. You can bypass the language file lookup by prefixing the string with a dollar character `$`, e.g. the property name `"plugins.sprite.debugger.foo"` will look up a string in the language file, but `"$foo"` will simply display the string `foo`.

The debugger runs in a separate context to the editor, and as such not all language strings are available. The language keys available in the debugger are:

- The addon name
- All property names

- All combo property items
- Everything under the "debugger" key

In general, if you need a language string for the debugger, simply place it under the "debugger" key, e.g. at "plugins.sprite.debugger.foo".

Sample code

The following code is used by the Sprite plugin to display its animation-related debugger properties. Notice how it uses language keys and calls actions to update properties.

```
_getDebuggerProperties()
{
	const prefix = "plugins.sample-plugin.debugger";
	return [{
		title: prefix + ".title",
		properties: [
			{name: prefix + ".speed",	value: this.speed,	onedit: v => this
			{name: prefix + ".angle",	value: this.angle,	onedit: v => this
		]
	}];
}
```

TIMELINE INTEGRATION

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/timeline-integration

Adding timeline support to a 3rd party addon, be it a plugin, behavior or effect is quite easy. A little bit of extra work is needed though, here is how to do it.

Plugins

1. Set the interpolatable plugin property option to true in all the plugin properties which should be supported by timelines.
2. Implement the GetPropertyValueByIndex(index) method in the plugin instance class.

   index argument

   Refers to the index of each property in the plugin as they are given to the constructor of the plugin instance class.

   return value

   The current value associated with the passed in index. Depending on the current implementation this could be as easy as returning an existing variable.

   Colors*: color properties should be returned as an array of 3 values. If the internal representation used by the plugin is different, the conversion needs to be made before returning the color.*

   Angles*: if a plugin uses a value as an angle, but it is not specifically defined as an angle in the plugin definition, this method needs to make sure the value is in the same format as it is shown in the editor before returning it.* ***Ex.*** *A plugin internally converting a property from degrees to radians, should make sure the value returned by* ***GetPropertyValueByIndex*** *is in degrees.*

3. Implement the SetPropertyValueByIndex(index, value) method in the plugin instance class.

   index argument

   Refers to the index of each property in the plugin as they are given to the constructor of the plugin instance class.

   value argument

The new value that needs to be applied to the specified property. The passed in value is absolute so it should be applied directly with the = operator.

return value

No return value is required.

Depending on the implementation, additional work might be needed when a property changes. ***Ex.*** *If the plugin relies on any sort of caching relating to a property, changing that property*
would require an update to the cache in order for everything to continue working properly as the plugin is modified by a timeline.

Colors*: these values are received as arrays of 3 values. The values should be applied according to the plugin's internal representation.*

Angles*: if a plugin uses a value as an angle, but it is not specifically defined as angle in the plugin definition. This method needs to make sure the incoming value is using the same format as the internal representation before making the assignment.* ***Ex.*** *A plugin internally converting a property from degrees to radians, will need to convert the incoming value of* ***SetPropertyValueByIndex*** *from degrees into radians.*

Plugin that need layout view preview updates

Some plugins might need to update the layout view to give a preview of the changes they are making. In this case a few more methods needs to be implemented so the plugin can update it's internal state when a timeline starts preview and when it stops preview.

1. Implement the OnTimelinePropertyChanged (id, value, detail) method in the plugin instance class.

id argument

The id of the property that is changing.

value argument

The value that is being applied by the timeline.

detail argument

An object with details about the value. It has a **"resultMode"** property with a value of either **"absolute"** or **"relative"**.

*"**value**" and "**detail**" are not needed in the most common use case of just updating the internal state of the plugin.*

*This method is similar to **OnPropertyChanged(id, value)** and in most cases can be implemented in similar fashion.*

*In this function the plugin needs to update the corresponding internal state, namely using the new **GetTimelinePropertyValue(id)** method from **IObjectInstance** which gets the value of a property with any changes a timeline might be applying to it. After the corresponding internal state is updated, refresh the layout view to view the changes.*

2 implement OnExitTimelineEditMode (). This method is called when timeline edit mode is turned off. In this method the plugin's internal state should be updated again so any timeline changes from the preview are reset. Using **GetPropertyValue(id)** to get values without timeline changes, applying those to the relevant internal variables of the plugin and refreshing the layout view should be enough.

Behaviors

See plugin integration above, all steps apply.

Effects

Edit the effect's .json file and add the interpolatable property with a value of true to each paramenter definition which should be supported by timelines. No additional modifications needed.

***NOTE**: Remember that not all properties need to be supported, if it looks like it doesn't make sense for a property to receive dynamic updates, it is ok to not support it.*

SCRIPT MINIFICATION

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/script-minification

Projects exported with *Minify script* enabled will run all script through an advanced minifier. This includes "property mangling" (renaming object properties and methods) to achieve maximum compression, and increase the difficulty of reverse engineering.

JavaScript in addons needs to be specially written to take in to account the minifier if they use external APIs that are not processed by the same minifier. Once you are familiar with what the minifier does, this is a straightforward process. **Be sure to test your addon with minification** to ensure it won't be broken when users export with minification enabled.

This process also affects JavaScript code that users write in Construct the same way as it affects your addon. Therefore for details on how to handle script minification, refer to the guide on exporting with advanced minification from the scripting section of the Construct manual.

WRAPPER EXTENSIONS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/wrapper-extensions

An additional feature of Construct's plugin SDK is that it allows bundling a *wrapper extension* for deeper platform integration. This is currently supported with the following exporters:

- Windows WebView2
- Xbox UWP (WebView2)
- macOS WKWebView
- Linux (CEF)

The Windows WebView2 exporter uses a traditional desktop Windows application using low-level Win32 APIs that embeds Microsoft WebView2 to load web content. Similarly the Xbox UWP (WebView2) exporter embeds WebView2 to load web content, but in the context of a Universal Windows Platform (UWP) app. The macOS WKWebView uses the system web view (based on WebKit, the Safari browser engine) to load web content in a macOS app. The Linux CEF exporter uses the Chromium Embedded Framework (CEF) instead of a system webview, as Linux does not provide a consistent webview in the platform.

These applications can be thought of as a "wrapper" around the web content. Plugins can provide a dynamic link library that extends the wrapper with custom features using the full capabilities of the wrapper application - hence the name *wrapper extension*. The model is similar to Cordova on mobile, where a Cordova plugin can be used for platform-specific integration and called from JavaScript, performing a similar role to a wrapper extension.

Dynamic link libraries use the extension **.dll** on Windows, **.dylib** on macOS and **.so** on Linux. For brevity this guide will use the term DLL to refer to any of these.

The wrapper extension system uses a minimal message-passing system to send small amounts of JSON data between the Construct plugin and the wrapper extension. This allows them to communicate so the wrapper extension can perform tasks for the Construct plugin that are not normally achievable in JavaScript alone. It is specifically designed for integrating C/C++ SDKs such as Steamworks.

The Construct Addon SDK includes the wrapper extension SDK under the path *plugin-sdk/wrapperExtensionPlugin*. A wrapper extension works as follows:

- A Visual Studio 2022 (the Community edition is a free download) solution for Windows, an Xcode project for macOS, and a CMake project for Linux, in the *extension* subfolder that uses C++ code to build a DLL which integrates custom features, such as a C/C++ SDK like Steamworks.

- The DLL uses *.ext.dll* (Windows), *.ext.dylib* (macOS) or *.ext.so* (Linux) as the file extension. The wrapper application looks for files with this name in the same folder as the executable, and will automatically load them on startup.
- The Construct plugin bundles the DLL file by calling `AddFileDependency()` with the type `"wrapper-extension"`. This means when a project using the addon is exported, it will also export the necessary file (e.g. the *.ext.dll* file for Windows).
- The Construct plugin can then detect that the wrapper extension is available, and if it is, send messages instructing the wrapper extension to perform certain tasks.

The sample in the SDK implements a wrapper extension that demonstrates returning data from C++ back to JavaScript, and implements an action that shows a message box to the user. This uses the Windows MessageBox API, the macOS NSAlert API, and GTK on Linux.

Note that in a UWP app (for the Xbox UWP exporter), the DLL must be configured as a Universal Windows DLL. See the Xbox Live UWP plugin code on GitHub for an example of a wrapper extension configured this way.

Messaging

In order to exchange messages, both the wrapper extension and the Construct plugin must set the same *component ID*. This must uniquely identify your plugin/extension combination. If any other plugin/extension uses the same component ID, it will cause a conflict and one of the plugins will fail.

The wrapper extension should call `iApplication->RegisterComponentId()` in the `WrapperExtension` constructor to register its component ID. The JavaScript plugin should call `this.SetWrapperExtensionComponentId()` in its constructor to register the same component ID. Then the two can exchange messages.

The JavaScript plugin should call `this._isWrapperExtensionAvailable()` after setting the component ID to check that the wrapper extension is available. This is because it is unavailable in other exporters, and also because it's possible that loading the wrapper extension could fail for some reason. If the wrapper extension is unavailable, async messages will return a promise that never resolves, which could cause the project to hang. It's advisable to also provide an Is available condition so users can check the plugin features are available in their event sheets.

There are two kinds of ways messages can be sent from JavaScript to the wrapper extension: a one-off message, and an async message.

One-off messages

A one-off message is a "fire and forget" scheme: a message will be sent but no attempt is made to receive a result or identify if the operation completed.

A one-off message can be sent from JavaScript with a call `this._sendWrapperExtensionMessage("message-id", [params...])`. The message ID identifies the kind of message. The second parameter is an optional array of parameters to pass with the message. These must only be boolean, number or string type values. Messages sent from the wrapper extension can be received with `this._addWrapperExtensionMessageHandler("message-id", handlerFunc)`. The handler function is passed an object with a small amount of JSON data sent from the wrapper extension.

A one-off message can be sent from the wrapper extension with a call like:

```
// C++
SendWebMessage("message-id", {
        { "sampleString1",      "Hello world!" },
        { "sampleString2",      "Foo bar baz" },
});
```

In this case the second parameter is a small amount of JSON data that is passed to the JavaScript message handler. The keys must be strings, and the values may only be boolean, number (`double` type to match JavaScript's number type), or string. (Strings in the C++ SDK must be `std::string` or C-style `char*` in UTF-8 encoding.)

***Note:** when reading JavaScript object properties sent from C++, be sure to use the minify-proof string syntax (e.g. `result["sampleString1"]`), as these properties come from an external source and so should not be changed by the minifier. See Script minification for more details.*

Wrapper extensions receive all messages from JavaScript to the same `HandleWebMessage()` method. That method receives a string of the message ID, and it's up to the wrapper extension to examine that string and respond appropriately depending on the kind of message. The recommended architecture is to use that method solely to distinguish the kind of message, unpack parameters, and then call a dedicated handler method.

Async messages

JavaScript can also send an asynchronous message to the wrapper extension. (This is only supported for JavaScript - there is not currently any support for the wrapper extension to send an asynchronous message to JavaScript.) This is done by calling `this._sendWrapperExtensionMessageAsync()` which works similarly to `this._sendWrapperExtensionMessage()`, except it returns a promise that resolves when the wrapper extension responds to the message. It is a useful way to retrieve data from the wrapper extension, including whether a requested operation completed successfully. It can also be used on startup to perform initialization work.

The wrapper extension receives asynchronous messages the same way as one-off messages, except the `asyncId` parameter is set to a unique number for the message. In order to respond to the message, it must call `SendAsyncResponse()` passing the same `asyncId` the message was received with, e.g.:

```
SendAsyncResponse({
        { "sampleString1",                      "Hello world!" },
        { "sampleString2",                      "Foo bar baz" },
}, asyncId);
```

The provided JSON data works the same as with `SendWebMessage()`, and is used as the value that the JavaScript call to `_sendWrapperExtensionMessageAsync()` resolves with.

The wrapper extension must respond to asynchronous messages on all codepaths, including in the event of an error. If it does not, the promise returned by `_sendWrapperExtensionMessageAsync()` *will never resolve, which could result in the project hanging.*

Exporting properties to package.json

A Construct plugin that uses a wrapper extension can make use of the IPluginInfo method `SetWrapperExportProperties()` to export the values of some plugin properties to the exported package.json file. The wrapper extension can then parse the contents of package.json on startup and find the values of these properties before any web content has loaded at all. This can be useful for loading SDKs with plugin properties specifying initialization details (like an app ID or API key) before anything else loads, which some SDKs recommend.

Suggested architecture

It is recommended that as much of your plugin logic as possible is implemented in JavaScript. Only send messages to the wrapper extension to make specific API calls that aren't possible from JavaScript. This way it minimizes the amount of platform-specific C++ code necessary, and ensures as much logic as possible happens in the same place, rather than spread across different codebases. Also, JavaScript is an easier programming language to work with, as it has easier-to-use facilities for async code and avoids the need for manual memory management (while still providing excellent performance).

Strings on Windows

For historical reasons, Windows APIs called from C++ that use strings generally use "wide strings" with UTF-16 encoding. These are strings of "wide characters" which are 16-bit types on Windows. This uses the `wchar_t` type for a character, and `std::wstring` for the STL string equivalent (as well as types like `LPCWSTR` for the C-style equivalent in Windows header files).

On the other hand, most modern software and recent C++ codebases use UTF-8 encoding. This uses the standard 8-bit `char` type and `std::string` in the STL (as well as types like `char*` for the C-style equivalent).

> *Remember that in both UTF-8 and UTF-16, a single "character" is in fact a Unicode code unit and doesn't necessarily correspond to a single visible character.*

Consistent with the modern style, the wrapper extension SDK uses UTF-8 encoding when dealing with strings. However this means strings must be converted when calling Windows APIs that use wide strings. The SDK provides the utility methods `Utf8ToWide()` which converts a UTF-8 `std::string` to a UTF-16 `std::wstring` suitable for passing to Windows APIs. The `c_str()` method of STL strings also provides a C-style string that Windows usually expects. The `WideToUtf8()` method can then also convert a UTF-16 `std::wstring` back to a UTF-8 `std::string`, suitable for converting back wide strings returned by Windows APIs. The recommended approach is to use UTF-8 everywhere, and only convert to UTF-16 to call a Windows API that requires it; if the API call returns a UTF-16 string then it should immediately be converted back to UTF-8. This means as much code as possible only uses UTF-8 and UTF-16 is used minimally solely to interact with Windows APIs.

> *The Windows WebView2 wrapper application also configures the process code page to UTF-8. This makes it possible to directly call '-A' variant Windows APIs (e.g. `MessageBoxA()`) with UTF-8 strings. However it is only supported in Windows 10 version 1903 (May 2019 Update) and newer, and may not be supported with all available Windows APIs. For more information see the Microsoft documentation Use UTF-8 code pages in Windows apps. While this option may be useful, particularly in future, for the most straightforward and consistent approach we recommend continuing to call all Windows APIs with wide strings in UTF-16 format.*

macOS architecture

Apple platforms typically use Objective-C or Swift. It is easiest to interoperate with C++ code with Objective-C and this is the approach we recommend. In Xcode you can configure .cpp files to be the type *Objective-C++ code*, and this allows mixing both Objective-C and C++ code. This means a .cpp file can also use `#import` and Objective-C style calls like `[alert setMessageText:nsTitle]`. The sample wrapper extension code uses this approach.

Apple platforms typically use NSString for strings. Similar to our advice for Windows, we recommend using `std::string` with UTF-8 encoding as much as possible, and only use `NSString` when interacting with platform APIs, converting to `NSString` to make Objective-C calls and immediately converting any `NSString` results back to `std::string`. Some sample code for converting between `NSString` and `std::string` is shown below.

```
std::string message = "Hello world";
```

```
// Convert std::string to NSString
NSString* nsMessage = [NSString stringWithUTF8String:message.c_str()];

// Convert NSString back to std::string
message = std::string([nsMessage UTF8String]);
```

By default Xcode builds universal binaries which include code for both Intel (x64) and Apple Silicon (ARM64) architectures.

Platform-specific code

You can use the following preprocessor definitions to identify the platform being built and so incorporate platform-specific code. Note these definitions only identify the OS, not the architecture.

- `_WIN32` is defined by Visual Studio for Windows builds
- `__APPLE__` is defined by Xcode for macOS
- `__linux__` is defined by gcc for Linux builds

Additional examples

There are real-world examples of using the wrapper extension SDK to integrate SDKs on the Scirra GitHub account, including open-source plugins that integrate the Steamworks SDK and the Epic Games Online Services (EOS) SDK. These should help provide sample code that demonstrates how to build a useful wrapper extension.

PORTING CONSTRUCT 2 PLUGINS/BEHAVIORS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/porting-c2-addons

---

To port a Construct 2 plugin or behavior to Construct 3, use the following checklist.

1. Start by copying the template SDK to a new folder.
2. Update the addon metadata in addon.json.
3. Update the icon. An SVG icon is preferable. The .ico files Construct 2 uses are not supported in Construct 3, but you can use a PNG icon. Just delete icon.svg, add icon.png, and call `this._info.SetIcon("icon.png", "image/png");` in the plugin/behavior constructor.
4. Update the plugin/behavior constants and identifiers in plugin.js/behavior.js, type.js and instance.js, as described in configuring plugins/configuring behaviors.
5. Match your Construct 2 addon's configuration by making calls to IPluginInfo/IBehaviorInfo in the addon constructor. For example if your Construct 2 plugin was a single-global plugin, the Construct 3 plugin should call `this._info.SetIsSingleGlobal(true);` in the plugin constructor.
6. Add equivalent properties as the Construct 2 addon has. See *Specifying plugin properties* in configuring plugins. (The process is identical for behaviors.)
7. Create corresponding action, condition and expression definitions. See defining actions, conditions and expressions. The key point to ensure Construct 2 projects using your addon can be imported to Construct 3 is:
    - 8 Give every action, condition and expression a new `id` based on a string
    - 9 Also set the `c2id` property to the corresponding numeric ID that the Construct 2 addon used
10. Update the language file to contain the UI strings for the addon, properties, and ACEs.
11. You'll then need to port the runtime script to the C3 runtime, since Construct 3 introduced an entirely rewritten engine.

Once complete, zip all the addon files and rename the .zip to .c3addon. You should now have a addon you can install via the Addon Manager in Construct 3.

PORTING TO ADDON SDK V2

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/porting-addon-sdk-v2

Construct formerly supported a legacy SDK, referred to as the Addon SDK v1. As of Construct 3 r450+, released in 2025, it now only supports a modern replacement with industry-standard encapsulation, referred to as the Addon SDK v2. This guide covers what you need to do to upgrade an existing addon using the Addon SDK v1 to v2.

Only plugins and behaviors need updating to the Addon SDK v2. This is because the main change is to how runtime scripts are written. Effects and themes do not need porting, as they do not use runtime scripts. Further, plugins and behaviors should not need any significant modifications to any of their editor code, DOM-side runtime code, or wrapper extensions; usually only the runtime scripts in the *c3runtime* folder need significant changes.

Porting guide

Follow these steps to update a plugin or behavior from Addon SDK v1 to v2.

Step 1: update SDK version

In addon.json, update the `"sdk-version"` field to 2. If the field is missing, add it with the value 2. This indicates to Construct that the addon is using the Addon SDK v2.

You'll probably want to also increment the `"version"` field as the update to Addon SDK v2 warrants an updated addon version too.

Step 2: update base classes

The base classes used in the Addon SDK v2 are different to v1. In other words, the class after the `extends` keyword needs to be updated.

The following table lists the changes to base classes for plugins.

SDK v1 base class	SDK v2 base class
C3.SDKPluginBase	globalThis.ISDKPluginBase
C3.SDKTypeBase	globalThis.ISDKObjectTypeBase
C3.SDKInstanceBase	globalThis.ISDKInstanceBase
C3.SDKWorldInstanceBase	globalThis.ISDKWorldInstanceBase
C3.SDKDOMPluginBase	globalThis.ISDKDOMPluginBase
C3.SDKDOMInstanceBase	globalThis.ISDKDOMInstanceBase

The following table lists the changes to base classes for behaviors.

SDK v1 base class	SDK v2 base class
C3.SDKBehaviorBase	globalThis.ISDKBehaviorBase
C3.SDKBehaviorTypeBase	globalThis.ISDKBehaviorTypeBase
C3.SDKBehaviorInstanceBase	globalThis.ISDKBehaviorInstanceBase

Step 3: update class constructors

In the Addon SDK v2, the constructors of the above base classes do not use any parameters. For example with the Addon SDK v1 class `C3.SDKInstanceBase`, the constructor used two parameters. With the Addon SDK v2 class `globalThis.ISDKInstanceBase`, the constructor uses no parameters.

Further, for instance classes specifically, instead of the `properties` parameter being passed in the constructor, instead call `this._getInitProperties()` to access the same information. An example of the difference is shown below.

```
// Instance constructor in Addon SDK v1
constructor(inst, properties)
{
	super(inst);

	if (properties)
	{
		// ... read properties ...
	}

	// ... rest of constructor ...
}

// Instance constructor in Addon SDK v2
constructor()
{
	super();

	const properties = this._getInitProperties();
	if (properties)
	{
		// ... read properties ...
	}

	// ... rest of constructor ...
}
```

Further, for DOM or wrapper extension plugins, note that the DOM component ID or wrapper extension ID are now passed as part of an options object in the `super()` call, e.g.:

```
// In ISDKDOMPluginBase or ISDKDOMInstanceBase constructor:
super({ domComponentId: DOM_COMPONENT_ID });

// In ISDKInstanceBase constructor for a wrapper extension:
```

```
super({ wrapperComponentId: "my-extension" });
// (instead of calling SetWrapperExtensionComponentId())
```

Step 4: remove separate script interface

In the Addon SDK v2, runtime script classes *are* the script interface classes: all public properties and methods are also accessible from Construct's scripting feature. Therefore there is no longer any need to define a separate script interface class.

The method `GetScriptInterfaceClass()` can be deleted and the entire script interface class deleted if one was specified.

To ensure backwards compatibility, if you did have a script interface class, make sure the main class now implements all the same properties and methods as the script interface class used to.

Step 5: update property and method names

While all previously documented properties and methods in SDK v1 are still supported in SDK v2, they have been renamed to follow the naming conventions of the rest of the scripting APIs. In some cases some details are different to simplify the SDK or to adapt to the particular requirements of the new architecture.

For example the SDK v1 instance method `GetDebuggerProperties()` is now named `_getDebuggerProperties()`, but otherwise works identically; `Trigger()` is now named `_trigger()` but otherwise works identically; and so on. Refer to the class links in the above table to review the full reference. Note also some features may now be in other base classes; for example the SDK v1 class `SDKInstanceBase` had a property `this._runtime`; the SDK v2 class `ISDKInstanceBase` does not define a property for the runtime, but it inherits from IInstance, which defines the property `this.runtime`.

For consistency, you will likely want to rename any other class methods to follow the new camelCase naming convention. As per the Addon SDK coding conventions, you may wish to use an underscore prefix to indicate methods which should not be called from the scripting feature, but cannot be made private.

Step 6: update remaining code

In some addons, such as if the main purpose is to integrate a third-party service, the addon should not require any further major changes. However if your addon has extensive logic using the runtime APIs from the Addon SDK v1, these will need to be rewritten in terms of the new runtime APIs based on Construct's scripting feature. A full reference of the available APIs can be found in the scripting reference section of Construct's manual.

Recommended architecture

Our recommended architecture is to implement core logic as methods and properties (or setters/getters) on your main instance class. This makes the features accessible to the scripting feature. Then actions, conditions and expressions just call the script APIs, so they work consistently with the scripting feature, and don't have to have any significant logic in ACE methods. Then the debugger methods can use the same methods so the debugger views and changes things consistently with the way actions/expressions or script APIs would. Overall this means event sheets, scripting, and the debugger all share the same implementation, which also makes maintenance easier, as there is only one place code needs to be updated.

Recommended additional changes

The following changes are not required, but are recommended.

Delete PLUGIN_VERSION

In the editor plugin script, delete `PLUGIN_VERSION` and delete the line `this._info.SetVersion(PLUGIN_VERSION);` . This is because with the Addon SDK v2, the addon version is taken from the version specified in addon.json only; the version in the editor plugin script is now ignored. A deprecation warning will appear in the console until these changes are made.

Use globalThis

In the past the global object may have been referred to using `window` or `self` . The modern standardized way to do this is with `globalThis` , so it is recommended to use that instead of any other alternatives.

Configure to use modules

The Addon SDK v2 now supports using JavaScript Modules (i.e. `import` and `export` statements) in your runtime scripts. By default Construct creates a new main script module for your addon so you don't have to set anything else up, but it's good practice to set it up anyway as it's how all modern addons are written, and it's necessary if you want to use `import` or `export` in your runtime scripts. For more details see the section *Configuring use of modules* in Runtime scripts.

Sample diffs

On the Construct Addon SDK GitHub repository, you can find commits that update the SDK samples from the Addon SDK v1 to the Addon SDK v2. The differences, or "diffs", in these commits can serve as a reference of what needs to be changed to update addons. With the sample addons the changes take in to above everything described above, including recommended changes. The following links display the diffs for the updates for each SDK sample.

- Update customImporterPlugin to addon SDK v2
- Update domElementPlugin to addon SDK v2

- Update domMessagingPlugin to addon SDK v2
- Update drawingPlugin to addon SDK v2
- Update editorTextPlugin to addon SDK v2
- Update singleGlobalPlugin to addon SDK v2
- Update wrapperExtensionPlugin to addon SDK v2
- Update sample behavior to addon SDK v2

Publishing

Addon developers should now be publishing updates to their addons using SDK v2. Support for the legacy SDK v1 will only continue until the middle of 2025, although beyond that support for SDK v1 addons will be extended until the end of 2026 with an LTS release.

In the Addons section of the website, there is a checkbox labelled **Uses Addon SDK v2** when editing an addon. When uploading an SDK v2 addon it should automatically detect that your addon uses SDK v2 and check this box for you. It's worth double-checking the checkbox for existing addons or after publishing an update using SDK v2, as this then ensures it is listed when using the *SDKv2 addons only* filtering option, which will eventually be enabled by default and therefore hide all old addons using SDK v1 by default unless this option has been checked.

THEME ADDONS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/themes

Construct 3 allows theme addons, which simply add some custom stylesheets to the document. This allows a great deal of flexibility in customising the appearance of the Construct 3 editor. Any features of CSS can be used to alter the UI appearance. You can use browser developer tools to identify the classes and DOM structure used in the editor, and override the styles Construct 3 applies by default in your own stylesheet.

Themes are based on the same .c3addon file that plugins and behaviors use, although with fewer necessary files. As you can see in the theme SDK download, all you need are **lang/en-US.json**, **addon.json**, an icon, and a stylesheet. Note your addon metadata must also contain a list of `stylesheets` — this is just a list of the CSS files to add to the document when your theme is applied.

As with other kinds of addon, you can test themes as developer addons for quicker development.

Tips for developing themes

- Construct 3 uses a range of CSS variables (aka custom properties) to more easily customise certain parts of the UI. These also allow customisation of colors not in the DOM, such as the Layout View (which is rendered with WebGL). The available CSS variables are listed in comments in the theme SDK. Note colors in CSS variables must always be written in hex format (#000000), except for layout view colors which can use rgba() syntax. (Other CSS properties can use any syntax; only CSS variables are restricted, since they are sometimes read from JavaScript.)
- Avoid making significant alterations to layout. In many cases Construct 3's code assumes certain layout of elements. Additionally it is time consuming to test layout changes work across every part of the UI. For example Construct 3 has over 50 dialogs, and testing all the dialogs still appear correctly after a change is a lot of work. On the other hand, cosmetic changes like colors and borders are usually safe.
- Be sure to also test your theme on mobile. Construct 3 uses different paradigms and layouts in a number of places when adapting to smaller mobile displays, and you should check your theme still appears correctly in that mode.
- Be wary of styles that could have a performance impact. For example heavy use of shadows, effects (such as blurs or other filters) or animations, could be taxing on the CPU or GPU. Fewer people will use your theme if it slows down their device, so try to make sure your styles are used efficiently.

Using themes

Once you have installed a theme addon, you can start using it by selecting it from the Settings dialog. Note this involves restarting Construct 3 twice: once when the addon is installed, after which it appears in the Settings dialog, and then again after selecting it in the Settings dialog.

ENABLING DEVELOPER MODE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/enabling-developer-mode

Construct 3's Developer Mode enables extra testing features for developers, such as a special addon testing mode.

To enable Developer Mode, open Construct 3's settings dialog and click or tap the dialog caption 10 times. A prompt will appear asking if you want to show developer mode settings. Click OK.

Now in the settings dialog there should be a checkbox named *Enable developer mode*. Tick the checkbox, close the settings dialog and restart Construct 3. You are now using Developer Mode.

The developer mode menu

When Developer Mode is active, a new menu named *Developer mode* appears in the main menu. This provides some shortcuts and tools useful in developer mode, such as a shortcut to install a developer mode addon, or set up TypeScript for an addon.

TESTING ADDONS IN DEVELOPER MODE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/using-developer-mode

While developing an addon, it is inconvenient to have to package and install a new .c3addon file every time you make a change. To make testing easier, you can load an addon from a local web server. This means every time you reload Construct 3, it re-loads the latest version of your addon from your local web server, saving you from having to make any .c3addon packages. Developer mode also means Construct reloads the addon files on every preview, making it much easier to check changes to runtime code, but note that most of the time changing code that affects the editor will still require a reload of the editor.

Step 1: enable Developer Mode

For these steps to work, you must first enable Developer Mode.

Step 2: start a local web server

There are many ways to run a local web server, e.g. with a Chrome extension, or a standalone server like nginx. Refer to your chosen server's documentation for installation and configuration.

Your local web server must host on *localhost*. Construct 3 will refuse to load addons from any other origin. You can host on any port, but it is recommended to use a port in the ephemeral port range 49152-65535.

The local web server must serve all the addon files with CORS (Cross-Origin Resource Sharing) enabled for Construct 3 to be able to load them, since it will be making a cross-domain request. In practice this means adding this HTTP header to the server response:

`Access-Control-Allow-Origin: *`

To do this you need to enable CORS if your web server provides a setting for that, or manually specify the header. For example in nginx, add the following directive in your server location section:

`add_header Access-Control-Allow-Origin *;`

You may also wish to review the caching headers to ensure your local server does not return old cached files. Disabling caching entirely will ensure Construct 3 always receives the latest files.

Once the server is fully configured, simply host the contents of the c3addon file in a folder on your local web server. For example the URL *http://localhost:65432/myaddon/addon.json* should serve the addon.json file for your addon.

In modern browsers, `http://localhost` counts as a secure context even though it does not use HTTPS. Therefore you should not need to set up SSL/TLS for the local server.

Step 3: update addon.json to include file list

Normally when files are extracted from the zip-format .c3addon file, Construct 3 can obtain a file list from the zip. However when loading from a local web server, Construct 3 needs another way to identify all available files. So you must update addon.json to include a complete file list. For more information see the section *Developer mode addons* in Addon metadata.

The file list can be left in when distributing your .c3addon file — there's no need to later remove it.

Step 4: install the developer mode addon

In Construct 3, open the Addon Manager. After enabling Developer Mode, there should be a new button at the bottom labelled *Add dev addon...*. Click this button. A dialog will appear asking for the URL to the addon's addon.json file on your local web server. Enter its path, e.g. *http://localhost:65432/myaddon/addon.json*, and press OK. If the addon.json file is reachable and parsed successfully, you'll see a message indicating to restart Construct 3 to load the addon. If an error occurs, check the browser console for more details.

Step 5: develop the addon

Now every time you reload Construct 3, or preview a project using the addon, the latest version of your addon is loaded from the local web server. This makes it much quicker to test changes to your addon. For example if you make a change that crashes Construct 3 on startup, there is no need to clear the browser cache to remove the addon; you can simply fix the problem and reload C3. Similarly you can adjust some runtime code and press preview, and your code changes will be immediately reflected.

Your addon will appear in the Addon Manager as having a "Developer" source. Note "Developer" addons cannot be bundled with projects when using the *Bundle addons* feature.

You can also uninstall your developer mode addon like any other addon, by right-clicking it in the Addon Manager and selecting *Uninstall*.

SAFE MODE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/safe-mode

For normal addon development, you should use addon testing mode. This allows quick iteration and you can easily fix addons even if they crash Construct 3 on startup. However in some cases you may build a .c3addon package which accidentally still crashes the editor on startup. To remove this addon, you can clear your browser storage. Alternatively you can use Safe Mode in Construct 3, which does not load third-party addons but still lists them in the Addon Manager so they can be uninstalled.

To use Safe Mode, add **?safe-mode** to the URL, e.g. editor.construct.net/?safe-mode. Note all third-party addons are disabled in this mode. You should immediately open the Addon Manager, uninstall the problematic addon, and then restart Construct 3. Be sure to remove the ?safe-mode part of the URL to re-enable loading third-party addons.

It is not recommended to open any projects or try to do any actual work in safe mode: it exists only so you can reach the Addon Manager dialog to uninstall the addon.

IBEHAVIORINSTANCEBASE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/base-classes/ibehaviorinstancebase

The `IBehaviorInstanceBase` interface is used as the base class for behavior instances in the SDK.

`IBehaviorInstanceBase` cannot be directly constructed; it should only be used as a base class.

Properties

this._sdkBehaviorType

Reference to the associated SDK type class.

this._behaviorInstance

Reference to the IBehaviorInstance interface representing this instance in the editor. This allows access to Construct's built-in features for behavior instances.

Methods

OnPropertyChanged(id, value)

Optional override for when a property with the given ID is changed. The value the property was changed to is also passed.

GetBehaviorInstance()

Return the IBehaviorInstance interface representing this instance in the editor.

GetSdkBehaviorType()

Return the associated SDK type class.

OnAddedInEditor()

Optional override for when the behavior instance has been created due to the user adding a new behavior in the editor.

IINSTANCEBASE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/base-classes/iinstancebase

The `IInstanceBase` interface is used as the base class for instances in the SDK. For "world" type plugins, instances instead derive from IWorldInstanceBase, which itself derives from `IInstanceBase`.

`IInstanceBase` cannot be directly constructed; it should only be used as a base class.

Properties

this._sdkType

Reference to the associated SDK type class.

this._inst

Reference to the IObjectInstance interface, or IWorldInstance interface for "world" type plugins, representing this instance in the editor. This allows access to Construct's built-in features for instances.

Methods

Release()

Optional override for when an instance is released.

OnCreate()

Optional override for when an instance is created in the editor.

OnPropertyChanged(id, value)

Optional override for when a property with the given ID is changed. The value the property was changed to is also passed.

LoadC2Property(name, valueString)

Optional override to use custom logic for importing properties from a Construct 2 project referencing a Construct 2 version of this plugin.

GetProject()

Return the IProject representing the instance's associated project.

GetObjectType()

Convenience method to return the IObjectType interface representing Construct's object type class.

GetInstance()

Return the IObjectInstance corresponding to this instance.

IWORLDINSTANCEBASE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/base-classes/iworldinstancebase

The `IWorldInstanceBase` interface is used as the base class for instances in the SDK for "world" type plugins. It derives from IInstanceBase.

`IWorldInstanceBase` cannot be directly constructed; it should only be used as a base class.

Methods

OnPlacedInLayout()

Optional override called when an instance is explicitly placed in the layout by the user. This is the right time to set any additional defaults such as the initial size or origin.

Draw(iRenderer, iDrawParams)

Called when Construct wants the instance to draw itself in the Layout View. `iRenderer` is an IWebGLRenderer interface, used for issuing draw commands. `iDrawParams` is an IDrawParams interface, used for providing additional information to the draw call.

GetTexture(animationFrame)

Load a texture from an IAnimationFrame. Texture loading is asynchronous and is started in the first call. The method will return `null` while the texture is loading. Construct will automatically refresh the Layout View when the texture finishes loading, at which point the method will return an IWebGLTexture interface that can be used for rendering. Plugins typically render a placeholder of a semitransparent solid color while the texture is loading.

GetTexRect()

When a texture has successfully loaded, returns an SDK.Rect indicating the dimensions of the image to render in texture co-ordinates. Note that due to Construct's in-editor spritesheeting engine, this is usually a subset of a texture.

HadTextureError()

Returns true to indicate texture loading failed. Plugins typically switch the placeholder to a red color in this circumstance.

IsOriginalSizeKnown()

GetOriginalWidth()

GetOriginalHeight()

Optional overrides to specify the "original size" of the instance. Typically if a plugin supports this, it is the size of the image. This enables percentage size options in the Properties Bar. The default implementation returns `false` from `IsOriginalSizeKnown()`, disabling the feature. To enable it, return `true` from `IsOriginalSizeKnown()`, and return the original size in the `GetOriginalWidth()` and `GetOriginalHeight()` methods.

HasDoubleTapHandler()

OnDoubleTap()

Optional override which is called when the user double-clicks or double-taps an instance in the Layout View. This also enables an *Edit* option in the context menu, which also calls the double-tap handler. Typically plugins with an image use this handler to edit the image. The default implementation returns `false` from `HasDoubleTapHandler()`, disabling the feature. To enable it, return `true` from `HasDoubleTapHandler()` and then override `OnDoubleTap()` to perform a task.

COLOR INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/geometry-interfaces/color

The `Color` interface represents a floating-point RGBA color in the SDK. It can also be constructed independently as a general-purpose color class. Each color component is normalized to the range [0, 1].

In the WebGL renderer, colors are normally required to have premultiplied alpha. Some APIs already return premultiplied colors, but others may not; check the documentation for any API methods returning colors to find out which are used. Wherever possible avoid using the `unpremultiply()` method, since it is lossy.

Constructor

```
new SDK.Color();
new SDK.Color(r, g, b, a);
```

A `Color` can be constructed with no parameters, which defaults all components to zero, or with given RGBA components.

Methods

setRgb(r, g, b)

Set the RGB components only, without affecting the alpha component, in a single call.

setRgba(r, g, b, a)

Set the RGBA components of the color in a single call.

copy(color)

Set the components of the color by copying another `SDK.Color`.

copyRgb(color)

Set the RGB components only, without affecting the alpha component, by copying another `SDK.Color`.

clone()

Return a new instance of an `SDK.Color` with an identical color to this one.

setR(r)

setG(g)

setB(b)

setA(a)

Set each component of the color individually. Note color components are floats in the range [0, 1].

getR()

getG()

getB()

getA()

Get each component of the color individually.

equals(color)

Return a boolean indicating if this color exactly matches another `SDK.Color`.

equalsIgnoringAlpha(color)

Return a boolean indicating if this color exactly matches the RGB components of another `SDK.Color`. The alpha component is ignored.

equalsRgb(r, g, b)

Return a boolean indicating if this color exactly matches the given RGB components.

equalsRgba(r, g, b, a)

Return a boolean indicating if this color exactly matches the given RGBA components.

premultiply()

Multiply the RGB components by the A component. This is usually required for rendering.

unpremultiply()

Divide the RGB components by the A component.

> *Avoid this method whenever possible, because it is lossy. (Unpremultiplying a premultiplied color will lose some precision in the RGB components and may not exactly match the original color.)*

QUAD INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/geometry-interfaces/quad

The `Quad` interface represents a shape made from four points in the SDK. Typically it is used to represent rotated rectangles. It is also the main primitive used for rendering, by passing object's bounding quads to the WebGL renderer.

Constructor

```
new SDK.Quad();
new SDK.Quad(tlx, tly, trx, try_, brx, bry, blx, bly);
```

A `Quad` can be constructed with no parameters, which defaults all co-ordinates to zero, or with given positions for each point. The naming convention is **tl** for the "top-left" point, **tr** for the "top-right" point, **br** for the "bottom-right" point, and **bl** for the "bottom-left" point, followed by "x" or "y" for each component of the point. Note that the points can appear at any orientation for rotated quads; the names only correspond to their actual position when the quad is set to an unrotated rectangle.

The name `try` is a keyword in JavaScript, so the "top-right Y" component is named with an underscore as `try_` to avoid colliding with the keyword.

Methods

set(tlx, tly, trx, try_, brx, bry, blx, bly)

Set all four points of the quad in a single call.

setRect(left, top, right, bottom)

Set the quad's points to represent an axis-aligned rectangle using the given positions. Note that this is only useful if you subsequently make further modifications to the quad, else you may as well use the Rect interface.

copy(quad)

Set all points of the quad by copying another `SDK.Quad`.

setTlx(n)

setTly(n)

setTrx(n)

setTry(n)

setBrx(n)

setBry(n)

setBlx(n)

setBly(n)

Set each point of the quad individually.

---

getTlx()

getTly()

getTrx()

getTry()

getBrx()

getBry()

getBlx()

getBly()

Get each point of the quad individually.

---

setFromRect(rect)

Set the points of the quad to an axis-aligned rectangle given by an SDK.Rect. Note that this is only useful if you subsequently make further modifications to the quad, else you may as well use the Rect interface directly.

---

setFromRotatedRect(rect, angle)

Set the points of the quad to a rotated rectangle given by an SDK.Rect, rotated about the origin by `angle` in radians.

---

getBoundingBox(rect)

Calculate the bounding box of the quad, and store the result by writing to a given SDK.Rect.

---

midX()

midY()

Return the average of the four points in the quad on each axis.

---

intersectsSegment(x1, y1, x2, y2)

Test if a segment, given as the line between points (x1, y1) and (x2, y2), intersects this quad, returning a boolean.

intersectsQuad(quad)

Test if another SDK.Quad intersects this quad, returning a boolean.

containsPoint(x, y)

Test if the given point is inside the bounds of this quad, returning a boolean.

RECT INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/geometry-interfaces/rect

The `Rect` interface represents an axis-aligned rectangle in the SDK. It can also be constructed independently as a general-purpose geometry class.

Constructor

```
new SDK.Rect();
new SDK.Rect(left, top, right, bottom);
```

A `Rect` can be constructed with no parameters, which defaults all co-ordinates to zero, or with given positions for the left, top, right and bottom positions.

Methods

set(left, top, right, bottom)

Set all sides of the rectangle in one call.

copy(rect)

Set all sides of the rectangle by copying another `SDK.Rect`.

clone()

Return a new instance of an `SDK.Rect` with identical values to this one.

setLeft(left)
setTop(top)
setRight(right)
setBottom(bottom)

Set the position of each side of the rectangle individually.

getLeft()
getTop()
getRight()
getBottom()

Get the positition of each side of the rectangle individually.

width()

height()

Get the width or height of the rectangle. Note if the right edge is to the left of the left edge, or the bottom edge above the top edge, this will return a negative size.

midX()

midY()

Return the average of the left and right, or top and bottom, positions.

offset(x, y)

Add `x` to the left and right positions, and `y` to the top and bottom positions, offsetting the entire rectangle.

inflate(x, y)

deflate(x, y)

Expand or shrink the rectangle using the given offsets. Inflating subtracts from the left and top edges and adds to the right and bottom edges, and deflating does the opposite.

multiply(x, y)

divide(x, y)

Multiply or divide each position by a given factor on each axis.

clamp(left, top, right, bottom)

Clamp each position in the rectangle to a given value, ensuring the rectangle does not extend beyond the bounds of the passed rectangle.

normalize()

Normalize the rectangle positions, swapping the left-right positions if the right position is on the left, and swapping the top-bottom positions if the bottom position is on the top. This ensures the width and height are positive.

intersectsRect(rect)

Test for an intersection with another `SDK.Rect`, returning a boolean indicating if it intersects.

containsPoint(x, y)

Test if the given point is inside the bounds of this rectangle, returning a boolean.

IDRAWPARAMS INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/graphics-interfaces/idrawparams

The `IDrawParams` interface provides additional parameters to a `Draw()` call in the SDK.

This interface cannot be directly constructed. It is only available in the `Draw()` call.

Methods

GetDt()

Return delta-time, the time since the last frame, in seconds. This is typically approximately 1/60th of a second (0.01666...). This value is only valid when the Layout View is continually scrolling, such as when dragging an instance to the edge of the Layout View window. Any other time it will be set to a dummy non-zero value, since there wasn't a frame immediately preceding the current one.

GetLayoutView()

Return an ILayoutView interface representing the current Layout View being drawn. This allows access to features of the Layout View in drawing code.

IWEBGLRENDERER INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/graphics-interfaces/iwebglrenderer

The `IWebGLRenderer` interface provides methods for rendering to the Layout View, which is rendered using a canvas. The interface's methods provide high-level drawing commands implemented by Construct, so you don't need to handle low-level concerns like vertex buffers.

Despite the name, the renderer may in fact be backed by a WebGPU context. However this does not affect how the interface works.

This interface cannot be directly constructed. It is only available as a parameter to a `Draw()` call.

Renderer state

Since IWebGLRenderer is based on WebGL, it also uses a persistent rendering state. Therefore to correctly render something, all the intended state must be specified, otherwise it will use an undefined previous state. IWebGLRenderer simplifies the renderer state to:

1. A blend mode. Typically a normal alpha blend mode is used.
2. A fill mode (internally, the current fragment shader). The fill modes can be *color fill* (draw a solid color), *texture fill* (draw a texture), and *smooth line fill* (for drawing smooth lines).
3. A color set by `SetColor()` or `SetColorRgba()`. The alpha component of the color is used as the opacity in texture fill mode.
4. A texture set by `SetTexture()`. This is only used in texture fill mode.

Therefore a `Draw()` method should begin by specifying the blend mode, the fill mode, the color, and the texture (if texture fill mode is used), before continuing to draw. The renderer efficiently discards redundant calls, so if the state does not actually change then these calls have minimal performance overhead.

Once all state is set up, quads can be issued using one of the `Rect()` or `Quad()` method overloads. These methods draw using the currently set state.

Methods

SetAlphaBlendMode()

Set the blend mode to a premultiplied alpha blending mode.

SetBlendMode(blendMode)

Set the blend mode by a string which must be one of `"normal"`, `"additive"`, `"copy"`, `"destination-over"`, `"source-in"`, `"destination-in"`, `"source-out"`, `"destination-out"`, `"source-atop"`, `"destination-atop"`, `"lighten"`, `"darken"`, `"multiply"`, `"screen"`. Passing `"normal"` is equivalent to calling `SetAlphaBlendMode()`.

SetColorFillMode()

Set the fill mode to draw a solid color, specified by the current color.

SetTextureFillMode()

Set the fill mode to draw a texture, specified by the current texture, and using the alpha component of the current color as the opacity.

SetSmoothLineFillMode()

Set the fill mode to draw smooth lines using the current color.

SetColor(color)

Set the current color with an SDK.Color.

SetColorRgba(r, g, b, a)

Set the current color by directly passing the RGBA components.

SetOpacity(o)

Set only the alpha component of the current color. Note this does not affect the RGB components.

SetCurrentZ(z)

GetCurrentZ()

Set and get the current Z component used for all 2D drawing commands that don't specify Z components, such as the `Rect2()` and `Quad3()`.

ResetColor()

Set the current color to (1, 1, 1, 1).

Rect(rect)

Draw a rectangle given by an SDK.Rect.

Rect2(left, top, right, bottom)

Draw a rectangle by directly passing the left, top, right and bottom positions.

Quad(quad)

Draw a quad given by an SDK.Quad.

Quad2(tlx, tly, trx, try_, brx, bry, blx, bly)

Draw a quad by directly passing the positions of each of the four points in the quad.

Quad3(quad, rect)

Draw a quad given by an SDK.Quad, using an SDK.Rect for the source texture co-ordinates to draw from.

Quad4(quad, texQuad)

Draw a quad given by an SDK.Quad, using another `SDK.Quad` for the source texture co-ordinates to draw from.

Quad3D(tlx, tly, tlz, trx, try_, trz, brx, bry, brz, blx, bly, blz, rect)

Quad3D2(tlx, tly, tlz, trx, try_, trz, brx, bry, brz, blx, bly, blz, texQuad)

Draw a 3D quad, specifying all four points of the quad with X, Y and Z co-ordinates. The first overload accepts texture co-ordinates via an SDK.Rect *rect*, and the second accepts texture co-ordinates via an SDK.Quad *texQuad*.

DrawMesh(posArr, uvArr, indexArr, colorArr)

Draw an array of textured triangles based on the given position, texture co-ordinate and index arrays, and an optional per-vertex color array. For more details refer to the documentation for the IRenderer `drawMesh()` method, which works the same (albeit with different casing on the method name).

ConvexPoly(pointsArray)

Draw a convex polygon using the given array of points, in alternating X, Y order. Therefore the size of the array must be even, and must contain at least six elements (to define three points).

Line(x1, y1, x2, y2)

Draws a quad from the point (x1, y1) to (x2, y2) with the current line width.

TexturedLine(x1, y1, x2, y2, u, v)

Draws a quad from the point (x1, y1) to (x2, y2) with the current line width, and using (u, 0) as the texture co-ordinates at the start, and (v, 0) as the texture co-ordinates at the end.

LineRect(left, top, right, bottom)

Draws four lines along the edges of a given rectangle.

LineRect2(rect)

Draws four lines along the edges of a given `SDK.Rect`.

LineQuad(quad)

Draws four lines along the edges of a given `SDK.Quad`.

PushLineWidth(w)

PopLineWidth()

Set the current line width for line-drawing calls. This must be followed by a `PopLineWidth()` call when finished to restore the previous line width.

PushLineCap(lineCap)

PopLineCap()

Set the current line cap for line-drawing calls. This must be followed by a `PopLineCap()` call when finished to restore the previous line cap. The available line caps are `"butt"` and `"square"`.

SetTexture(texture)

Set the current texture to a given IWebGLTexture.

CreateWebGLText()

Return a new IWebGLText interface. This manages text wrapping, drawing text, and uploading the results to a WebGL texture.

CreateDynamicTexture(width, height, opts)

Create a new empty IWebGLTexture for dynamic use, i.e. expecting the texture content to be replaced using `UpdateTexture()`. The size of the texture is given by `width` and `height` which must be positive integers. `opts` specifies options for the texture which is an object that can include the following properties:

- `wrapX` : the texture horizontal wrap mode: one of `"clamp-to-edge"` , `"repeat"` , `"mirror-repeat"`
- `wrapY` : as with `wrapX` but for the vertical wrap mode
- `sampling` : the texture sampling mode, one of `"nearest"` , `"bilinear"` or `"trilinear"` (default)
- `pixelFormat` : the texture pixel format, one of `"rgba8"` (default), `"rgb8"` , `"rgba4"` , `"rgb5_a1"` or `"rgb565"`
- `mipMap` : boolean indicating if mipmaps should be used for this texture, default true
- `mipMapQuality` : if `mipMap` is true, one of `"default"` (default), `"low"` or `"high"`

UpdateTexture(data, texture, opts)

Upload *data* as the new texture contents for the IWebGLTexture *texture*. This can only be used for textures created with `CreateDynamicTexture()` and managed by your addon.
data can be one of the following types: `HTMLImageElement` , `HTMLVideoElement` , `HTMLCanvasElement` , `ImageBitmap` , `OffscreenCanvas` or `ImageData` . Note in worker mode the DOM types cannot be used (`HTMLImageElement` , `HTMLVideoElement` , `HTMLCanvasElement`); in this case use `ImageBitmap` or `OffscreenCanvas` instead. This method cannot resize an existing texture, so the data must match the size the texture was created with; if the size needs to change, destroy and re-create the texture.
opts specifies options for the texture upload which is an object that can include the following properties:

- `premultiplyAlpha` : a boolean indicating whether to premultiply alpha of the image content specified by *data* (default true). Construct always renders using premultiplied alpha so this is normally necessary; however if the data is known to already be premultiplied, set this to false.

DeleteTexture(texture)

Delete a IWebGLTexture, releasing its resources. This can only be used for textures created with `CreateDynamicTexture()` and managed by your addon. Do not attempt to delete textures managed by the Construct engine.

IWEBGLTEXT INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/graphics-interfaces/iwebgltext

The `IWebGLText` interface manages text wrapping, drawing text to a canvas, and then uploading the result to a WebGL texture. This makes it easy to display text in a WebGL renderer. It is created via the IWebGLRenderer `CreateWebGLText()` method.

Methods

Release()

Destroy the object and its resources. `IWebGLText` must be released when it is no longer needed; do not simply drop references, otherwise not all of its resources will be collected. If your plugin creates an IWebGLText, it should release any it still uses in its own `Release()` method.

SetFontName(name)

Set the name of the font face used for drawing text.

SetFontSize(ptSize)

Set the size of the font, in points, used for drawing text.

SetLineHeight(px)

Set the extra line height spacing, in pixels, used for drawing text. Note 0 is the default, indicating no offset to the default line height.

SetBold(b)

Set the bold flag used for drawing text.

SetItalic(i)

Set the italic flag used for drawing text.

SetColor(color)

Set the color of the text using a SDK.Color or a string. If a string is passed, it is passed directly to a 2D canvas `fillStyle` property, so can be anything that property accepts, e.g. "red", "#00ffee", "rgb(0, 128, 192)" etc.

SetColorRgb(r, g, b)

Set the color of the text using separate RGB components.

SetHorizontalAlignment(h)

Set the horizontal alignment of the text within its bounding box. This can be one of `"left"`, `"center"` or `"right"`.

SetVerticalAlignment(v)

Set the vertical alignment of the text within its bounding box. This can be one of `"top"`, `"center"` or `"bottom"`.

SetWordWrapMode(m)

Set the word wrapping mode. This can be one of `"word"` (for space-delimited word wrapping) or `"character"` (for wrapping on any character).

SetText(text)

Set the text string to be drawn.

SetSize(width, height, zoomScale)

Set the size of the area that text can be drawn in. The size is specified in CSS pixels. The `zoomScale` can be increased to render the text at a higher resolution, which is useful when zooming in the Layout View.

GetTexture()

Get an IWebGLTexture interface representing the texture with the requested text rendered on to it. **Note:** the texture is generated asynchronously, so can return `null` when first requested. Use `SetTextureUpdateCallback()` to get a callback when the texture has updated, where the relevant Layout View can be redrawn to render with the updated texture.

GetTexRect()

Return a SDK.Rect representing the content area of the text on the WebGL texture. This is the subset of the texture that ought to be rendered. Note: this is only valid when `GetTexture()` returns a non-null result.

SetTextureUpdateCallback(callback)

Set a function to call when the texture containing the rendered text is updated. Since the texture is generated asynchronously, this is necessary to know when to redraw any views that may be displaying the text, so they can redraw with the updated texture.

ReleaseTexture()

Release the underlying WebGL texture. This can be used to save memory. However the texture will be re-created the next time `GetTexture()` is called.

GetTextWidth()

GetTextHeight()

Return the size of the text bounding box after processing word wrap. This allows determining the size of the actual visible text, rather than the box used for word wrap bounds.

IWEBGLTEXTURE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/graphics-interfaces/iwebgltexture

The `IWebGLTexture` interface represents a texture in the WebGL renderer.

This interface cannot be directly constructed. It is only available through other APIs.

Methods

GetWidth()

GetHeight()

Return the width or height of the texture. Note this refers to the source texture. Construct's in-editor spritesheeting engine means the texture could be significantly larger than an object's image to be rendered from it.

ILANG INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/misc-interfaces/ilang

The `ILang` interface allows for looking up translated strings in the language file.

Only look up language strings from your own addon. Any other strings in the language file are subject to change at any time.

Methods

PushContext(prefix)

Push a prefix to the context stack. For example `PushContext("foo")` followed by `Get(".bar")` will return the same string as if `Get("foo.bar")` were used. Pushing a context beginning with a dot will append to the current prefix, but pushing an entry not beginning with a dot will reset the current prefix. **Note:** be sure to always call `PopContext()` afterwards.

PopContext()

Pop a prefix from the context stack.

Get(context)

Look up a string in the language file. If the context begins with a dot, it is relative to the current context prefix. Otherwise it is treated as an absolute context.

For convenience Construct also adds this method as a global function `self.lang()`.

IZIPFILE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/misc-interfaces/izipfile

The `IZipFile` interface represents a zip file in the SDK. It allows access to the file list and reading files contained within the zip.

Methods

PathExists(path)

Return a boolean indicating if a given path exists within the zip.

GetFileList()

Return an array of all file paths contained within the zip.

GetFirstEntryWithExtension(ext)

Return a IZipFileEntry representing the first entry found with a given file extension, or `null` if none was found.

GetEntry(path)

Return a IZipFileEntry representing the file at the given path. If the path does not exist in the zip, this returns `null`.

ReadText(entry)

Return a promise that resolves with the contents of the given IZipFileEntry read as plain text.

ReadJson(entry)

Return a promise that resolves with the contents of the given IZipFileEntry, read as plain text and then parsed as JSON.

ReadBlob(entry)

Return a promise that resolves with the contents of the given IZipFileEntry, read as a `Blob`. (This is a raw binary format that can be read with other JavaScript APIs.)

The returned blob will have `name` and `lastModified` properties added, reflecting the properties of the file in the zip.

IZIPFILEENTRY INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/misc-interfaces/izipfileentry

The `IZipFileEntry` interface is an opaque reference to an file entry in IZipFile. It has no methods - instead, simply pass it to one of the other methods in IZipFile.

IEVENTBLOCK INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/model-interfaces/ieventblock

The `IEventBlock` interface represents an event block in the event sheet. Event blocks are the most important kind of event, and consist of a number of conditions, actions to run when the conditions are met, and optionally further sub-events. It derives from IEventParentRow.

Creating an event block

The following code sample demonstrates the calls necessary to add an *On start of layout* event to the associated event sheet for a given ILayoutView. This is useful with the Custom Importer API, since the AddDragDropFileImportHandler callback provides the ILayoutView that content was dropped in to.

```
// Note: this code is assumed to be in an async function
// First get the associated event sheet for the layout view
const eventSheet = layoutView.GetLayout().GetEventSheet();
if (eventSheet) // note the layout may not have an event sheet
{
	// Get the IObjectType for the System plugin
	const systemType = eventSheet.GetProject().GetSystemType();

	// Create an empty event block at the root level of the event sheet
	const eventBlock = await eventSheet.GetRoot().AddEventBlock();

	// Add an 'On start of layout' condition
	eventBlock.AddCondition(systemType, null, "on-start-of-layout");

	// Example code for adding a 'Set position' action
	//eventBlock.AddAction(iObjectType, null, "set-position", [100, 200]);
}
```

Finding condition and action IDs

Some developer methods are available to explore the list of condition and action IDs that can be used to create events with. See Finding addon IDs for more information.

Methods

AddCondition(iObjectClass, reserved, cndId, params)

AddAction(iObjectClass, reserved, actId, params)

Add a condition or action to this event block. These methods are very similar so they are documented together. *iObjectClass* must be an IObjectClass (i.e. an IObjectType or IFamily) to create the condition and action for. The next parameter is reserved for future use; you must pass `null`. *cndId* or *actId* must be a string specifying the condition or action to create; for example the System *On start of layout* condition ID is `"on-start-of-layout"`. If the condition or action uses any parameters, then *params* must be an array with enough elements for every parameter. Each parameter can be a string, number, boolean or IObjectType. Expression parameters use a string, which can be any valid expression (including calculations like `"1+1"` for number parameters); if you pass a number, it will be converted to a string. IObjectClass can also be passed for object parameters.

IEVENTPARENTROW INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/model-interfaces/ieventparentrow

The `IEventParentRow` interface is a base class representing any row in the event sheet that can have other events nested beneath it. For example an event group is a parent row since it can have other events nested inside it, but an event comment is not a parent row, because nothing can be nested inside it. Note that the root node of the event sheet is a parent row.

Methods

async AddEventBlock()

Add an empty child event block, with no conditions or actions. Returns a promise resolving with the created IEventBlock.

IEVENTSHEET INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/model-interfaces/ieventsheet

The `IEventSheet` interface represents an event sheet in the project model.

Since events can be nested underneath each other, they are represented as a tree. `GetRoot()` returns the root node of the tree.

Methods

GetProject()

Return the associated IProject.

GetName()

Return the name of the event sheet.

GetRoot()

Return the root node of the event sheet. This is an IEventParentRow representing the top level of the event sheet.

ILAYER INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/model-interfaces/ilayer

The `ILayer` interface represents a layer in the project model.

Methods

GetName()

Return the name of the layer.

GetLayout()

Return the ILayout this layer belongs to.

ILAYOUT INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/model-interfaces/ilayout

The `ILayout` interface represents a layout in the project model. Note that ILayoutView represents the editor view, but ILayout represents the layout in the project.

Methods

GetProject()

Return the associated IProject.

GetName()

Return the name of the layout.

GetAllLayers()

Return an array of ILayer representing all the layers on this layout.

GetEventSheet()

Return the IEventSheet assigned for this layout. Note that layouts do not have to have an event sheet assigned, so this can return `null`.

IPROJECT INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/model-interfaces/iproject

The `IProject` interface provides access to a project from the SDK.

Methods

GetName()

Return the project name.

GetObjectTypeByName(name)

GetFamilyByName(name)

GetObjectClassByName(name)

Look up an object class by a case-insensitive string of its name, returning either an IObjectType (for *GetObjectTypeByName*), an IFamily (for *GetFamilyByName*), or either (for *GetObjectClassByName*), or `null` if not found.

GetObjectClassBySID(sid)

Look up an object class by its SID (Serialization ID), returning either an IObjectType or IFamily, or `null` if not found.

"object" type properties store the SID of the chosen object class, so this method allows identifying the corresponding object class in the editor.

CreateObjectType(pluginId, name)

Add a new object type to the project. Returns a promise that resolves with an IObjectType representing the new object type. See Finding addon IDs to get a list of possible plugin IDs that can be used. `name` is the requested name to use for the object type. If the name is free, it will be used directly; however if the name is already in use, Construct will change the name to one which is available. Call `GetName()` on the returned IObjectType to determine what name it was assigned.

GetSystemType()

Return an IObjectType representing the System plugin, which exists in every project.

GetSingleGlobalObjectType(pluginId)

Return an IObjectType representing a single-global plugin in the project. Returns `null` if the given plugin ID does not exist, is not a single-global plugin, or the plugin has not been added to the project. See Finding addon IDs to get a list of possible plugin IDs that can be used.

CreateFamily(name, members)

Create a new family in the project. *name* is an optional family name (pass `null` to use a default name). *members* must be an array of IObjectType representing the object types to add to the family. Families must be created with at least one object type, and if they have multiple object types, they must all be from the same kind of plugin (e.g. all Sprites). Returns an IFamily representing the created family.

GetInstanceByUID(uid)

Look up an instance by its UID (Unique ID), returning either a IObjectInstance or IWorldInstance depending on the kind of instance, or `null` if not found.

GetProjectFileByName(name)

Look up a project file by a case-insensitive string of its filename, returning an IProjectFile if found, else `null`.

GetProjectFileByExportPath(path)

Look up a project file by a string of its path after export, returning an IProjectFile if found, else `null`. Note the path after export depends on the project *Export file structure* setting. In the legacy *Flat* mode, file paths are always at the root level (even if in a subfolder in the Project Bar) and names are case-insensitive. In the modern *Folders* mode, file paths correspond to the subfolders in the Project Bar and are case-sensitive. This method is useful for being able to identify in the editor the project file that corresponds to a relative URL in the runtime.

GetProjectFileBySID(sid)

Look up a project file by its SID (Serialization ID), returning an IProjectFile if found, else `null`.

> *"projectfile" type properties store the SID of the chosen project file, so this method allows identifying the corresponding project file in the editor.*

AddOrReplaceProjectFile(blob, filename, kind = "general")

Create a new project file in the project, or replace the content of the file if it already exists, using a `Blob` for the file content and a string for the filename. The *kind* defaults to `"general"`, which causes the file to be placed in the "Files" folder in the Project Bar. Other options are `"sound"`, `"music"`, `"video"`, `"font"` and `"icon"`.

ShowImportAudioDialog(fileList)

Bring up the *Import audio* dialog to import a list of audio files given in `fileList`. This will automatically transcode the audio files to WebM Opus (when supported for the audio formats), which is the main format Construct uses. Prefer importing PCM WAV files to ensure transcoding is supported and is lossless. The file list should be an array of `Blob` or `File`; if blobs, then ensure a `name` property is assigned to the blob object to indicate the intended filename.

> *Reading blobs from IZipFile automatically assigns a name property so the blobs can be directly passed to this method.*

EnsureFontLoaded(f)

Make sure a given font name is loaded so it can be used when drawing text. This is necessary for plugins that render text.

UndoPointChangeObjectInstancesProperty(instances, propertyId)

Create a new undo point that undoes changes to `propertyId`. `instances` must be either an IObjectInstance or an array of IObjectInstance. Call this method before changing an instance's property value and the action will be undoable.

IPROJECTFILE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/model-interfaces/iprojectfile

The `IProjectFile` interface represents a project file added in the Project Bar in Construct.

Methods

GetName()

Return the filename of the project file.

GetPath()

Return the full path to the project file, including any subfolders, separated by forwards slashes. The path returned matches its location after exporting the project, which may mean the subfolder path does not exactly match its location in the Project Bar. For example a file named *music.webm* in the Music folder of the Project Bar will have a path of `"media/music.webm"`. However files in the general-purpose *Files* folder are always relative to the root folder, so no additional subfolders will appear in the path.

GetProject()

Return the IProject the project file belongs to.

GetBlob()

Return a Blob representing the contents of the file. The standard web APIs for reading blobs can be used to access the content.

IANIMATION INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/ianimation

The `IAnimation` interface represents an animation within an animated object type. This is only applicable to animated plugins such as Sprite.

Methods

GetName()

Return a string of the animation name.

GetObjectType()

Return the IObjectType that this animation belongs to.

GetFrames()

Return an array of IAnimationFrame representing the frames in this animation.

AddFrame(blob, width, height)

Add a new animation frame to the animation. All the parameters are optional. There are four overloads of this method:

1. No parameters passed: add an empty animation frame with a default size
2. Blob passed with no size: use the blob as the animation frame image file, and decompress the image to determine the size
3. Blob passed with size: use the blob as the animation frame image file and use the provided size (which must be correct) to skip having to decompress the image to find its size
4. No blob passed but size provided: use the size for the empty animation frame

The method returns a promise that resolves with the added IAnimationFrame.

SetSpeed(s)

GetSpeed()

Set and get the animation speed in animation frames per second.

SetLooping(l)

IsLooping()

Set and get the looping flag for the animation, indicating if the animation will repeat.

SetPingPong(p)

IsPingPong()

Set and get the ping-pong flag for the animation, indicating if the animation will repeat alternating forwards and backwards.

SetRepeatCount(r)

GetRepeatCount()

Set and get the number of times the animation is set to repeat.

SetRepeatTo(f)

GetRepeatTo()

Set and get the animation frame index to return to when repeating the animation. This must be a valid index.

Delete()

Immediately deletes this animation from its object without any confirmation prompt. This cannot be undone.

The last animation is not allowed to be deleted. Construct requires that animated objects have at least one animation.

Use this with care as it does not warn the user and cannot be undone.

IANIMATIONFRAME INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/ianimationframe

The `IAnimationFrame` interface represents an image for an object type. Despite the name, this interface is also used if the plugin uses a single image, like Tiled Background does.

Note `IAnimationFrame` cannot be directly rendered. You must first create a texture from it.

> *This interface provides methods for loading a texture, but you don't normally need to use them. Simply pass the `IAnimationFrame` to IWorldInstanceBase's `GetTexture()` method, which provides a default implementation using these methods.*

Methods

GetObjectType()

Return the associated IObjectType interface.

GetWidth()

GetHeight()

Return the size of the image, in pixels.

GetCachedWebGLTexture()

Return an IWebGLTexture interface if the texture is already loaded, else `null`.

GetTexRect()

Return an SDK.Rect representing the texture co-ordinates of this image on the loaded texture. This can only be called if `GetCachedWebGLTexture()` returned a texture.

async LoadWebGLTexture()

Start asynchronously loading a texture for this image. This should only be used if `GetCachedWebGLTexture()` returned `null`. Returns a promise that resolves with a IWebGLTexture representing the loaded texture.

GetBlob()

Return a Blob representing the current image content of the animation frame as a compressed image in PNG, WebP or AVIF format.

ReplaceBlobAndDecode(blob)

Replace the image content of the animation frame with the given blob. The blob will be decoded as an image and the previous content of the animation frame overwritten with the image content of the blob. This may also change the size of the frame. Returns a promise that resolves when the image content has been updated.

SetDuration(d)

GetDuration()

Set and get the individual frame duration. This is a multiplier, e.g. 1 for normal speed, 2 for twice as long, etc.

SetOriginX(x)

SetOriginY(y)

GetOriginX()

GetOriginY()

Set and get the origin for this image. The origin is specified in texture co-ordinates, i.e. from 0 to 1. The default is 0.5, indicating the middle of the image.

GetImagePoints()

Return an array of IImagePoint representing the image points added to the image.

AddImagePoint(name, x, y)

Add a new image point to the image with the specified name and position. As with the origin, image point positions are specified in texture co-ordinates, i.e. from 0 to 1. Returns an IImagePoint representing the added image point.

GetCollisionPoly()

Return the ICollisionPoly representing the image's collision polygon.

Delete()

Immediately deletes this frame from its animation without any confirmation prompt. This cannot be undone.

The last frame is not allowed to be deleted. Construct requires that animations have at *least one frame.*

Use this with care as it does not warn the user and cannot be undone.

IBEHAVIORINSTANCE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/ibehaviorinstance

The `IBehaviorInstance` interface represents a behavior instance in Construct.

Methods

GetProject()

Return the IProject representing the behavior instance's associated project.

GetObjectInstance()

Returns an IObjectInstance or IWorldInstance (depending on the type of object) of the object instance associated with this behavior instance.

GetPropertyValue(id)

Get the value of a behavior property for this behavior instance by its property ID.

SetPropertyValue(id, value)

Set the value of a behavior property for this instance by its property ID.

GetExternalSdkInstance()

Return the custom behavior-specific SDK editor instance class for this behavior instance, which will be a derivative of IBehaviorInstanceBase. For example if called for a behavior instance of the addon SDK's sample behavior, this would return the `MyCustomBehaviorInstance` class. This method can only be used for installed addons - it will return `null` for any built-in behaviors.

> *Be careful if taking a dependency on a behavior class provided by another developer. Make sure to only use documented and supported methods. If you use features which are changed or removed by a future addon update, then your addon may crash the editor. Scirra will not provide support for crashes involving third-party addons and we will direct affected users to contact the addon developer.*

IBEHAVIORTYPE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/ibehaviortype

The `IBehaviorType` interface represents a behavior type in Construct. A behavior type is the behavior equivalent of an object type: when a behavior is added to an object type, there is one behavior type created on the object type, and one behavior instance created per object instance.

Methods

GetProject()

Return the IProject representing the behavior type's associated project.

GetName()

Returns a string of the behavior type name.

ICOLLISIONPOLY INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/icollisionpoly

The `ICollisionPoly` interface represents the collision polygon for an IAnimationFrame. It is represented as a list of numbers representing points connected in a loop. As with image points, the collision polygon points are specified in texture co-ordinates, i.e. from 0 to 1.

Methods

Reset()

Reset the collision polygon to the default, which matches the bounding box of the object.

IsDefault()

Return a boolean indicating if the collision polygon is set to the default, matching the bounding box of the object.

GetPoints()

Return an array of numbers representing the points in the collision polygon. The array elements are alternating X and Y components for the points, so its size is always even. The collision polygon is guaranteed to have at least three points.

SetPoints(arr)

Set the collision polygon points by passing an array of numbers. The array elements must be alternating X and Y components for the points, so its size must be even. There must be at least three points in a collision polygon, therefore the passed array must have at least 6 elements.

ICONTAINER INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/icontainer

The `IContainer` interface represents a container in Construct, which is a group of object types that are always created, destroyed and picked together.

Methods

GetMembers()

Return an array of IObjectType representing the object types in the container. Containers always have at least two members.

SetSelectMode(m)

GetSelectMode()

Set or get the select mode of the container, corresponding to the *Select mode* property in Construct. Allowed modes are `"normal"`, `"all"` and `"wrap"`.

RemoveObjectType(objectType)

Remove a member IObjectType from this container.

> *A container must have at least two object types. If the second-last member is removed, the container becomes inactive and is effectively deleted. The last remaining member will also act as if it's no longer in a container.*

IsActive()

Return a boolean indicating if the container is active. It becomes inactive if there are fewer than the minimum required two members, at which point it is effectively deleted.

IFAMILY INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/ifamily

The `IFamily` interface represents a family in Construct, which is a group of object types that can be treated as one. All object types in the family must be from the same plugin. It derives from IObjectClass. Families can be created in the SDK using IProject.`CreateFamily()`.

Methods

GetMembers()

Return an array of IObjectType representing the object types in the family.

SetMembers(objectTypes)

Set the members of the family by passing an array of IObjectType. Note all the specified object types must be compatible with the family, including using the same plugin, and not having any naming conflicts between instance variables, behaviors and effects.

IIMAGEPOINT INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/iimagepoint

The `IImagePoint` interface represents an image point on an IAnimationFrame.

Methods

GetAnimationFrame()

Return the associated IAnimationFrame.

SetName(name)

GetName()

Set or get the name of the image point.

SetX(x)

SetY(y)

GetX()

GetY()

Set or get the position of the image point in texture co-ordinates, i.e. from 0 to 1.

IOBJECTCLASS INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/iobjectclass

The `IObjectClass` interface is the base class of IObjectType and IFamily.

`IObjectClass` cannot be created directly. However any parameter that accepts an `IObjectClass` can accept any derivative, i.e. an object type or a family.

Methods

GetName()

Return the name of the object class.

GetProject()

Return the IProject representing the object class's associated project.

Delete()

Immediately deletes this object class from the project without any confirmation prompt. All events referencing it will also be removed. This cannot be undone.

Use this with care as it does not warn the user and cannot be undone.

IOBJECTINSTANCE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/iobjectinstance

The `IObjectInstance` interface represents an instance in Construct.

Methods

GetProject()

Return the IProject representing the instance's associated project.

GetObjectType()

Return the associated IObjectType interface for this instance.

GetUID()

Return the UID (unique ID) the editor has assigned to this instance.

GetPropertyValue(id)

Get the value of a plugin property for this instance by its property ID. Color properties return a SDK.Color.

SetPropertyValue(id, value)

Set the value of a plugin property for this instance by its property ID. For color properties a SDK.Color must be passed as the value.

GetExternalSdkInstance()

Return the custom plugin-specific SDK editor instance class for this object instance, which will be a derivative of IInstanceBase. For example if called for an instance of the addon SDK's *drawingPlugin* sample, this would return the `MyDrawingInstance` class. This method can only be used for installed addons - it will return `null` for any built-in plugins.

Be careful if taking a dependency on a class provided by another developer. Make sure to only use documented and supported methods. If you use features which are changed or removed by a future addon update, then your addon may crash the editor. Scirra will not provide support for crashes involving third-party addons and we will direct affected users to contact the addon developer.

IOBJECTTYPE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/iobjecttype

The `IObjectType` interface represents an object type in Construct. It derives from IObjectClass.

Methods

GetImage()

Return an IAnimationFrame representing the object type's image. The plugin must have specified `SetHasImage(true)` in IPluginInfo.

EditImage()

Open the Animations Editor in Construct to edit the object's image. The plugin must have specified `SetHasImage(true)`.

GetAnimations()

Return an array of IAnimation representing the animations in the object type. Note this is only applicable to animated plugin types, e.g. Sprite.

AddAnimation(animName, frameBlob, frameWidth, frameHeight)

Add a new animation to the object type. The object type's plugin must be animated (e.g. the Sprite plugin). Animations must have at least one frame, so this method also adds an animation frame. The *frameBlob*, *frameWidth* and *frameHeight* parameters are all optional: if they are omitted, Construct will add a default empty animation frame. If they are provided they are interpreted the same way as IAnimation.AddFrame(); see the linked documentation for more details. The call returns a promise that resolves with the newly created IAnimation.

GetAllInstances()

Return an array of all IObjectInstances or IWorldInstances of this object type in the project.

The returned instances may be placed on different layouts.

CreateWorldInstance(layer)

Create a new instance from this object type, and add it to the given ILayer. Returns a new IWorldInstance interface representing the new instance. Note this method is only applicable to `"world"` type plugins.

IsInContainer()

Return a boolean indicating if the object type belongs to a container.

GetContainer()

Return the IContainer the object type belongs to if any, else `null`.

CreateContainer(initialObjectTypes)

Create a new container using an array of IObjectType for the members of the container. The array must include the IObjectType this call is made on, must contain at least two elements, and the object type must not already be in a container. Returns an IContainer representing the created container.

IWORLDINSTANCE INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/object-interfaces/iworldinstance

The `IWorldInstance` interface represents an instance of a "world" type plugin in Construct. It derives from IObjectInstance.

Methods

GetLayer()

Return the ILayer this instance belongs to.

GetLayout()

Return the ILayout this instance belongs to.

GetBoundingBox()

Returns an SDK.Rect representing the bounding box of the instance in the layout.

GetQuad()

Returns an SDK.Quad representing the bounding quad of the instance in the layout.

GetColor()

Returns an SDK.Color representing the premultiplied color of the instance. This combines the instance's color tint with its opacity in the alpha channel.

SetOpacity(o)

GetOpacity()

Set or get the alpha component of the instance's color, representing its opacity, in the 0-1 range.

SetX(x)

SetY(y)

SetXY(x, y)

GetX()

GetY()

Set and get the position of this instance in layout co-ordinates.

SetZElevation(z)

GetZElevation()

Set and get the Z elevation (position on Z axis) of this instance. Note this is relative to the Z elevation of the layer the instance is on.

GetTotalZElevation()

Get the total Z elevation of this instance, which is its own Z elevation added to the Z elevation of the layer it is on.

SetAngle(a)

GetAngle()

Set and get the angle of the instance, in radians.

SetWidth(w)

SetHeight(h)

SetSize(w, h)

GetWidth()

GetHeight()

Set and get the size of the instance, in pixels.

SetOriginX(x)

SetOriginY(y)

SetOrigin(x, y)

GetOriginX()

GetOriginY()

Set and get the current origin of the instance in the layout. Note this is normalized to a [0, 1] range, e.g. 0.5 is the middle.

ApplyBlendMode(iRenderer)

Sets the current blend mode of the given IWebGLRenderer according to the *Blend mode* property of the instance in Construct. This is only relevant if the plugin specifies that it supports effects. Use this in the `Draw()` method to set the correct blend mode.

ILAYOUTVIEW INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/ui-interfaces/ilayoutview

The `ILayoutView` interface provides access to a Layout View from the SDK. Note that this interface represents the editor view; the ILayout interface provides the interface to the actual layout in the project model.

Methods

GetProject()

Return the IProject representing the project associated with this Layout View.

GetLayout()

Return an ILayout representing the layout in the project model that this Layout View is showing.

GetActiveLayer()

Return an ILayer representing the current active layer selected in this Layout View.

GetZoomFactor()

Return the current zoom factor of the Layout View. For example 1 represents 100% zoom, 0.5 represents 50% zoom, etc.

LayoutToClientDeviceX(x)

LayoutToClientDeviceY(y)

Convert from layout co-ordinates to device pixel co-ordinates in the layout view canvas. This is useful for rendering at device pixel sizes after calling `SetDeviceTransform()`.

SetDeviceTransform(iRenderer)

Set the given IWebGLRenderer to a device pixel co-ordinate transform. This means co-ordinates used for rendering are based on device pixel co-ordinates relative to the layout view canvas, rather than layout co-ordinates.

SetDefaultTransform(iRenderer)

Set the given IWebGLRenderer to a layout co-ordinate transform. This is the default and should be restored after using `SetDeviceTransform()`.

Refresh()

Schedules the layout view to be redrawn at the next animation frame. Avoid unnecessarily refreshing the layout view, such as refreshing on a timer, since this can waste battery life.

UTIL INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/ui-interfaces/utilities

The `SDK.UI.Util` interface provides access to various user interface utilities in the SDK.

Methods

AddDragDropFileHandler(callback, opts)

Register a callback for handling files drag-and-dropped in to the Construct 3 window. This is part of the Custom Importer API, allowing addons to handle importing files in a custom format. The given callback is invoked when a file is dropped in to the Construct 3 window, providing nothing else has handled it first. The callback must return a promise that resolves with `true` if the drop was recognised and imported, otherwise `false` if the drop was not recognised as a supported format (in which case Construct will continue running other handlers).
The `opts` parameter of `AddDragDropFileHandler()` is an options object, which may specify the following:

- `isZipFormat`: boolean indicating to handle dropped zip files only. If `true`, the callback will only be run if Construct recognises the dropped file as a zip file. Consequently the `file` parameter of the callback will be an IZipFile, from which the contents of the zip file can be read. If `false`, the callback will only be run if Construct does *not* recognise the dropped file as a zip file, and consequently the `file` parameter of the callback will be a `Blob`.

- `toLayoutView`: boolean indicating to handle files dropped to an open Layout View only. If `true`, the callback will only be run if a Layout View is open, and the `opts` parameter of the callback will contain information about the Layout View and the drop position. If `false`, the callback will be run regardless of whether a Layout View is open or not, and no further options will be provided to the callback.

The `callback` should have the signature `async function(filename, file, opts)`. The type of the `file` parameter is an IZipFile or Blob depending on the `isZipFormat` option. The `opts` parameter of the callback will provide the following additional details only when the `toLayoutView` option was specified:

- `layoutView`: an ILayoutView interface representing the Layout View that was open when the file was dropped. This also provides access to the associated project, layout, current active layer, and so on.

- `clientX` and `clientY`: the drop position within the window in client co-ordinates.
- `layoutX` and `layoutY`: the drop position within the Layout View in layout co-ordinates. This is the position to create any new instances relevant to.

---

static ShowLongTextPropertyDialog(text, caption)

Show the same dialog used to edit `longtext` properties. This is simply a large multi-line text field in a dialog, allowing for long text strings to be more conveniently edited, since the Properties Bar often can only show a small amount of text. Returns a promise that resolves with `null` if the dialog was cancelled, else a string of the text in the dialog.

FINDING ADDON IDS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/finding-addon-ids

Some APIs use special IDs, such as a plugin ID or action ID. The editor provides some methods you can use from the browser console to explore which IDs are available.

Listing addon IDs

The `C3SDK_ListAddonIDs(addonType)` method lists all installed addon IDs in the editor. The *addonType* parameter must be `"plugin"` or `"behavior"`. For example, load Construct 3 in a browser, open the browser console (usually F12), and enter the following call to list all plugin IDs:

```
C3SDK_ListAddonIDs("plugin")
```

This may take a moment to load, but then it will log to the console the ID for every installed plugin, along with its name as used in the editor (since this is sometimes different to the ID for legacy reasons).

Listing ACE IDs

The `C3SDK_ListACEIDs(addonType, addonId, aceType)` method lists the IDs for an addon's actions, conditions or expressions (ACEs). The parameters it takes are:

- *addonType*: as before, either `"plugin"` or `"behavior"`.
- *addonId*: the ID of the addon, which can be found using `C3SDK_ListAddonIDs()`.
- *aceType*: one of `"actions"`, `"conditions"` or `"expressions"`, determining which to list.

For example the following call will list all of the Sprite plugin's action IDs. The parameters each takes are also listed.

```
C3SDK_ListACEIDs("plugin", "Sprite", "actions")
```

IBEHAVIORINFO INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/ibehaviorinfo

IBehaviorInfo defines the configuration for a behavior. It is typically accessed in the behavior constructor via `this._info`.

Methods

SetName(name)

Set the name of the addon. Typically this is read from the language file.

SetDescription(description)

Set the description of the addon. Typically this is read from the language file.

SetVersion(version)

Set the version string of the addon, in A.B.C.D form. Typically this is set to the `BEHAVIOR_VERSION` constant.

SetCategory(category)

Set the category of the addon. Typically this is set to the `BEHAVIOR_CATEGORY` constant. It must be one of `"attributes"`, `"general"`, `"movements"`, `"other"`.

SetAuthor(author)

Set a string identifying the author of the addon.

SetHelpUrl(url)

Set a string specifying a URL where the user can view help and documentation resources for the addon.

SetIcon(url, type)

Set the addon icon URL and type. By default the URL is `"icon.svg"` and the type is `"image/svg+xml"`. It is recommended to leave this at the default and use an SVG icon, since it will scale well to any display size or density. However you can change your addon to load a PNG icon with `SetIcon("icon.png", "image/png")`.

SetIsOnlyOneAllowed(isOnlyOneAllowed)

Set a boolean of whether the behavior is allowed to be added more than once to the same object. The default is `false`, which means the behavior can be added multiple times to the same object. Set to `true` to only allow it to be added once to each object.

SetIsDeprecated(isDeprecated)

Set a boolean of whether the addon is deprecated or not. If you wish to replace your addon with another one, the old one can be deprecated with `SetIsDeprecated(true)`. This makes it invisible in the editor so it cannot be used in new projects; however old projects with the addon already added can continue to load and work as they did before. This discourages use of the deprecated addon without breaking existing projects that use it.

SetCanBeBundled(canBeBundled)

Pass `false` to prevent the addon from being bundled via the *Bundle addons* project property. By default all addons may be bundled with a project, and it is recommended to leave this enabled for best user convenience. However if you publish a commercial addon and want to prevent it being distributed by project-bundling, you may wish to disable this.

SetProperties(propertiesArray)

Set the available addon properties by passing an array of PluginProperty. See Configuring Behaviors for more information.

AddCordovaPluginReference(opts)

Add a dependency on a Cordova plugin, that will be included when using the Cordova exporter. For more information see Specifying dependencies.

AddFileDependency(opts)

Add a dependency on another file included in the addon. For more information see Specifying dependencies.

AddRemoteScriptDependency(url) Not recommended

Add a script dependency to a remote URL (on a different origin). For more information see Specifying dependencies.

SetC3RuntimeScripts(arr)

Pass an array of strings to set the list of runtime scripts the addon uses. The default list is the following, and note that this method entirely replaces it: *"c3runtime/behavior.js"*, *"c3runtime/type.js"*, *"c3runtime/instance.js"*, *"c3runtime/conditions.js"*, *"c3runtime/actions.js"*, *"c3runtime/expressions.js"*.

AddC3RuntimeScript(path)

Add a single runtime script path to the existing list of runtime scripts the addon uses, e.g. "c3runtime/additionalScript.js".

SetRuntimeModuleMainScript(path)

Set the main script that the runtime loads as a module. When this method is called, Construct will only load that script, and it is expected that all your other scripts are imported in the main script. If this method is not called, Construct automatically generates a main script that imports every single runtime script - but note that makes it difficult to use modules properly. See runtime scripts for more information.

SetScriptInterfaceNames(opts)

Use this method to tell Construct the names of your script interface classes. This is necessary to generate the correct TypeScript definition files. `opts` is an object which allows specifying the names for the `instance` , `behaviorType` and `behavior` interface names as necessary, e.g.:

```
this._info.SetScriptInterfaceNames({
        instance: "IBulletBehaviorInstance"
});
```

SetTypeScriptDefinitionFiles(arr)

Specify an array of TypeScript definition files (.d.ts) your addon provides. This should be used to provide full TypeScript definitions of any script interfaces your addon provides, which is necessary for projects using TypeScript with your addon. Example:

```
this._info.SetTypeScriptDefinitionFiles(["c3runtime/IBulletBehaviorInstance.d.ts"]);
```

IPLUGININFO INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/iplugininfo

IPluginInfo defines the configuration for a plugin. It is typically accessed in the plugin constructor via `this._info`.

Methods

SetName(name)

Set the name of the addon. Typically this is read from the language file.

SetDescription(description)

Set the description of the addon. Typically this is read from the language file.

SetVersion(version)

Set the version string of the addon, in A.B.C.D form. Typically this is set to the `PLUGIN_VERSION` constant.

SetCategory(category)

Set the category of the addon. Typically this is set to the `PLUGIN_CATEGORY` constant. It must be one of `"data-and-storage"`, `"form-controls"`, `"general"`, `"input"`, `"media"`, `"monetisation"`, `"platform-specific"`, `"web"`, `"other"`.

SetAuthor(author)

Set a string identifying the author of the addon.

SetHelpUrl(url)

Set a string specifying a URL where the user can view help and documentation resources for the addon. The website should be hosted with HTTPS.

SetPluginType(type)

Set the plugin type. This can be `"object"` or `"world"`. The world typeh represents a plugin that appears in the Layout View, whereas the object type represents a hidden plugin, similar to the Audio plugin (a single-global type) or Dictionary. World type plugins must derive from `SDK.IWorldInstanceBase` instead of `SDK.IInstanceBase` and implement a `Draw()` method.

SetIcon(url, type)

Set the addon icon URL and type. By default the URL is `"icon.svg"` and the type is `"image/svg+xml"`. It is recommended to leave this at the default and use an SVG icon, since it will scale well to any display size or density. However you can change your addon to load a PNG icon with `SetIcon("icon.png", "image/png")`.

SetIsResizable(isResizable)

For `"world"` type plugins only. Pass `true` to enable resizing instances in the Layout View.

SetIsRotatable(isRotatable)

For `"world"` type plugins only. Pass `true` to enable the *Angle* property and rotating instances in the Layout View.

SetIs3D(is3d)

For `"world"` type plugins only. Pass `true` to specify that this plugin renders in 3D. This will cause the presence of the plugin in a project to enable 3D rendering when the project *Rendering mode* property is set to *Auto* (which is the default setting).

SetHasImage(hasImage)

For `"world"` type plugins only. Pass `true` to add a single editable image, such as used by the Tiled Background plugin.

SetDefaultImageURL(url)

For plugins that use a single editable image only. Set the URL to an image file in your addon to use as the default image when the object is added to a project, e.g. `"default.png"`.

> *When using developer mode addons, remember to add the image file to the file list in addon.json.*

SetIsTiled(isTiled)

For `"world"` type plugins only. Pass `true` to indicate that the image is intended to be tiled. This adjusts the texture wrapping mode when Construct creates a texture for its image.

SetIsDeprecated(isDeprecated)

Set a boolean of whether the addon is deprecated or not. If you wish to replace your addon with another one, the old one can be deprecated with `SetIsDeprecated(true)`. This makes it invisible in the editor so it cannot be used in new projects; however old projects with the addon already added can continue to load and work as they did before. This discourages use of the deprecated addon without breaking existing projects that use it.

SetIsSingleGlobal(isSingleGlobal)

Pass `true` to set the plugin to be a *single-global* type. The plugin type must be `"object"`. Single-global plugins can only be added once to a project, and they then have a single permanent global instance available throughout the project. This is the mode that plugins like Touch and Audio use.

SetSupportsZElevation(supportsZElevation)

Pass `true` to allow using Z elevation with this plugin. The plugin type must be `"world"`. By default the renderer applies the Z elevation before calling the `Draw()` method on an instance, which in many cases is sufficient to handle rendering Z elevation correctly, but be sure to take in to account Z elevation in the drawing method if it does more complex rendering.

> *`AddCommonZOrderACEs()` will add common ACEs for Z elevation if supported.*

SetSupportsColor(supportsColor)

Pass `true` to allow using the built-in color property to tint the object appearance. The plugin type must be `"world"`. By default the renderer sets the color before calling the `Draw()` method on an instance, which in many cases is sufficient to handle rendering with the applied color, but be sure to take in to account the instance color in the drawing method if it does more complex rendering.

> *`AddCommonAppearanceACEs()` will add common ACEs for color if supported.*

SetSupportsEffects(supportsEffects)

Pass `true` to allow using effects, including the *Blend mode* property, with this plugin. The plugin type must be `"world"`. If the plugin does not simply draw a texture the size of the object (as Sprite does), you should also call `SetMustPreDraw(true)`.

SetMustPreDraw(mustPreDraw)

Pass `true` to disable an optimisation in the effects engine for objects that simply draw a texture the size of the object (e.g. Sprite). This is necessary for effects to render correctly if the plugin draws anything other than the equivalent the Sprite plugin would.

SetCanBeBundled(canBeBundled)

Pass `false` to prevent the addon from being bundled via the *Bundle addons* project property. By default all addons may be bundled with a project, and it is recommended to leave this enabled for best user convenience. However if you publish a commercial addon and want to prevent it being distributed by project-bundling, you may wish to disable this.

AddCommonPositionACEs()

AddCommonSceneGraphACEs()

AddCommonSizeACEs()

AddCommonAngleACEs()

AddCommonAppearanceACEs()

AddCommonZOrderACEs()

Add common built-in sets of actions, conditions and expressions (ACEs) to the plugin relating to various built-in features.

> ***Note:** if adding common scene graph ACEs, your plugin must be prepared to handle being added in to a scene-graph hierarchy, and having its position, size and angle controlled automatically. It must also support all the properties modifiable by hierarchies, otherwise the scene graph feature may not work as expected.*

SetProperties(propertiesArray)

Set the available addon properties by passing an array of PluginProperty. See Configuring Plugins for more information.

AddCordovaPluginReference(opts)

Add a dependency on a Cordova plugin, that will be included when using the Cordova exporter. For more information see Specifying dependencies.

AddCordovaResourceFile(opts)

Add a resource file to be included with Cordova exports. For more information see Specifying dependencies.

AddFileDependency(opts)

Add a dependency on another file included in the addon. For more information see Specifying dependencies.

AddRemoteScriptDependency(url, type)

Add a script dependency to a remote URL (on a different origin). By default it loads the URL as a "classic" script; the `type` parameter is optional and can be set to the string `"module"` to load the dependency as a module script instead. For more information see Specifying dependencies.

SetGooglePlayServicesEnabled(enabled)

Pass `true` to enable Google Play Services in Cordova Android exports. `<preference name="GradlePluginGoogleServicesEnabled" value="true" />` will be added in config.xml.

> *Since this can only be configured once, if any plugin in the project specifies to enable Google Play Services, it will be enabled for the entire project.*

SetC3RuntimeScripts(arr)

Pass an array of strings to set the list of runtime scripts the addon uses. The default list is the following, and note that this method entirely replaces it: *"c3runtime/plugin.js"*, *"c3runtime/type.js"*, *"c3runtime/instance.js"*, *"c3runtime/conditions.js"*, *"c3runtime/actions.js"*, *"c3runtime/expressions.js"*.

AddC3RuntimeScript(path)

Add a single runtime script path to the existing list of runtime scripts the addon uses, e.g. "c3runtime/additionalScript.js".

SetRuntimeModuleMainScript(path)

Set the main script that the runtime loads as a module. When this method is called, Construct will only load that script, and it is expected that all your other scripts are imported in the main script. If this method is not called, Construct automatically generates a main script that imports every single runtime script - but note that makes it difficult to use modules properly. See runtime scripts for more information.

SetDOMSideScripts(arr)

Specify an array of script paths to load in the main document context rather than the runtime context. For more information see the section *DOM calls in the C3 runtime* in Runtime scripts.

SetScriptInterfaceNames(opts)

Use this method to tell Construct the names of your script interface classes. This is necessary to generate the correct TypeScript definition files. `opts` is an object which allows specifying the names for the `instance`, `objectType` and `plugin` interface names as necessary, e.g.:

```
this._info.SetScriptInterfaceNames({
        instance: "ISpriteInstance"
});
```

SetTypeScriptDefinitionFiles(arr)

Specify an array of TypeScript definition files (.d.ts) your addon provides. This should be used to provide full TypeScript definitions of any script interfaces your addon provides, which is necessary for projects using TypeScript with your addon. Example:

```
this._info.SetTypeScriptDefinitionFiles(["c3runtime/ISpriteInstance.d.ts"]);
```

SetWrapperExportProperties(componentId, propertyIds)

For use with single-global plugins using wrapper extensions. Specify an array of property IDs which will have their values exported to package.json under the key `"exported-properties"`. For example the following call:

```
this._info.SetWrapperExportProperties("my-component-id", ["first-property", "second-property
```

will result in the following content being included in the exported package.json:

```
{
	"exported-properties": {
		"my-component-id": {
			"first-property": "...",
			"second-property": "..."
		}
	}
}
```

This can then be read by the wrapper extension using the IApplication method `GetPackageJsonContent()` and parsing it with a library like json.hpp. This allows the wrapper extension to initialize early, before any web content is loaded, while still making use of settings specified in plugin properties.

PLUGINPROPERTY CLASS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/pluginproperty

PluginProperty defines a single property for an addon that will appear in the Properties Bar. Typically an array of PluginProperty is passed to `this._info.SetProperties()`. See Configuring Plugins for more information. Note that despite the name, PluginProperty is also used to define properties for behaviors.

Note properties do not directly define any strings that appear in the editor UI. These are defined in The Language File.

Constructor

```
new SDK.PluginProperty(type, id, initialValue_or_options)
```

type

The type of the property. This can be one of:

- `"integer"` — an integer number property, always rounded to a whole number.
- `"float"` — a floating-point number property.
- `"percent"` — a floating-point number in the range [0-1] represented as a percentage. For example if the user enters 50%, the property will be set to a value of 0.5.
- `"text"` — a field the user can enter a string in to.
- `"longtext"` — the same as `"text"`, but a button with an ellipsis ("...") appears on the right side of the field. The user can click this button to open a dialog to edit a long string more conveniently. This is useful for potentially long content like the project description, or the main text of the Text object.
- `"check"` — a checkbox property, returning a boolean.
- `"font"` — a field which displays the name of a font and provides a button to open a font picker dialog. The property is set to a string of the name of the font.

- `"combo"` — a dropdown list property. The property is set to the zero-based index of the chosen item. The `items` field of the options object must be used to specify the available items.

- `"color"` For plugins only — a color picker property. The initial value must be an array, e.g. `[1, 0, 0]` for red.

- `"object"` For plugins only — an object picker property allowing the user to pick an object class. **Note:** At runtime, this passes a SID (Serialization ID) for the chosen object class, or -1 if none was picked. Use getObjectClassBySid() to look up the corresponding `IObjectClass`.

- `"projectfile"` For plugins only Addon SDK v2 only (r426+) — a dropdown list from which any project file in the project can be chosen. The property value at runtime is a relative path to fetch the project file from. The `"filter"` option can also be specified to filter the list by a file extension, e.g. ".txt" to only list .txt files.

- `"group"` — creates a new group in the Properties Bar. There is no value associated with this property.

- `"link"` For plugins only — creates a clickable link in the Properties Bar. There is no value associated with this property. A `linkCallback` function must be specified in the options object.

- `"info"` — creates a read-only string that cannot be edited. There is no value associated with this property. A `infoCallback` function must be specified in the options object.

id

A string of the ID for the property. This is used in the language file to identify related strings.

initialValue_or_options

For many properties, the only extra information needed is the initial value. For example for a `"float"` parameter this parameter can be a number of the initial value to use for the property. However to configure more options for the property, pass an object instead, and see the section on using an options object below. Some property types require the use of an options object, e.g. `"combo"` requires it to specify the item list.

The options object

If the third parameter of the constructor is an object, use the following properties to specify further configuration of the property.

initialValue

Specify the initial value for the property, since the third parameter is occupied by the options object. Note when using a `"combo"` type this must be a string of the initial item ID, and when using a `"color"` type, this must be a normalized RGB array, e.g. `[1, 0, 0]` for red.

minValue

Specify a minimum value for a numeric property.

maxValue

Specify a maximum value for a numeric property.

items

Only valid with the `"combo"` property type. Specify an array of strings representing the available item IDs in the dropdown list. The actual displayed strings are read from the language file.

dragSpeedMultiplier

Only valid with numeric properties. Pass a ratio to modify how quickly the value changes when it is being dragged up or down. For example passing 2 would cause the value to increase twice as fast as the mouse moves while dragging the value.

allowedPluginIds

For `"object"` type properties only. An array of plugin ID strings to filter the object picker by. This can also contain the special string `"<world>"` to allow any world-type plugin.

filter

Addon SDK v2 only Optional and only valid with the `"projectfile"` type. Set to a file extension including the dot to filter the list of provided project files to only those with the given file extension, e.g. `".txt"`.

linkCallback

For `"link"` type properties only. A function that is called when the link is clicked. The number of calls, and the type of the parameter, are determined by the `callbackType` option.

callbackType

For `"link"` type properties only. Specifies how the link callback function is used. This can be one of the following:

- `"for-each-instance"` default — the callback is run once per selected instance in the Layout View. The callback parameter is an instance of your addon (deriving from `SDK.IWorldInstanceBase`). This is useful for per-instance modifications, such as a link to make all instances their original size.

- `"once-for-type"` — the callback is run once regardless of how many instances are selected in the Layout View. The callback parameter is your addon's object type (deriving from `SDK.ITypeBase`). This is useful for per-type modifications, such as a link to edit the object image.

infoCallback

For `"info"` type properties only. A function that is called to get the value to display as a read-only string. The function is automatically called when any other properties change. The parameter is an instance of your addon, which you can use to read other property values and use them in the returned value.

interpolatable

For `"integer"`, `"float"`, `"percent"`, `"text"`, `"longtext"`, `"check"`, `"combo"` and `"color"` type properties only. Has a default value of `false`, set to `true` so the property can be supported by timelines. In order to fully support timelines it is also needed to follow the Timeline Integration Guide.

SPECIFYING DEPENDENCIES

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/specifying-dependencies

Plugins and behaviors can specify dependencies on additional files, or Cordova plugins for inclusion with the Cordova exporter. Dependencies are added using the `AddFileDependency(opts)` and `AddCordovaPluginReference(opts)` methods on both IPluginInfo and IBehaviorInfo. Remote scripts can also be added with `AddRemoteScriptDependency`, but this is not recommended.

File dependencies

A file dependency refers to another file in the addon. Note the file must be bundled with the addon; you cannot refer to URLs elsewhere on the Internet. There are several kinds of file dependency, which correspond to the `type` property in the options object:

copy-to-output

This simply causes the file to be copied to the output folder when exporting. The file is also available in preview mode. This is useful for bundling additional resources, such as an image file that needs to be loaded at runtime, or a script that is dynamically loaded.

external-dom-script

A script dependency that is included via the addition of an extra `<script>` tag in the exported HTML file. The `scriptType` option can be set to `"module"` to load the script as a module (see below). Note in worker mode the script is loaded in the DOM, so is not directly available to the runtime code in the worker. The script is not minified on export. This is suitable for large external libraries that the addon references.

external-runtime-script

A script dependency that is included via the addition of an extra `<script>` tag in the exported HTML file, or loaded on the worker with `importScripts()` in worker mode. This means the script is always directly available to runtime code. However the dependency must be designed to work in a Web Worker, e.g. not assuming the DOM is present. The script is not minified on export.

external-css

A stylesheet dependency that is included via the addition of an extra `<link rel="stylesheet">` tag in the exported HTML file, in case the addon needs to specify CSS styles.

wrapper-extension

A DLL to be bundled for a wrapper extension. See Wrapper extensions for more details.

To add a file dependency, call `AddFileDependency` with an options object, such as in this example:

```
this._info.AddFileDependency({
        filename: "mydependency.js",
        type: "external-dom-script"
});
```

The options object uses the following properties.

filename

Name of the dependency file in the addon. This must be bundled with the addon; it cannot refer to a URL. It is recommended to bundle the script with your addon, but if you must use a URL, see the section *Remote script dependencies*. The file path is relative to the root (the location of addon.json).

> *For developer mode addons, make sure the dependency file is also included in the `"file-list"` key. For more information see the section on Developer mode addons in Addon metadata.*

type

One of the types described above, e.g. `"external-dom-script"`.

fileType

When `type` is `"copy-to-output"`, this must specify the MIME type of the file. For example if including `"image.png"` as a `"copy-to-output"` dependency, the `fileType` must be set to `"image/png"`.

scriptType

Currently only supported when `type` is `"external-dom-script"`. By default external DOM scripts are loaded as "classic" scripts. This property can be set to the string `"module"` to instead load the external DOM script as a module (i.e. with `<script src="filename.js" type="module"></script>`).

platform

When `type` is `"wrapper-extension"`, this specifies the platform architecture of the DLL. The supported options are `"windows-x86"` (for 32-bit Windows), `"windows-x64"`, `"windows-arm64"`, `"xbox-uwp-x64"` (for Xbox UWP export option only), `"macos-universal"` (for macOS WKWebView, with a universal binary including both Intel and Apple Silicon code), `"linux-x64"`, `"linux-arm"` (for 32-bit ARM) and `"linux-arm64"`.

Cordova plugin dependencies

Addons can specify dependencies on Cordova plugins. These only apply to the Cordova exporter, which covers both Android and iOS. When exporting a Cordova project, the additional Cordova plugin dependencies are automatically included in the exported **config.xml**. This allows convenient integration of a Construct addon with a Cordova plugin.

To add a Cordova plugin dependency, call `AddCordovaPluginReference` with an options object, such as in this example:

```
this._info.AddCordovaPluginReference({
	id: "cordova-plugin-inappbrowser"
});
```

If you wish to provide variables to the Cordova plugin, use the `variables` property of the options object to pass an array of `[variableName, pluginProperty]` pairs. In this case the plugin must also be passed in the `plugin` property. An example is shown below.

```
const property = new SDK.PluginProperty("integer", "test-property", 0);

this._info.SetProperties([
	property
]);

this._info.AddCordovaPluginReference({
	id: "cordova-plugin-inappbrowser",
	plugin: this,
	variables: [
		["MY_VAR", property]
	]
});
```

Cordova plugins that require variables will not compile if the variable is omitted from config.xml.

See Cordova plugin variables for more information.

The options object uses the following properties.

id

The ID of the Cordova plugin to reference.

version Optional

A version spec for the Cordova plugin, e.g. `"1.0.4"`. If this is not specified, the latest version will be used.

platform Optional

Specify a specific platform the Cordova plugin applies to. By default this is `"all"` meaning it will be used in both Android and iOS exports. However it can be set to `"android"` or `"ios"` to only be included when exporting to a specific platform. This is useful to switch between different Cordova plugins on different platforms.

variables Optional

Specify variables to be used with the Cordova plugin as an array of `[variableName, pluginProperty]` pairs. The variable name is bound to a `SDK.PluginProperty`. When the project is exported a variable is added under the plugin reference in config.xml with the given name and a value taken from the specified property. When variables are specified, the `plugin` property must also be set.

plugin Optional

Used to specify the plugin when using variables. Normally this should be set to `this`.

Note for security reasons the Construct mobile app build service does not allow arbitrary Cordova plugins to be used. The build service uses an allowlist of allowed Cordova plugins. If you'd like a Cordova plugin to be added to the allowlist, please file an issue on the Construct issue tracker. Please note we cannot guarantee that Cordova plugins will be allowed, and approval is subject to a security review. Other build systems, including compiling with the Cordova CLI, do not impose this restriction.

Cordova resource file dependencies

Addons can further specify dependencies on additional resource files for Cordova exports. When exporting a Cordova project, the additional Cordova resource file dependencies are automatically included in the exported **config.xml** as `<resource file src="..." target="...">` tags.

To add a Cordova resource file dependency, call `AddCordovaResourceFile` with an options object, such as in this example:

```
this._info.AddCordovaResourceFile({
	src: "myfile.txt"
});
```

This will insert `<resource-file src="myfile.txt">` to the exported config.xml.

More information about how resource files are used in Cordova can be found in the Cordova documentation.

The options object uses the following properties.

src

The `src` attribute of the `resource-file` tag. Location of the file relative to config.xml.

target Optional

The `target` attribute of the `resource-file` tag. Path to where the file will be copied.

platform Optional

Specify a specific platform the Cordova plugin applies to. By default this is `"all"` meaning it will be used in both Android and iOS exports. However it can be set to `"android"` or `"ios"` to only be included when exporting to a specific platform.

Remote script dependencies

Plugins and behaviors may also specify remote script dependencies. These are loaded from a cross-origin URL, e.g. `https://example.com/api.js`.

Avoid using remote script dependencies where possible. They have some drawbacks:

- *Construct 3 games work offline. However remote scripts cannot be cached for offline use, so will fail to load when the user is offline.*
- *Remote scripts can also fail to load due to unreliable connections or service outages.*
- *In some native apps, e.g. Cordova on Android/iOS, the native platform may block any access to URLs that are not on an allowlist of allowed origins. This can cause the script to fail to load unless the user does additional configuration of their app.*

Prefer using a file dependency instead, and bundle the script with your addon. If you must use a remote script, ensure your addon gracefully handles the case the remote script fails to load.

Use `AddRemoteScriptDependency` to add a remote URL to load a script from. The second parameter can optionally be set to `"module"` to load the script as a module. For example:

```
// Load remote "classic" script
this._info.AddRemoteScriptDependency("https://example.com/api.js");

// Load remote module script
this._info.AddRemoteScriptDependency("https://example.com/api-module.js", "module");
```

This will produce the following tags on export, loaded before the runtime:

```
<!-- Classic script -->
<script src="https://example.com/api.js"></script>

<!-- Module script -->
<script src="https://example.com/api-module.js" type="module"></script>
```

*The script URL must **not** use `http:` in its URL. On the modern web this will often be blocked from secure sites as mixed content. You must either use secure HTTPS, or a same-protocol URL like `//example.com/api.js`.*

RUNTIME API REFERENCE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/runtime-reference

With Construct's Addon SDK, the runtime APIs are the same as are available using Construct's scripting feature. For more details see the guide on Runtime scripts.