Solarus quests

1.5

Generated by Doxygen 1.8.11

Contents

Chapter 1

Main Page

This is the official Solarus documentation intended to quest makers.

A quest is a data package containing graphics, sounds, maps, dialogs, scripts and other resources of your game. The Solarus *engine* is an executable binary that can run your *quest*. We describe here the format of all files of a quest.

This documentation is organized in two main parts:

- The Lua scripting API reference that you can use to make a quest. You will need it to program your menus, your items, your map scripts and your enemies.
- The specification of the format of all data files included in a quest. All of them are handled by Solarus Quest Editor, so you normally don't need to edit these files manually.

Chapter 2

Solarus 1.5 - Lua API reference

This is the API specification of Lua functions, methods, callbacks and types defined by Solarus. This documentation page is intented to quest makers who want to write scripts for their maps, items, enemies and menus. For the point of view of the C++ code of the engine, see the documentation of class LuaContext.

Most of the data types defined in the C++ engine (like sprites, map entities, movements, savagames, etc.) are exported as Lua types in the scripting API of Solarus. We give here the full reference of these types and the functions available for each type. The API exports C++ functions and C++ datatypes that may be used by your scripts. Examples of such features are creating a sprite, drawing an image or moving an enemy. In the opposite way, Solarus will also call your own Lua functions (if you define them), for example to notify your script that an enemy has reached an obstacle, that a pressure plate has just been activated or that the hero is talking to a particular non-playing character.

The following script files are loaded by the engine when they exist:

- The main script (`main.lua`): global script that controls the menus (if any) before starting a game and decides to start a game.
- The script of a map (`maps/XXXX.lua`): controls the map `XXXX`. Called when the player enters the map.
- The script of an enemy (`enemies/XXXX.lua`): controls an enemy whose breed is `XXXX`. Called when an enemy of this breed is created on the map.
- The script of an item (`items/XXXX.lua`): controls the equipment item named `XXXX`. Called when a savegame is loaded or created.

All these various scripts run in the same Lua state. In other words, they share global values.

Interactions between your Lua world and the engine are managed through a predefined global table called `sol`. The whole Solarus Lua API is available in the `sol` table. It contains functions, types and values that allow you to interact with the C++ engine.

Most types of the Lua API (game, item, map, entity, movement and sprite) are Lua userdata that have something special: they can also be indexed like tables. This mechanism is used by the engine when it needs to invoke callback methods that you defined on your objects. But you can also extend these objects with your own functions and data. This is very useful in the game and map objects to implement and store everything that is not built-in in the Solarus API: your pause menu, your HUD, a puzzle, or some properties and utility functions specific to your quest.

The following features are defined in the global `sol` table. See the specification page of each feature for more details.

- sol.main: some general-purpose features.
- sol.audio: playing musics and sounds.
- sol.video: setting the video mode.
- sol.input: checking keyboard and joypad state.
- sol.file: directly accessing data files.
- sol.menu: showing various information on the screen.
- sol.language: handling translations.
- sol.timer: making an action later with a delay.
- sol.sprite, sol.surface, sol.text_surface: displaying animated images, fixed images or text, respectively.
- sol.movement: moving objects.
- sol.game: handling data saved (life, equipment, etc.) and running a game.
- sol.item: controls a particular type of equipment item and its behavior.
- sol.map: handling the current map and its properties (only during a game).
- sol.entity: controls entities placed on the map (only during a game), like the hero, enemies, chests, non-playing characters, etc.

2.1 General features

`sol.main` contains general features and utility functions that are global to the execution of the program, no matter if a game or some menus are running.

2.1.1 Functions of sol.main

2.1.1.1 sol.main.get_solarus_version()

Returns the current Solarus version.

The Solarus version includes three numbers: x.y.z, where x is the major version, y is the minor version and z is the patch version.

Changes of major and minor versions may introduce some incompatibilities in the Lua API. Patch versions are only for bug fixing and never introduce incompatibilities. See the `migration guide` for instructions about how to upgrade your quest to the latest version.

- Return value (string): The Solarus version.

2.1.1.2 sol.main.get_quest_format()

Returns the format of this quest.

This corresponds to a version of Solarus with major and minor version numbers (no patch number), for example `"1.5"`.

- Return value (string): The format of this quest.

2.1.1.3 sol.main.load_file(script_name)

Loads a Lua file (but does not run it).

This function is a replacement to the usual Lua function `loadfile()`. The difference is that it looks for a file in the quest tree (which may be a directory or an archive) and also in the quest write directory. The quest write directory is tried first.

- `script_name` (string): Name of the Lua file to load (with or without extension), relative to the data directory or the write directory of your quest.
- Return value (function): A function representing the chunk loaded, or `nil` if the file does not exist or could not be loaded as Lua.

Remarks

Keep in mind that Lua files, as all data files of your quest, may be located inside an archive instead of being regular files. Therefore, to run them, you cannot use usual Lua functions like `loadfile()` or `dofile()`. Use sol.main.load_file() and sol.main.do_file() instead to let Solarus determine how to locate and open the file.
Note however that `require()` can be used normally because it is a higher-level function. Indeed, a specific loader is automatically set by the engine so that `require()` looks in the archive if necessary and in the quest write directory. `require()` is the recommended way to load code from another file, because unlike sol.main.load_file() and sol.main.do_file(), it does not parses the file again every time you call it.

2.1.1.4 sol.main.do_file(script_name)

Loads and runs a Lua file into the current context.

This function is a replacement to the usual Lua function `dofile()`. The difference is that it looks for a file in the quest tree (which may be a directory or an archive) and also in the quest write directory. The quest write directory is tried first. The file must exist.

Use sol.main.load_file() explicitly if you need to check the existence of the file or to use parameters and return values.

- `script_name` (string): Name of the Lua file to load (with or without extension), relative to the data directory or the write directory of your quest.

Remarks

This function is equivalent to `sol.main.load_file(script_name)()`.
`require()` is the recommended way to load code from another file, because unlike sol.main.load_file() and sol.main.do_file(), it does not parses the file again every time you call it.

2.1.1.5 sol.main.reset()

Resets the whole program after the current cycle. Lua will be entirely shut down and then initialized again.

2.1.1.6 sol.main.exit()

Exits the program after the current cycle.

2.1.1.7 sol.main.get_elapsed_time()

Returns the simulated time elapsed since Solarus was started.

This corresponds to real time, unless the system is too slow to play at normal speed.

- Return value (number): The number of simulated milliseconds elapsed since the beginning of the program.

Remarks

This time is not reset when you call sol.main.reset().

2.1.1.8 sol.main.get_quest_write_dir()

Returns the subdirectory where files specific to the quest are saved, like savegames and settings. The quest write directory is specified in your quest.dat file and may be changed dynamically with sol.main.set_quest_write_dir().

- Return value (string): The quest write directory, relative to the Solarus write directory, or `nil` if it was not set.

2.1.1.9 sol.main.set_quest_write_dir(quest_write_dir)

Changes the subdirectory where files specific to the quest are saved, like savegames and settings. Note that the quest write directory can already be specified in your quest.dat file. You usually don't have to call this function, unless you need to change it dynamically for some reason.

- `quest_write_dir` (string): The quest write directory, relative to the Solarus write directory, or `nil` to unset it.

2.1.1.10 sol.main.load_settings([file_name])

Loads and applies the built-in settings from a file previously saved with sol.main.save_settings(). Settings from the file include user preferences such as the language, the video mode and the audio volume.

Note that all these settings can already be modified individually with the Lua API, so you can either use this function or implement something more fitted to your needs.

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise this function generates a Lua error.

- `file_name` (string, optional): Settings file to read, relative to the quest write directory. The default file name is `settings.dat`.
- Return value (boolean): `true` if settings were successfully loaded and applied.

Remarks

When you launch a quest from the Solarus GUI, user preferences that are currently set in the menus of the GUI are automatically written to `settings.dat` before the quest starts and are automatically applied as soon as the quest starts.

2.1.1.11 sol.main.save_settings([file_name])

Saves the current built-in settings into a file. This file can be reloaded later with sol.main.load_settings() to restore the saved settings. Settings saved include user preferences such as the current language, the current video mode and the current audio volume.

Note that all these settings can already be modified individually with the Lua API, so you can either use this function or implement something more fitted to your needs.

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise this function generates a Lua error.

- `file_name` (string, optional): Settings file to read, relative to the quest write directory. The default file name is `settings.dat`.
- Return value (boolean): `true` if settings were successfully saved.

Remarks

When you launch a quest from the Solarus GUI, user preferences that are currently set in the menus of the GUI are automatically written to `settings.dat` before the quest starts and are automatically applied as soon as the quest starts.

2.1.1.12 sol.main.get_distance(x1, y1, x2, y2)

Utility function that computes the distance in pixels between two points.

- `x1` (number): X coordinate of the first point.
- `y1` (number): Y coordinate of the first point.
- `x2` (number): X coordinate of the second point.
- `y2` (number): Y coordinate of the second point.
- Return value (number): The distance in pixels.

2.1.1.13 sol.main.get_angle(x1, y1, x2, y2)

Utility function that computes the angle in radians between the X axis and the specified vector.

- `x1` (number): X coordinate of the first point.
- `y1` (number): Y coordinate of the first point.
- `x2` (number): X coordinate of the second point.
- `y2` (number): Y coordinate of the second point.
- Return value (number): The angle in radians between the x axis and this vector.

2.1.1.14 sol.main.get_type(value)

Returns the type name of a value.

This function is similar to the standard Lua function type(), except that for userdata known by Solarus, it returns the exact Solarus type name.

- `value` (any type): Any value or `nil`.
- Return value (string): The name of the type. Can of one of standard Lua type names: `"nil"` (a string, not the value nil), `"number"`, `"string"`, `"boolean"`, `"table"`, `"function"`, `"thread"`. If it is a userdata unknown to Solarus, returns `"userdata"`. If it is a Solarus userdata, returns one of: `"game"`, `"map"`, `"item"`, `"surface"`, `"text_surface"`, `"sprite"`, `"timer"`, `"movement"`, `"straight_movement"`, `"target_movement"`, `"random_movement"`, `"path_movement"`, `"random_path_movement"`, `"path_finding_movement"`, `"circle_movement"`, `"jump_movement"`, `"pixel_movement"`, `"hero"`, `"dynamic_tile"`, `"teletransporter"`, `"destination"`, `"pickable"`, `"destructible"`, `"carried_object"`, `"chest"`, `"shop_treasure"`, `"enemy"`, `"npc"`, `"block"`, `"jumper"`, `"switch"`, `"sensor"`, `"separator"`, `"wall"`, `"crystal"`, `"crystal_block"`, `"stream"`, `"door"`, `"stairs"`, `"bomb"`, `"explosion"`, `"fire"`, `"arrow"`, `"hookshot"`, `"boomerang"`, `"camera"`, `"custom_entity"`.

2.1.1.15 sol.main.get_metatable(type_name)

Returns the metatable of a Solarus userdata type.

This function is very powerful and should be used with care.

All userdata objects of a type share the same metatable. So there is a metatable for maps, a metatable for games, a metatables for enemies, etc.

The metatable of a type stores what is common to all instances of this type. For example, the metatable of the `"enemy"` type has a field `"get_life"` that is the Solarus function enemy:get_life() shared by all enemies.

Note that you can already get the metatable of any object with the standard Lua function `getmetatable(object)`. This function does the same thing, except that you don't have to provide an existing object: you just provide a type name. This allows you do manipulate the metatable of a type before objects of this type start to exist, typically to set up things before a game is started.

You can use the metatable to add a function to all instances of a type. Thus, you can extend the Solarus API with your own functions. This also work for events (functions that the engine automatically calls when they exist). For example, you can easily provide a function `add_overlay()` to all your maps by defining it only once in the map metatable:

```
-- Somewhere in your main script, at initialization time:

local map_metatable = sol.main.get_metatable("map")
function map_metatable:add_overlay(image_file_name)
  -- Here, self is the map.
  self.overlay = sol.surface.create(image_file_name)
end

function map_metatable:on_draw(dst_surface)
  if self.overlay ~= nil then
    self.overlay:draw(dst_surface)
  end
end

-- Now, all your maps will have a function map:add_overlay() and an event
-- map:on_draw() that allows to draw an additional image above the map!
```

When you define a field in a metatable, everything acts like if you defined it in all existing and future instances of the type. However, an individual instance is still able to define a field with the same name. In this case, the instance has the priority: the field from the metatable is overridden.

Similarly, you can even remove (by assigning `nil`) or modify (by assigning a new value) any function of the Solarus API. We don't recommend to do that because introducing differences with the official API would be surprising for people who contribute to your game.

- `type_name` (string): Name of a Solarus userdata Lua type. Can be one of: `"game"`, `"map"`, `"item"`, `"surface"`, `"text_surface"`, `"sprite"`, `"timer"`, `"movement"`, `"straight_movement"`, `"target_movement"`, `"random_movement"`, `"path_movement"`, `"random_path_movement"`, `"path_finding_movement"`, `"circle_movement"`, `"jump_movement"`, `"pixel_movement"`, `"hero"`, `"dynamic_tile"`, `"teletransporter"`, `"destination"`, `"pickable"`, `"destructible"`, `"carried_object"`, `"chest"`, `"shop_treasure"`, `"enemy"`, `"npc"`, `"block"`, `"jumper"`, `"switch"`, `"sensor"`, `"separator"`, `"wall"`, `"crystal"`, `"crystal_block"`, `"stream"`, `"door"`, `"stairs"`, `"bomb"`, `"explosion"`, `"fire"`, `"arrow"`, `"hookshot"`, `"boomerang"`, `"camera"`, `"custom_entity"`.
- Return value (table): The metatable of this type, or `nil` if there is no such Solarus type.

Remarks

If you are a Lua expert, you know that metatables are a very powerful mechanism. They are where the magic happens. Solarus uses metatables to do a lot of things, like allowing you to access userdata as tables. Therefore, you should never touch any metamethod (fields whose name starts with two underscores) in the metatable of a userdata type, in particular `__index`, `__newindex` and `__gc`. This may result in unspecified behavior, but most likely crashes.

2.1.1.16 sol.main.get_os()

Returns the name of the running OS. Possible values are : `"Windows"`, `"Mac OS X"`, `"Linux"`, `"iOS"`, `"Android"`. If the correct OS name is not available, returns a string beginning with the text `"Unknown"`.

- Return value (string): The name of the running OS.

2.1.2 Events of sol.main

Events are callback methods automatically called by the engine if you define them.

2.1.2.1 sol.main:on_started()

Called at the beginning of the program.

This function is called when Lua starts. In other words, the function is called when the program begins or after it was reset. In this function, you will typically start an initial menu like a title screen or a language selection screen. If you prefer, you can also start immediately a game.

2.1.2.2 sol.main:on_finished()

Called at the end of the program.

This function is called when Lua is about to be shut down, i.e., just before the program stops or is reset.

2.1.2.3 sol.main:on_update()

Called at each cycle of the program's main loop.

Remarks

This event is very powerful (because it is called at global level of the program) but it may be costly if you do heavy operations. Keep in mind that it is called at each cycle. You can usually use other callbacks instead to get notified of when an event happens. You can also use timers if you want to regularly check something.

2.1.2.4 sol.main:on_draw(dst_surface)

Called when the program's screen is redrawn.

At this point, the engine has already drawn the game (if a game is running) and has not drawn yet the menus of `sol.main` (if you have created menus on `sol.main`). Use this event if you want to draw some additional content.

- `dst_surface` (surface): The surface where the screen is drawn.

2.1.2.5 sol.main:on_key_pressed(key, modifiers)

Called when the user presses a keyboard key.

- `key` (string): Name of the raw key that was pressed.
- `modifiers` (table): A table whose keys indicate what modifiers were down during the event. Possible table keys are `"shift"`, `"control"` and `"alt"`. Table values don't matter.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like the game). However, if the key corresponds to a character, another event sol.main:on_character_pressed() will also be called.

Remarks

This event indicates the raw key pressed. If you want to know the corresponding character instead (if any), see sol.main:on_character_pressed().

2.1.2.6 sol.main:on_key_released(key, modifiers)

Called when the user releases a keyboard key.

- `key` (string): Name of the raw key that was released.
- `modifiers` (table): A table whose keys indicate what modifiers were down during the event. Possible table keys are `"shift"`, `"control"` and `"alt"`. Table values don't matter.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like the game).

2.1.2.7 sol.main:on_character_pressed(character)

Called when the user enters text.

- `character` (string): A utf-8 string representing the character that was entered.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like the game).

Remarks

When a character key is pressed, two events are called: sol.main:on_key_pressed() (indicating the raw key) and sol.main:on_character_pressed() (indicating the utf-8 character). If you need to input text from the user, sol.main:on_character_pressed() is what you want because it considers the keyboard's layout and gives you international utf-8 strings.

2.1.2.8 sol.main:on_joypad_button_pressed(button)

Called when the user presses a joypad button.

- `button` (number): Index of the button that was pressed.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like the game).

2.1.2.9 sol.main:on_joypad_button_released(button)

Called when the user releases a joypad button.

- `button` (number): Index of the button that was released.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like the game).

2.1.2.10 sol.main:on_joypad_axis_moved(axis, state)

Called when the user moves a joypad axis.

- `axis` (number): Index of the axis that was moved. Usually, `0` is an horizontal axis and `1` is a vertical axis.
- `state` (number): The new state of the axis that was moved. `-1` means left or up, `0` means centered and `1` means right or down.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like the game).

2.1.2.11 sol.main:on_joypad_hat_moved(hat, direction8)

Called when the user moves a joypad hat.

- `hat` (number): Index of the hat that was moved.
- `direction8` (number): The new direction of the hat. `-1` means that the hat is centered. `0` to `7` indicates that the hat is in one of the eight main directions.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like the game).

2.1.2.12 sol.main:on_mouse_pressed(button, x, y)

Called when the user presses a mouse button.

- `button` (string): Name of the mouse button that was pressed. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- `x` (integer): The x position of the mouse in quest size coordinates.
- `y` (integer): The y position of the mouse in quest size coordinates.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.1.2.13 sol.main:on_mouse_released(button, x, y)

Called when the user releases a mouse button.

- `button` (string): Name of the mouse button that was released. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- `x` (integer): The x position of the mouse in quest size coordinates.
- `y` (integer): The y position of the mouse in quest size coordinates.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.2 Audio

You can play musics and sound effects through `sol.audio`.

2.2.1 Functions of sol.audio

2.2.1.1 sol.audio.play_sound(sound_id)

Plays a sound effect.

Generates a Lua error if the sound does not exist.

Several sounds can be played in parallel. In the current version, a sound cannot be interrupted after you start playing it.

Unlike musics, sounds files are entirely read before being played. A file access is made only the first time you play each sound. You can use sol.audio.preload_sounds() if you want to also avoid this initial file access.

- `sound_id` (string): Name of the sound file to play, relative to the `sounds` directory and without extension. Currently, `.ogg` is the only extension supported.

2.2.1.2 sol.audio.preload_sounds()

Loads all sounds effects into memory for faster future access.

If you don't call this function, you can still play sound effects, but the first access to each sound effect will require a file access that might be perceptible on slow machines. It is recommended to call this function at the beginning of the program (typically from sol.main:on_started()).

The list of sound files to load is read from the project database file.

This function does nothing if you already called it before.

2.2.1.3 sol.audio.play_music(music_id, [action])

Plays a music.

If the music does not exist, a Lua error is generated.

Only one music can be played at a time. If the same music was already playing, it continues normally and does not restart. If a different music was already playing, it is stopped and replaced by the new one.

When the music reaches the end, the `action` parameter indicates what to do next. The default behavior is to loop from the beginning.

However, some music files already have their own loop internal loop information. Such musics are able to loop to a specific point rather than to the beginning. Since they already loop forever internally, they don't have an end and the `action` parameter has no effect on them. See Music loop settings to know how Solarus supports internal loop information for each format.

- `music_id` (string): Name of the music file to play, relative to the `musics` directory and without extension. The following extensions will be tried in this order: `.ogg`, `.it` and `.spc`. `nil` stops playing any music (the second parameter has no effect in this case). If you set the music name to the same music that is already playing, or to the special value `"same"`, then this function does nothing: the music keeps playing (it does not restart) and the second parameter is ignored.
- `action` (function or boolean, optional): What to do when the music finishes (reaches its end). A boolean value indicates whether or not the music should loop. The default is `true`. A function value indicates a custom action (and implies no loop). It will be called when the music finishes. This allows you to perform an action of your choice, like playing another music.

2.2.1.4 sol.audio.get_music()

Returns the name of the music currently playing.

- Return value (string): Name of the music file currently playing, relative to the `musics` directory and without extension. Returns `nil` if no music is playing.

2.2.1.5 sol.audio.stop_music()

Stops playing music.

This function has no effect if no music was playing.

Remarks

This is equivalent to `sol.audio.play_music("none")`.

2.2.1.6 sol.audio.get_sound_volume()

Returns the current volume of sound effects.

This volume applies to all sounds played by sol.audio.play_sound() and by the engine.

- Return value (number): The current volume of sound effects, as an integer between `0` (mute) and `100` (full volume).

2.2.1.7 sol.audio.set_sound_volume(volume)

Sets the volume of sound effects.

This volume applies to all sounds played by sol.audio.play_sound() and by the engine.

- `volume` (number): The new volume of sound effects, as an integer between `0` (mute) and `100` (full volume).

Remarks

When the quest is run from the Solarus GUI, the user can also change the volume from the menus of the GUI.

2.2.1.8 sol.audio.get_music_volume()

Returns the current volume of musics.

This volume applies to all musics played by sol.audio.play_music(music_id, [action]). "sol.audio.play_music()"

- Return value (number): The current volume of musics, as an integer between `0` (no sound effects) and `100` (full volume).

Remarks

When the quest is run from the Solarus GUI, the user can also change the volume from the menus of the GUI.

2.2.1.9 sol.audio.set_music_volume(volume)

Sets the volume of musics.

This volume applies to all musics played by sol.audio.play_music(music_id, [action]). "sol.audio.play_music()"

- `volume` (number): The new volume of musics, as an integer between `0` (no music) and `100` (full volume).

2.2.1.10 sol.audio.get_music_format()

Returns the format of the music currently playing.

- Return value (string): Format of the music: `"ogg"`, `"it"` or `"spc"`. Returns `nil` if no music is playing.

2.2.1.11 sol.audio.get_music_num_channels()

Returns the number of channels of the current .it music.

This function is only supported for .it musics.

- Return value (number): Number of channels of the music. Returns `nil` if the current music format is not .it.

2.2.1.12 sol.audio.get_music_channel_volume(channel)

Returns the volume of notes of a channel for the current .it music.

This function is only supported for .it musics.

- `channel` (number): Index of a channel (the first one is zero).
- Return value (number): Volume of the channel. Returns `nil` if the current music format is not .it.

Remarks

The channel should have the same volume for all its notes. Otherwise, calling this function does not make much sense.

2.2.1.13 sol.audio.set_music_channel_volume(channel, volume)

Sets the volume of all notes of a channel for the current .it music.

This function has no effect for musics other than .it.

- `channel` (number): Index of a channel (the first one is zero).
- `volume` (number): The volume to set.

2.2.1.14 sol.audio.get_tempo()

Returns the tempo of the current .it music.

This function is only supported for .it musics.

- Return value (number): Tempo of the music. Returns `nil` if the current music format is not .it.

2.2.1.15 sol.audio.set_tempo(tempo)

Sets the tempo of the current .it music.

This function is only supported for .it musics.

- `tempo` (number): Tempo to set.

2.3 Video

`sol.video` allows you to manage the window and to change the video mode.

The area where the game takes place has a fixed size called the quest size. The quest size is in a range specified in quest.dat. This quest size defines how much content the player can see on the map. It is usually 320x240, but some systems may prefer other sizes, like 400x240 on Android. You should set a range of quest sizes in the quest properties files: this is a good idea for portability. However, it requires more work from the quest designer: in particular, you have to implement menus and a HUD that can adapt to any size in this range. Note however that the quest size never changes after the program is started.

At runtime, the video mode can be changed. The video mode determines whether a pixel scaling algorithm is applied to the image, and which one.

Any video mode can be played in windowed mode or in fullscreen. When the user plays in fullscreen, the quest image is automatically stretched and keeps the same pixel ratio, possibly by adding black borders.

The video mode can be saved and loaded with the global settings, as well as the language and the volume, independently of any savegame (see sol.main.load_settings()).

Video modes are identified by strings in the Lua API. The following modes are implemented:

- `"normal"` (default): The quest screen is unchanged. In windowed mode, the image is stretched by a factor of 2.
- `"scale2x"`: The quest screen is scaled by a factor of 2 with the `Scale2X` algorithm.
- `"hq2x"`: The quest screen is scaled by a factor of 2 with the `hq2x` algorithm.
- `"hq3x"`: The quest screen is scaled by a factor of 3 with the `hq3x` algorithm.
- `"hq4x"`: The quest screen is scaled by a factor of 4 with the `hq4x` algorithm.

In windowed mode, the user and the scripts can resize the window. The image is then automatically stretched and keeps the same pixel ratio, possibly by adding black borders.

2.3.1 Functions of sol.video

2.3.1.1 sol.video.get_window_title()

Returns the text of the title bar of the window.

- Return value (string): The window title.

2.3.1.2 sol.video.set_window_title(window_title)

Sets the text of the title bar of the window.

By default, the window title is set to the title of your quest followed by its version. Both these properties are indicated in the quest.dat file.

- `window_title` (string): The window title to set.

2.3.1.3 sol.video.get_mode()

Returns the current video mode.

- Return value (string): Name of the video mode (see the list above).

2.3.1.4 sol.video.set_mode(video_mode)

Sets the current video mode.

- `video_mode` (string): Name of the video mode (see the list above).

Remarks

When the quest is run from the Solarus GUI, the user can also change the video mode from the menus of the GUI.

2.3.1.5 sol.video.switch_mode()

Sets the next video mode in the list of video modes supported.

You can use this function if you want to change the video mode without specifying which one to use. The fullscreen flag is never changed by this function.

2.3.1.6 sol.video.is_mode_supported(video_mode)

Returns whether a video mode is supported.

- `video_mode` (string): Name of a video mode (see the list above).

2.3.1.7 sol.video.get_modes()

Returns an array of all video modes supported.

- `Return` value (table): An array of names of all video modes supported.

2.3.1.8 sol.video.is_fullscreen()

Returns whether the current video mode is fullscreen.

- Return value (boolean): `true` if the video mode is fullscreen.

2.3.1.9 sol.video.set_fullscreen([fullscreen])

Turns on or turns off fullscreen, keeping an equivalent video mode.

- `fullscreen` (boolean, optional): `true` to set fullscreen (no value means `true`).

Remarks

When the quest is run from the Solarus GUI, the user can also switch fullscreen from the menus of the GUI.

2.3.1.10 sol.video.is_cursor_visible()

Returns whether the mouse cursor is currently visible.

- Return value (boolean): `true` if the mouse cursor is currently visible.

2.3.1.11 sol.video.set_cursor_visible([cursor_visible])

Shows or hides the mouse cursor, keeping an equivalent video mode.

- `visible_cursor` (boolean, optional): `true` to show mouse cursor (no value means `true`).

2.3.1.12 sol.video.get_quest_size()

Returns the size of the quest screen.

This quest size is fixed at runtime. It is always in the range of allowed quest sizes specified in quest.dat.

The quest size is independent from the video mode: video modes are just various methods to draw the quest on the window.

- Return value 1 (number): Width of the quest screen in pixels.
- Return value 2 (number): Height of the quest screen in pixels.

2.3.1.13 sol.video.get_window_size()

Returns the size of the window.

The quest image has a fixed size determined when the program starts. This quest image is then stretched on the window, and the size of the window size can be changed by the user or by sol.video.set_window_size(). Black borders are added if necessary to keep the correct pixel ratio.

When the window is in fullscreen, this function returns the size to be used when returning to windowed mode.

By default, the size of the window depends on the video mode. If no scaling algorithm is used (that is, in `"normal"` mode), the window size is initially twice the quest size to avoid a too small window. If a scaling algorithm is used, the scaling factor is naturally applied to the window size. For example, with hq2x, if your quest size is 320x240, the window will initially have a size of 640x480.

- Return value 1 (number): Width of the window in pixels.
- Return value 2 (number): Height of the window in pixels.

2.3.1.14 sol.video.set_window_size(width, height)

Sets the size of the window.

See sol.video.get_window_size() for a detailed description of the window size.

When the window is in fullscreen, this function still works: the changes will be visible when returning to windowed mode.

Note that when the video mode is changed, the window size is reset to its default value for the new video mode.

- Return value 1 (number): Width of the window in pixels.
- Return value 2 (number): Height of the window in pixels.

2.3.1.15 sol.video.reset_window_size()

Restores the size of the window to its default value.

The default size of the window depends on the video mode: see sol.video.get_window_size() for a detailed explanation.

When the window is in fullscreen, this function still works: the changes will be visible when returning to windowed mode.

2.4 Inputs

You can get information about the low-level keyboard and joypad inputs through `sol.input`.

But remember that when a low-level keyboard or joypad input event occurs, all useful objects (sol.main, the game, map and menus) are already notified. For example, when the user presses a keyboard key, the engine automatically calls sol.main:on_key_pressed().

Also note that during the game, there exists the higher-level notion of game commands to ease your life.

2.4.1 Functions of sol.input

2.4.1.1 sol.input.is_joypad_enabled()

Returns whether joypad support is enabled.

This may be true even without any joypad plugged.

- Return value (boolean): `true` if joypad support is enabled.

2.4.1.2 sol.input.set_joypad_enabled([joypad_enabled])

Enables or disables joypad support.

Joypad support may be enabled even without any joypad plugged.

- `joypad_enabled` `true` to enable joypad support, false to disable it. No value means `true`.

2.4.1.3 sol.input.is_key_pressed(key)

Returns whether a keyboard key is currently down.

- `key` (string): The name of a keyboard key.
- Return value (boolean): `true` if this keyboard key is currently down.

2.4.1.4 sol.input.get_modifiers()

Returns the keyboard key modifiers currently active.

- Return value (table): A table whose keys indicate what modifiers are currently down. Possible table keys are `"shift"`, `"control"`, `"alt"` and `"caps lock"`. Table values don't matter.

2.4.1.5 sol.input.is_joypad_button_pressed(button)

Returns whether a joypad button is currently down.

- `button` (number): Index of a button of the joypad.
- Return value (boolean): `true` if this joypad button is currently down.

2.4.1.6 sol.input.get_joypad_axis_state(axis)

Returns the current state of an axis of the joypad.

- `axis` (number): Index of a joypad axis. The first one is `0`.
- Return value (number): The state of that axis. `-1` means left or up, `0` means centered and `1` means right or down.

2.4.1.7 sol.input.get_joypad_hat_direction(hat)

Returns the current direction of a hat of the joypad.

- `hat` (number): Index of a joypad hat. The first one is `0`.
- Return value (number): The direction of that hat. `-1` means that the hat is centered. `0` to `7` indicates that the hat is in one of the eight main directions.

2.4.1.8 sol.input.get_mouse_position()

Returns the current position of the mouse cursor.

- Return value 1 (integer): The `x` position of the mouse in quest size coordinates. Return `nil` if the mouse position is outside the quest screen.
- Return value 2 (integer): The `y` position of the mouse in quest size coordinates. Unused if the mouse position is outside the quest screen.

2.4.1.9 sol.input.is_moutton_button_pressed(button)

Returns whether a mouse button is currently down.

- `button` (string): The name of a mouse button. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- Return value (boolean): `true` if mouse button is down.

2.5 Files

This module provides functions to manually read and write files from the quest data directory and from quest write directory.

2.5.1 Overview

Data files of the quest are located either in a `data/` directory or in a `data.solarus` or `data.solarus.zip` archive. Files that you write are in the write directory of the quest.

This module encapsulates the transparent search of these files in both locations. You can access them independently of their actual place.

However, note that all built-in file accesses in Solarus (loading maps, loading sprites, reading and writing savegames, etc.) already implement this transparent search. You will only need this module if you implement your custom data files or if you have some custom files to save.

In all functions of `sol.file`, the notation for the directory separator is always `"/"` (slash), no matter the underlying platform.

You can read and write files in the quest write directory, and you can only read files in the data directory or archive. If a file to read exists in both locations, the quest write directory is given priority.

Examples of use:

- You can implement your own savegame system if the built-in one does not fit your needs.
- You can save a file that contain some user settings, and have a default version of that file in the data directory. Since the quest write directory has the priority, the saved version will be used if any.
- You can save dynamic content like maps automatically generated. Since data files are also read from the quest write directory, map files can be loaded from there.
- You can make an in-game dialog editor for translators.

2.5.2 Functions of sol.file

2.5.2.1 sol.file.open(file_name, [mode])

Same as `io.open()`, but relative to the quest write directory or to the data directory.

If a valid quest write directory is set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), that directory is tried first. Then, the `data` directory of your quest is tried if the mode is `"r"` (read mode).

This function just calls `io.open()` with the actual path and the mode as parameters, and returns its results.

- `file_name` (string): Name of the file to open, relative to the quest write directory or to the data directory.
- `mode` (string, optional): Opening mode (see the Lua documentation of `io.open()`).
- Return value (file or nil+string): The file object created, or `nil` plus an error message in case of failure. The file value is the standard Lua file object as returned by `io.open()`, and you can then use all Lua file functions (file:read(), file:write(), file:lines(), file:close(), etc.).

2.5.2.2 sol.file.exists(file_name)

Returns whether the specified file exists in the quest write directory of in the data directory.

If a valid quest write directory is set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), that directory is tried first. Then, the `data` directory of your quest is tried.

- `file_name` (string): Name of the file to test, relative to the quest write directory or to the data directory.
- Return value (boolean): `true` if there exists a file with this name.

2.5.2.3 sol.file.remove(file_name)

Deletes a file or a directory from the quest write directory.

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise this function generates a Lua error.

- `file_name` (string): Name of the file to delete, relative to the quest write directory. If it is a directory, it must be empty before you delete it.
- Return value (boolean and string): `true` in case of success, `nil` plus an error message in case of failure.

2.5.2.4 sol.file.rename(old_file_name, new_file_name)

Renames a file or a directory in the quest write directory.

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise this function generates a Lua error.

- `old_file_name` (string): Name of the file to rename, relative to the quest write directory.
- `new_file_name` (string): New name to set, relative to the quest write directory.
- Return value (boolean and string): `true` in case of success, `nil` plus an error message in case of failure.

2.5.2.5 sol.file.mkdir(dir_name)

Creates a directory in the quest write directory.

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise this function generates a Lua error.

- `dir_name` (string): Name of the directory to delete, relative to the quest write directory.
- Return value (boolean or nil+string): `true` in case of success, `nil` plus an error message in case of failure.

2.6 Menus

To display various information such as a title screen, a dialog box, a HUD (head-up display) or a pause screen, you can use one or several menus.

A menu is an arbitrary Lua table. A menu belongs to a context that may be the current map, the current game, the sol.main table or even another menu. This context is the lifetime of your menu. As long as your menu is active, the engine will call events that are defined in your table, i.e. callback methods like menu:on_started() menu:on_draw(), menu:on_key_pressed(), etc., to notify your menu of something (the player pressed a key, your menu needs to be redrawn, etc.).

If you prefer, you can also make menus without this API. Indeed, the map, game and sol.main APIs already have everything you need, so you can do what you want from there manually. But menus built with the API described on this page are just handy, because they automatically receive events whenever they need to be notified.

2.6.1 Functions of sol.menu

2.6.1.1 sol.menu.start(context, menu, [on_top])

Starts a menu in a context.

The Solarus engine will then call the appropriate events on your menu until it is stopped.

- `context` (map, game or table): The context your menu will belong to. Similarly to the case of timers, the context determines the lifetime of your menu. The context must be one of the following four objects:
  - If you make a map menu, your menu will be drawn above the map surface. It will be stopped when the player goes to another map. This may be useful to show head-up information local to a precise map.
    Example: a counter or a mini-game that only exists on a specific map.
  - If you make a game menu, your menu will be global to all maps. As long as the game is running, it will persist accross map changes.
    Example: the player's life counter.
  - If you make a main menu, your menu will be global to the whole program. It can exist outside a game (and it even persists during the game if you don't stop it).
    Example: the title screen.
  - If you set the context to another menu, then its lifetime will be limited to this other menu. This allows to make nested menus. Example: a popup that shows some information above another menu.
  - `menu` (table): The menu to activate. It can be any table. The only thing that makes it special is the presence of callback functions (events) as described in section Events of a menu.
- `on_top` (boolean, optional): Whether this menu should be drawn on top of other existing menus of the same context or behind them. If `true`, the on_draw() event of your menu will be the last to be called if there are several menus in the context. If `false`, it will be the first one. No value means `true`.

2.6.1.2 sol.menu.stop(menu)

Stops a menu previously activated with sol.menu.start().

After this call, the Solarus engine will no longer call events on your menu. But you can restart it later if you want.

Nothing happens is the menu was already stopped or never started.

- `menu` (table): The menu to stop.

2.6.1.3 sol.menu.stop_all(context)

Stops all menus that are currently running in a context.

- `context` (map, game or sol.main): The context where you want to stop menus.

This function is not often needed since menus are already automatically stopped when their context is closed.

2.6.1.4 sol.menu.is_started(menu)

Returns whether a table is a currently active menu.

- `menu` (table): The menu to test.
- Return value (boolean): `true` if this is an active menu.

2.6.2 Events of a menu

Events are callback methods automatically called by the engine if you define them.

2.6.2.1 menu:on_started()

Called when your menu is started.

This event is triggered when you call sol.menu.start().

2.6.2.2 menu:on_finished()

Called when your menu is stopped.

This event is triggered when you call sol.menu.stop() or when the context of your menu finishes.

2.6.2.3 menu:on_update()

Called at each cycle of the main loop while your menu is active.

Menus of are updated in the following order:

1. Map menus (only during a game).
2. Game menus (only during a game).
3. Main menus (the more general ones).

When several menus exist in the same context, they are updated from the back one to the front one. You can control this order thanks to the `on_top` parameter of menu:start() when you start a menu.

Remarks

As this function is called at each cycle, it is recommended to use other solutions when possible, like timers and other events.

2.6.2.4 menu:on_draw(dst_surface)

Called when your menu has to be redrawn.

Use this event to draw your menu.

- `dst_surface` (surface): The surface where you should draw your menu.

Menus of are drawn in the following order:

1. Map menus (only during a game).
2. Game menus (only during a game).
3. Main menus (the more general ones).

When several menus exist in the same context, they are updated from the back one to the front one. You can control this order thanks to the `on_top` parameter of menu:start() when you start a menu.

2.6.2.5 menu:on_key_pressed(key, modifiers)

Called when the user presses a keyboard key while your menu is active.

- `key` (string): Name of the raw key that was pressed.
- `modifiers` (table): A table whose keys indicate what modifiers were down during the event. Possible table keys are `"shift"`, `"control"` and `"alt"`. Table values don't matter.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus or the built-in game commands).

For all keyboard and joypad events, most recent menus are notified first. For example, if some dialog box is shown during the pause menu and appears above that pause menu, it will naturally receive keyboard and joypad events first.

When a menu handles the event, it should return `true` to make the event stop being propagated. Menus (and other objects) below it won't be not notified. On the contrary, if no script has handled the event, then the engine can handle it with a built-in behavior.

Remarks

This event indicates the raw key pressed. If you want the corresponding character instead (if any), see menu:on_character_pressed().

2.6.2.6 menu:on_key_released(key, modifiers)

Called when the user releases a keyboard key while your menu is active. Menus on top are notified first.

- `key` (string): Name of the raw key that was released.
- `modifiers` (table): A table whose keys indicate what modifiers were down during the event. Possible table keys are `"shift"`, `"control"` and `"alt"`. Table values don't matter.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus or the built-in game commands).

2.6.2.7 menu:on_character_pressed(character)

Called when the user enters text while your menu is active. Menus on top are notified first.

- `character` (string): A utf-8 string representing the character that was pressed.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus).

Remarks

When a character key is pressed, two events are called: menu:on_key_pressed() (indicating the raw key) and menu:on_character_pressed() (indicating the utf-8 character). If your menu needs to input text from the user, menu:on_character_pressed() is what you want because it considers the keyboard's layout and gives you international utf-8 strings.

2.6.2.8 menu:on_joypad_button_pressed(button)

Called when the user presses a joypad button while your menu is active. Menus on top are notified first.

- `button` (number): Index of the button that was pressed.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus or the built-in game commands).

2.6.2.9 menu:on_joypad_button_released(button)

Called when the user releases a joypad button while your menu is active. Menus on top are notified first.

- `button` (number): Index of the button that was released.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus or the built-in game commands).

2.6.2.10 menu:on_joypad_axis_moved(axis, state)

Called when the user moves a joypad axis while your menu is active. Menus on top are notified first.

- `axis` (number): Index of the axis that was moved. Usually, `0` is an horizontal axis and `1` is a vertical axis.
- `state` (number): The new state of the axis that was moved. `-1` means left or up, `0` means centered and `1` means right or down.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus or the built-in game commands).

2.6.2.11 menu:on_joypad_hat_moved(hat, direction8)

Called when the user moves a joypad hat while your menu is active. Menus on top are notified first.

- `hat` (number): Index of the hat that was moved.
- `direction8` (number): The new direction of the hat. `-1` means that the hat is centered. `0` to `7` indicates that the hat is in one of the eight main directions.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus or the built-in game commands).

2.6.2.12 menu:on_command_pressed(command)

Called during a game when the player presses a game command (a keyboard key or a joypad action mapped to a built-in game behavior). You can use this event to override the normal built-in behavior of the game command. Menus on top are notified first.

- `command` (string): Name of the built-in game command that was pressed (see the game API for the list of existing game commands).
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (you are overriding the built-in behavior of pressing this game command).

Remarks

As the notion of game commands only exist during a game, this event is only called for game menus and map menus.
This event is not triggered if you already handled the underlying low-level keyboard or joypad event.

2.6.2.13 menu:on_command_released(command)

Called during a game when the player released a game command (a keyboard key or a joypad action mapped to a built-in game behavior). You can use this event to override the normal built-in behavior of the game command. Menus on top are notified first.

- `command` (string): Name of the built-in game command that was released (see the game API for the list of existing game commands).
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (you are overriding the built-in behavior of releasing this game command).

Remarks

As the notion of game commands only exist during a game, this event is only called for game menus and map menus.
This event is not triggered if you already handled the underlying low-level keyboard or joypad event.

2.6.2.14 menu:on_mouse_pressed(button, x, y)

Called when the user presses a mouse button while this menu is active. Menus on top are notified first.

- `button` (string): Name of the mouse button that was pressed. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- `x` (integer): The x position of the mouse in quest size coordinates.
- `y` (integer): The y position of the mouse in quest size coordinates.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus).

2.6.2.15 menu:on_mouse_released(button, x, y)

Called when the user releases a mouse button while this menu is active.

- `button` (string): Name of the mouse button that was released. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- `x` (integer): The x position of the mouse in quest size coordinates.
- `y` (integer): The y position of the mouse in quest size coordinates.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (like other menus).

2.7 Language functions

`sol.language` lets you get and set the current language and manage language-specific data.

Each language is identified by a code (like `"en"`, `"fr"`) and has a human-readable name (like `"English"`, `"Français"`). The language code corresponds to the name of the directory with files translated in this language (sprites, strings and dialogs).

The list of languages available in your quest is specified in the quest resource list project_db.dat.

2.7.1 Functions of sol.language

2.7.1.1 sol.language.get_language()

Returns the code of the current language.

- Return value (string): The code that identifies the current language, or `nil` if no language was set.

2.7.1.2 sol.language.set_language(code)

Changes the current language.

- `code` (string): The code that identifies the new language to set. It must be a valid code as defined in project_db.dat.

2.7.1.3 sol.language.get_language_name([code])

Returns the human-readable name of a language.

- `code` (string, optional): The code of a language (no value means the current language).
- Return value (string): the human-readable name of this language.

2.7.1.4 sol.language.get_languages()

Returns the list of available languages.

- Return value (table): An array of all language codes defined in project_db.dat. Languages are in the same order as in `project_db.dat`.

2.7.1.5 sol.language.get_string(key)

Returns a string translated in the current language.

Translated strings are defined in the file text/strings.dat of the language-specific directory (e.g. `languages/en/text/strings.dat`).

- `key` (string): Key of the string to get. The corresponding key-value pair must be defined in text/strings.dat.
- Return value (string): The value associated to this key in text/strings.dat, or `nil` if it does not exist.

2.7.1.6 sol.language.get_dialog(dialog_id)

Returns a dialog translated in the current language.

Translated dialogs are defined in the file text/dialogs.dat of the language-specific directory (e.g. `languages/en/text/dialogs.dat`).

- `dialog_id` (string): Id of the dialog to get.
- Return value (table): The corresponding dialog in the current language, or `nil` if it does not exist. The dialog is a table with at least the following two entries:
  - `dialog_id` (string): Id of the dialog.
  - `text` (string): Text of the dialog. The table also contains all custom entries defined in text/dialogs.dat for this dialog. These custom entries always have string keys and string values. Values that were defined as numbers in `"text/dialogs.dat"` are replaced in this table by their string representation, and values that were defined as booleans are replaced by the string `"1"` for `true` and `"0"` for `false`.

2.8 Timers

Timers allow you to call a function in the future with a specified delay.

Here is a first example of use:

```
-- Play sound "secret" in one second.
local function play_secret_sound()
  sol.audio.play_sound("secret")
end

sol.timer.start(1000, play_secret_sound)
```

Shorter version to do the same thing:

```
-- Equivalent code using an anonymous function.
sol.timer.start(1000, function()
  sol.audio.play_sound("secret")
end)
```

You can repeat a timer by returning `true` from your function:

```
-- Call a function every second.
sol.timer.start(1000, function()
  sol.audio.play_sound("danger")
  return true  -- To call the timer again (with the same delay).
end)
```

To make a timer that repeats itself 10 times, just return `false` after 10 calls:

```
-- Call a function ten times, with one second between each call.
local num_calls = 0
sol.timer.start(1000, function()
  sol.audio.play_sound("danger")
  num_calls = num_calls + 1
  return num_calls < 10
end)
```

It is possible to restrict the lifetime of a timer to a context, like the game, the map or an enemy:

```
-- Enemy that shoots a fireball every 5 seconds until it is killed.
sol.timer.start(your_enemy, 5000, function()
  sol.audio.play_sound("attack_fireball")
  map:create_enemy(...)  -- Code that creates the fireball.
  return true  -- Repeat the timer.
end)
```

Setting the context to an enemy ensures that when the enemy is killed, the timer is canceled. Otherwise, the callback function would still be called: in this example, you would hear the "`attack_fireball`" sound and the fireball would be created even if the enemy is killed in the meantime.

2.8.1 Functions of sol.timer

2.8.1.1 sol.timer.start([context], delay, callback)

Sets a function to be called after a delay.

If the delay is set to zero, the function is called immediately.

- `context` (map, game, item, map entity, menu or sol.main; optional): Determines the lifetime of the timer. The context is where the timer belongs.
  If the context gets closed before the timer is finished, then the timer is automatically canceled. More precisely, the following rules are applied.
  - If you set the context to a map, the timer is canceled when the player goes to another map.
    Example: a button that opens a door for a limited time.
  - If you set the context to a game or an item, the timer is canceled when the game is closed. (Items have the same lifetime as the game they belong to.) This is only possible when the game is running.
    Example: hot water that becomes cold after a few minutes, and that the player should bring to an NPC on another map while it's still hot.
  - If you set the context to a map entity, the timer is canceled when the entity is removed from the map. In the case of an enemy, the timer is also canceled when the enemy is hurt, immobilized or restarts. Also note that while the entity is suspended, the timer is temporarily paused. An entity may be suspended when the game is suspended, or when the entity is disabled.
    Example: a boss who shoots fireballs every 10 seconds. Most enemy scripts usually create timers.
  - If you set the context to a menu, the timer is canceled when the menu is closed.
    Example: in the title screen, show some animations after a few seconds without action from the user.
  - If you set the context to the sol.main table, the timer is canceled when Lua is closed. Thus, it will be a global timer. This kind of timer is not often needed.
    Example: dumping some global information periodically while the program is running.
  - If you don't specify a context, then a default context is set for you: the current map during a game, and sol.main if no game is running.
- `delay` (number): Delay before calling the function in milliseconds.
- `callback` (function): The function to be called when the timer finishes. If this callback function returns `true`, then the timer automatically repeats itself with the same delay. In this case, if the delay is shorter than the time of a cycle of the main loop, the callback may be executed several times in the same cycle in order to catch up.
- Return value (timer): The timer created. Most of the time, you don't need to store the returned timer. Indeed, there is no problem if it gets garbage-collected: the timer persists in the engine side until its completion or the end of its context. Usually, you will store the return value only if you need to stop the timer explicitly later or to call another method on it.

Remarks

When they are created, map timers, map entity timers and item timers are initially suspended if a dialog is active. After that, they get automatically suspended and unsuspended when the map is suspended or unsuspended. This default behavior is suited for most use cases, but if you want to change it, you can use timer:set_suspended() and timer:set_suspended_with_map().

2.8.1.2 sol.timer.stop_all(context)

Cancels all timers that are currently running in a context.

- `context` (map, game, item, map entity, menu or sol.main): The context where you want to stop timers.

This function is equivalent to calling timer:stop() on each timer of the context. It may allow you to avoid to store explicitly all your timers.

Remarks

Canceling timers by hand may be tedious and error-prone. In lots of cases, you can simply pass a context parameter to sol.timer.start() in order to restrict the lifetime of your timer to some other object.

2.8.2 Methods of the type timer

2.8.2.1 timer:stop()

Cancels this timer.

If the timer was already finished or canceled, nothing happens.

Remarks

Canceling timers by hand may be tedious and error-prone. In lots of cases, you can simply pass a context parameter to sol.timer.start() in order to restrict the lifetime of your timer to some other object.

2.8.2.2 timer:is_with_sound()

Returns whether a clock sound is played repeatedly during this timer.

- Return value (boolean): `true` if a clock sound is played with this timer.

2.8.2.3 timer:set_with_sound(with_sound)

Sets whether a clock sound is played repeatedly during this timer.

- `with_sound` (boolean, optional): `true` to play a clock sound repeatedly (no value means `true`).

2.8.2.4 timer:is_suspended()

Returns whether this timer is currently suspended.

- Return value (boolean): `true` if this timer is currently suspended.

2.8.2.5 timer:set_suspended([suspended])

Returns whether this timer is currently suspended.

- `suspended` (boolean, optional): `true` to suspend the timer, `false` to unsuspend it (no value means `true`).

2.8.2.6 timer:is_suspended_with_map()

Returns whether this timer gets automatically suspended when the map is suspended.

- Return value (boolean): `true` if this timer gets suspended when the map is suspended.

2.8.2.7 timer:set_suspended_with_map([suspended_with_map])

Sets whether this timer should automatically be suspended when the map gets suspended.

The map is suspended by the engine in a few cases, like when the game is paused, when there is a dialog or when the camera is being moved by a script. When this happens, all map entities stop moving and most sprites stop their animation. With this setting, you can choose whether your timer gets suspended too.

By default, map timers, and item timers are suspended with the map. Entity timers are more complex: an entity (and its timers) also get suspended when the entity gets disabled. You should not use this function on entity timers.

- `suspended_with_map` (boolean, optional): `true` to suspend the timer when the map is suspended, `false` to continue (no value means `true`).

Remarks

Even when this setting is set to `true`, you can override its behavior by calling timer:set_suspended().

2.8.2.8 timer:get_remaining_time()

Returns the remaining time of this timer.

- Return value (number): The time remaining in milliseconds. `0` means that the timer is finished (or will finish in the current cycle) or was canceled.

2.8.2.9 timer:set_remaining_time(remaining_time)

Changes the remaining time of this timer.

This function has no effect if the timer is already finished.

- `remaining_time` (number): The time remaining in milliseconds. `0` makes the timer finish now and immediately executes its callback.

Remarks

When a timer is repeated (that is, if its callback returns `true`), the timer gets rescheduled with its initial delay again, no matter if you called this function in the meantime.

2.9 Drawable objects

Drawable objects are things that can be drawn on a destination surface. They include the following types: surface, text surface and sprite. This page describes the methods common to those types.

2.9.1 Methods of all drawable types

These methods exist in all drawable types.

2.9.1.1 drawable:draw(dst_surface, [x, y])

Draws this object on a destination surface.

- `dst_surface` (surface): The destination surface.
- `x` (number, optional): X coordinate of where to draw this object (default `0`).
- `y` (number, optional): Y coordinate of where to draw this object. (default `0`).

2.9.1.2 drawable:draw_region(region_x, region_y, region_width, region_height, dst_surface, [x, y])

Draws a subrectangle of this object on a destination surface.

- `region_x` (number): X coordinate of the subrectangle to draw.
- `region_y` (number): Y coordinate of the subrectangle to draw.
- `region_width` (number): Width of the subrectangle to draw.
- `region_height` (number): Height of the subrectangle to draw.
- `dst_surface` (surface): The destination surface.
- `x` (number, optional): X coordinate of where to draw this rectangle on the destination surface (default `0`).
- `y` (number, optional): Y coordinate of where to draw this rectangle. on the destination surface (default `0`).

2.9.1.3 drawable:get_blend_mode()

Returns the blend mode of this drawable object.

The blend mode defines how this drawable object will be drawn on other surfaces when you call drawable:draw() or drawable:draw_region().

- Return value (string): The blend mode. See drawable:set_blend_mode() for the possible values.

2.9.1.4 drawable:set_blend_mode(blend_mode)

Sets the blend mode of this drawable object.

The blend mode defines how this drawable object will be drawn on other surfaces when you call drawable:draw() or drawable:draw_region().

- `blend_mode` (string): The blend mode. Can be one of:
  - `"none"`: No blending. The destination surface is replaced by the pixels of this drawable object.
    ```
    dstRGBA = srcRGBA
    ```
  - `"blend"` (default): This drawable object is alpha-blended onto the destination surface.
    ```
    dstRGB = (srcRGB * srcA) + (dstRGB * (1 - srcA))
    dstA = srcA + (dstA * (1 - srcA))
    ```
  - `"add"`: This drawable object is drawn onto the destination surface with additive blending. The clarity of the destination surface is kept. Useful to color and lighten the destination surface.
    ```
    dstRGB = (srcRGB * srcA) + dstRGB
    ```
  - `"multiply"`: Color modulation. Can be used to darken the destination surface without degrading its content.
    ```
    dstRGB = srcRGB * dstRGB
    ```

2.9.1.5 drawable:fade_in([delay], [callback])

Starts a fade-in effect on this object.

You can specify a callback function to be executed when the fade-in effect finishes.

If the drawable object is a sprite attached to a map entity during a game, the fade-in effect gets the lifetime of that entity. The behavior is probably what you expect: the fade-in effect gets suspended when the entity gets suspended, and it gets canceled (that is, the callback is never executed) when the map entity is destroyed.

- `delay` (number, optional): Delay in milliseconds between two frames of the fade-in animation (default `20`).
- `callback` (function, optional): A function to call when the fade-in effect finishes.

Note

When your drawable object does not belong to a map entity (typically in a title screen before a game is started, or in your pause menu), the fade-in effect continues until the drawable object is garbage-collected. In other words, the callback can be executed even if you have stopped using the drawable object in the meantime. Therefore, you should use the `callback` parameter with care. In these situations, using a timer for your callback is easier because timers have an explicit lifetime.

2.9.1.6 drawable:fade_out([delay], [callback])

Starts a fade-out effect on this object.

You can specify a callback function to be executed when the fade-out effect finishes.

If the drawable object is a sprite attached to a map entity during a game, the fade-out effect gets the lifetime of that entity. The behavior is probably what you expect: the fade-out effect gets suspended when the entity gets suspended, and it gets canceled (that is, the callback is never executed) when the map entity is destroyed.

- `delay` (number, optional): Delay in milliseconds between two frames of the fade-out animation (default `20`).
- `callback` (function, optional): A function to call when the fade-out effect finishes.

Note

When your drawable object does not belong to a map entity (typically in a title screen before a game is started, or in your pause menu), the fade-out effect continues until the drawable object is garbage-collected. In other words, the callback can be executed even if you have stopped using the drawable object in the meantime. Therefore, you should use the `callback` parameter with care. In these situations, using a timer for your callback is easier because timers have an explicit lifetime.

2.9.1.7 drawable:get_xy()

Returns the offset added where this drawable object is drawn.

This value is initially `0,0`. It is added to whatever coordinates the object is drawn at.

They can be modified by a movement or by drawable:set_xy().

- Return value 1 (number): X offset of the drawable object.
- Return value 2 (number): Y offset of the drawable object.

2.9.1.8 drawable:set_xy(x, y)

Sets the offset added where this drawable object is drawn.

This value is initially `0,0`. It is added to whatever coordinates the object is drawn at.

- `x` (number): X offset to set.
- `y` (number): Y offset to set.

2.9.1.9 drawable:get_movement()

Returns the current movement of this drawable object.

- Return value (movement): The current movement, or `nil` if the drawable object is not moving.

2.9.1.10 drawable:stop_movement()

Stops the current movement of this drawable object if any.

2.9.2 Surfaces

A surface is a 2D image. It is essentially a rectangle of pixels. Its main feature is that you can draw objects on it.

2.9.2.1 Functions of sol.surface

2.9.2.1.1 sol.surface.create([width, height])

Creates an empty surface.

- `width` (number, optional): Width of the surface to create in pixels. The default value is the width of the logical screen.
- `height` (number, optional): Height of the surface to create in pixels. The default value is the height of the logical screen.
- Return value (surface): The surface created.

2.9.2.1.2 sol.surface.create(file_name, [language_specific])

Creates a surface from an image file.

- `file_name` (string): Name of the image file to load.
- `language_specific` (boolean, optional): `true` to load the image from the `images` subdirectory of the current language directory (default is `false` and loads the image from the `sprites` directory).
- Return value (surface): The surface created, or `nil` if the image file could not be loaded.

2.9.2.2 Methods inherited from drawable

Surfaces are particular drawable objects. Therefore, they inherit all methods from the type drawable.

See Methods of all drawable types to know these methods.

2.9.2.3 Methods of the type surface

The following methods are specific to surfaces.

2.9.2.3.1 surface:get_size()

Returns the size of this surface.

- Return value 1 (number): Width of the surface in pixels.
- Return value 2 (number): Height of the surface in pixels.

2.9.2.3.2 surface:clear()

Erases all pixels of this surface.

All pixels become transparent. The opacity property and the size of the surface are preserved.

2.9.2.3.3 surface:fill_color(color, [x, y, width, height])

Fills a region of this surface with a color.

If no region is specified, the entire surface is filled. If the color has an alpha component different from 255 (opaque), then the color is blended onto the existing pixels.

- `color` (table): The color as an array of 3 RGB values or 4 RGBA values (`0` to `255`).
- `x` (number, optional): X coordinate of the region to fill on this surface.
- `y` (number, optional): Y coordinate of the region to fill on this surface.
- `width` (number, optional): Width of the region to fill on this surface.
- `height` (number, optional): Height of the region to fill on this surface.

2.9.2.3.4 surface:get_opacity()

Returns the opacity of this surface.

- Return value (integer): The opacity: `0` (transparent) to `255` (opaque).

2.9.2.3.5 surface:set_opacity(opacity)

Sets the opacity of this surface.

All surfaces are initially opaque.

- `opacity` (integer): The opacity: `0` (transparent) to `255` (opaque).

2.9.3 Text surfaces

A text surface is a single line of text that you can display. A text surface can be seen as a special surface able to contain text.

2.9.3.1 Functions of sol.text_surface

2.9.3.1.1 sol.text_surface.create([properties])

Creates a text surface with the specified properties.

- `properties` (optional table): A table that describes all properties of the text surface to create. Its key-value pairs are all optional, they can be:
  - `horizontal_alignment` (string, default `"left"`): `"left"`, `"center"` or `"right"`. When you draw the text surface at some coordinates on a destination surface, it is anchored at this position.
  - `vertical_alignment` (string, default `"middle"`): `"top"`, `"middle"` or `"bottom"`. When you draw the text surface at some coordinates on a destination surface, it is anchored at this position.
  - `font` (string, default the first one in alphabetical order): Name of the font file to use, relative to the `fonts` directory and without extension. It must be a declared in the quest resource list. The following extensions are auto-detected in this order: `.png`, `.ttf`, `.ttc` and `.font`.
  - `rendering_mode` (string, default `"solid"`): `"solid"` (faster) or `"antialiasing"` (smooth effect on letters).
  - `color` (table, default white): Color of the text to draw (array of 3 RGB values between 0 and 255). No effect on bitmap fonts.
  - `font_size` (number, default `11`): Font size to use. No effect on bitmap fonts.
  - `text` (string, default `""`): The text to show (must be valid UTF-8).
  - `text_key` (string, default `nil`): Key of the localized text to show. The string must exist in the file text/strings.dat of the current language.
  - Return value (text surface): The text surface created.

2.9.3.2 Methods inherited from drawable

Text surfaces are particular drawable objects. Therefore, they inherit all methods from the type drawable.

See Methods of all drawable types to know these methods.

2.9.3.3 Methods of the type text surface

The following methods are specific to text surfaces.

2.9.3.3.1 text_surface:get_horizontal_alignment()

Returns the horizontal alignment of the text.

When you draw the text surface at some coordinates on a destination surface, it is anchored at this position.

- Return value (string): `"left"`, `"center"` or `"right"`.

2.9.3.3.2 text_surface:set_horizontal_alignment(horizontal_alignment)

Sets the horizontal alignment of the text.

When you draw the text surface at some coordinates on a destination surface, it is anchored at this position.

- `horizontal_alignment` (string): `"left"`, `"center"` or `"right"`.

2.9.3.3.3 text_surface:get_vertical_alignment()

Returns the vertical alignment of the text.

When you draw the text surface at some coordinates on a destination surface, it is anchored at this position.

- Return value (string): `"top"`, `"middle"` or `"bottom"`.

2.9.3.3.4 text_surface:set_vertical_alignment(vertical_alignment)

Sets the vertical alignment of the text.

When you draw the text surface at some coordinates on a destination surface, it is anchored at this position.

- `vertical_alignment` (string): `"top"`, `"middle"` or `"bottom"`.

2.9.3.3.5 text_surface:get_font()

Returns the font used to draw this text surface.

- Return value (string): Id of the font of this text surface.

2.9.3.3.6 text_surface:set_font(font_id)

Sets the font used to draw this text surface.

- `font_id` (string): Name of the font file to use, relative to the `fonts` directory and without extension. It must be a declared in the quest resource list. The following extensions are auto-detected in this order: `.png`, `.ttf`, `.ttc` and `.fon`.

2.9.3.3.7 text_surface:get_rendering_mode()

Returns the rendering mode of the text.

- Return value (string): `"solid"` (faster) or `"antialiasing"` (smooth effect on letters).

2.9.3.3.8 text_surface:set_rendering_mode(rendering_mode)

Sets the rendering mode of the text.

- `rendering_mode` (string): `"solid"` (faster) or `"antialiasing"` (smooth effect on letters).

2.9.3.3.9 text_surface:get_color()

Returns the color used to draw the text.

This only has an effect for outline fonts.

- Return value (table): The text color as an array of 3 RGB values (`0` to `255`).

2.9.3.3.10 text_surface:set_color(color)

Sets the color used to draw the text.

This only has an effect for outline fonts.

- `color` (table): The text color as an array of 3 RGB values (`0` to `255`).

2.9.3.3.11 text_surface:get_font_size()

Returns the font size used to draw the text.

This only has an effect for outline fonts.

- Return value (number): The font size.

2.9.3.3.12 text_surface:set_font_size(font_size)

Sets the size used to draw the text.

This only has an effect for outline fonts.

- `font_size` (number): The font size.

2.9.3.3.13 text_surface:get_text()

Returns the string displayed in this object.

- Return value (string): The current text (possibly an empty string).

2.9.3.3.14 text_surface:set_text([text])

Sets the string displayed in this object.

The string must be encoded in UTF-8.

- `text` (string, optional): The text to set. No value or an empty string mean no text.

2.9.3.3.15 text_surface:set_text_key(key)

Sets the text as a localized string in the current language.

- `key` (string): Key of the text to set.

Remarks

This function is equivalent to `text_surface:set_text(sol.language.get_string(key))`.

2.9.3.3.16 text_surface:get_size()

Returns the size of this text surface.

- Return value 1 (number): Width of the text surface in pixels.
- Return value 2 (number): Height of the text surface in pixels.

Remarks

Note that you cannot set the size of a text surface. The size is determined by the text and the font.

2.9.4 Sprites

A sprite is an animated image. It is managed by an animation set. The animation set defines which animations are available and describes, for each animation, a sequence of images in each direction.

A sprite has the following properties:

- a current animation from its animation set (like `"walking"` or `"hurt"`),
- a current direction that indicates where the sprite is facing,
- a current frame: the index of the current individual image in the sequence.

The animation set of a sprite is composed of one or several PNG images that store all the frames, and a data file that describes how frames are organized in the PNG images. The data file also indicates the delay to make between frames when animating them and other properties like whether the animation should loop. See the sprites syntax for more information about the format of sprites.

We describe here the Lua API that you can use to show sprites during your game or your menus.

2.9.4.1 Functions of sol.sprite

2.9.4.1.1 sol.sprite.create(animation_set_id)

Creates a sprite.

- `animation_set_id` (string): Name of the animation set to use. This name must correspond to a valid sprite sheet data file in the `sprites` directory (without its extension).
- Return value (sprite): The sprite created, or `nil` if the sprite data file could not be loaded.

2.9.4.2 Methods inherited from drawable

Sprites are particular drawable objects. Therefore, they inherit all methods from the type drawable.

See Methods of all drawable types to know these methods.

2.9.4.3 Methods of the type sprite

The following methods are specific to sprites.

2.9.4.3.1 sprite:get_animation_set()

Returns the name of the animation set used by this sprite.

- Return value (string): Name of the animation set used by this sprite. This name corresponds to a sprite sheet data file in the `sprites` directory (without its extension).

2.9.4.3.2 sprite:get_animation()

Returns the name of the current animation of this sprite.

- Return value (string): Name of the current animation.

2.9.4.3.3 sprite:set_animation(animation_name, [next_action])

Sets the current animation of this sprite.

- `animation_name` (string): Name of the animation to set. This animation must exist in the animation set.
- `next_action` (function or string, optional): What to do when the animation finishes. This parameter only has an effect if the animation finishes, that is, if it does not loop. If you pass a function, this function will be called when the animation finishes. If you pass a string, this should be the name of an animation of the sprite, and this animation will automatically be played next. No value means no action after the animation finishes: in this case, the sprite stops being displayed.

2.9.4.3.4 sprite:has_animation(animation_name)

Returns whether the specified animation exists on a sprite.

- `animation_name` (string): Name of the animation to check.
- Return value (boolean): `true` if the sprite animation set contains an animation with this name.

2.9.4.3.5 sprite:get_direction()

Returns the current direction of this sprite.

- Return value (number): The current direction (the first one is `0`).

2.9.4.3.6 sprite:set_direction(direction)

Sets the current direction of this sprite.

- `direction` (number): The direction to set (the first one is `0`). This direction must exist in the current animation.

2.9.4.3.7 sprite:get_num_directions([animation_name])

Returns the number of direction of an animation of this sprite.

- `animation_name` (string, optional): Name of an animation of the sprite. This animation must exist in the animation set. No value means the current animation.
- Return value (number): The number of directions in this animation.

2.9.4.3.8 sprite:get_frame()

Returns the index of the current frame of this sprite.

- Return value (number): The current frame (the first one is `0`).

2.9.4.3.9 sprite:set_frame(frame)

Sets the current frame of this sprite.

- `frame` (number): The frame to set (the first one is `0`). This frame must exist in the current direction.

2.9.4.3.10 sprite:get_num_frames()

Returns the number of frames of this sprites in the current animation and direction.

- Return value (number): The number of frames.

2.9.4.3.11 sprite:get_frame_delay()

Returns the delay between two frames of this sprite in the current animation.

This delay is defined by the current animation but may be overriden by set_frame_delay().

- Return value (number): The delay in milliseconds between two frames in the current animation. `nil` means infinite and it is only allowed for single-frame animations.

2.9.4.3.12 sprite:set_frame_delay(delay)

Changes the delay between two frames of this sprite in the current animation.

Use this function if you want to override the normal delay (the one defined in the sprite data file).

- `delay` (number): The new delay in milliseconds. `nil` means infinite and is only allowed for single-frame animations.

2.9.4.3.13 sprite:get_size()

Returns the frame size of this sprite in the current animation and direction.

The frame size is the same for all frames of a given animation and direction.

- Return value 1 (number): The width of a frame.
- Return value 2 (number): The height of a frame.

2.9.4.3.14 sprite:get_origin()

Returns the coordinates of the origin point of this sprite in the current animation and direction, relative to the upper left corner of a frame.

The origin is the point of synchronization for sprites that have several animations or directions of different sizes, and for entity sprites that are larger than the entity itself.

See entity:get_origin() for more details.

In a given animation and direction of a sprite, the origin point is the same for all frames.

- Return value 1 (number): X coordinate of the origin point.
- Return value 2 (number): Y coordinate of the origin point.

2.9.4.3.15 sprite:is_paused()

Returns whether this sprite is paused.

- Return value (boolean): `true` if this sprite is paused.

2.9.4.3.16 sprite:set_paused([paused])

Pauses or resumes the animation of this sprite.

- `paused` (boolean, optional): `true` to pause the sprite, `false` to unpause it. No value means `true`.

2.9.4.3.17 sprite:set_ignore_suspend([ignore])

During a game, sets whether the animation should continue even when the map is suspended.

- `ignore` (boolean, optional): `true` to continue animation even when the map is suspended, `false` otherwise. No value means `true`.

2.9.4.3.18 sprite:synchronize([reference_sprite])

Synchronizes the frames of this sprite with the frames of a reference sprite. The synchronization will be performed whenever both animation names match. The current sprite will no longer apply its normal frame delay: instead, it will now always set its current frame to the current frame of its reference sprite.

- `reference_sprite` (sprite, optional): The reference sprite. `nil` means stopping any previous synchronization.

2.9.4.4 Events of the type sprite

Events are callback methods automatically called by the engine if you define them.

The following events are specific to sprites.

2.9.4.4.1 sprite:on_animation_finished(animation)

Called when the current animation of this sprite is finished.

If the animation loops, this function is never called. Unless you start another animation, the sprite is no longer shown.

- `animation` (string): Name of the animation that has just finished.

2.9.4.4.2 sprite:on_animation_changed(animation)

Called whenever the animation of this sprite has changed.

- `animation` (string): Name of the new animation.

2.9.4.4.3 sprite:on_direction_changed(animation, direction)

Called whenever the direction of this sprite has changed.

- `animation` (string): Name of the current animation.
- `direction` (number): The new current direction (the first one is `0`).

2.9.4.4.4 sprite:on_frame_changed(animation, frame)

Called whenever the frame of this sprite has changed.

- `animation` (string): Name of the current animation.
- `frame` (number): The new current frame (the first one is `0`).

2.10 Movements

If you need to move an enemy of the map, a sprite in a menu or simply an arbitrary point, you can create a movement object and set its properties. There are several types of movements. They differ by the kind of trajectory they can make. When you create a movement, you obtain a value of the movement type you chose. Then, to get and set its properties (like the speed, the angle, etc.), a movement object has several methods available. As detailed below, the methods available differ depending on the movement type because all movement types don't have the same properties.

The following movement types are available.

- Straight movement: Rectilinear trajectory in any direction.
- Random movement: A straight movement whose direction changes randomly from time to time.
- Target movement: Straight trajectory towards a possibly moving target.
- Path movement: Predetermined path composed of steps in the 8 main directions.
- Random path movement: Like a path movement, but with random steps.
- Path finding movement: Like a path movement, but calculated to reach a possibly moving target.
- Circle movement: Circular trajectory around a possibly moving center.
- Jump movement: An illusion of jump above a baseline.
- Pixel movement: A trajectory described pixel by pixel.

This page desribes the methods and callbacks common to all movement types.

Movements can be applied in-game to map entities, but also outside a game, typically in a menu to move a sprite, an image or just an `(x,y)` value. However, some properties of movements (like movement:set_ignore_obstacles()) only take effect in the case of a map entity because they refer to map-specific notions like obstacles.

2.10.1 Functions of sol.movement

2.10.1.1 sol.movement.create(movement_type)

Creates a movement.

Depending on the movement type, several methods are then available to get and set its properties.

- `movement_type` (string): Type of movement to create. Must be one of:
  - `"straight"`: Follows a rectilinear trajectory.

  - `"random"`: Like "straight" but with random, changing angles.
  - `"target"`: Like "straight" but goes into the direction of a fixed point or a moving entity.
  - `"path"`: Follows a specified succession of basic moves on an 8x8 pixels grid.
  - `"random_path"`: Like "path" but computes the path randomly.
  - `"path_finding"`: Like "path" but computes the shortest path to an entity, avoiding obstacles of the map (only possible in game).
  - `"circle"`: Follows a circular trajectory around a center.
  - `"jump"`: Makes a jump above a rectilinear trajectory.
  - `"pixel"`: Follows a trajectory specified pixel by pixel.

- Return value (movement): The movement created. See the sections below to know the get and set methods available for your movement type.

2.10.2 Methods of all movement types

These methods exist in all movement types.

2.10.2.1 movement:start(object_to_move, [callback])

Starts this movement on an object.

The movement will be applied until it finishes (if it has an end) or until it is replaced by another one. It does not matter if the movement gets out of scope in your Lua script.

- `object_to_move` (map entity, drawable object or table): The object to move. It may be a map entity, a drawable object or a table with two fields `x` and `y`. In the case of the table, if the fields `x` and `y` don't exist, they are created and initialized to `0`.

  An empty table will be initialized with `{x = 0, y = 0}`.

- `callback` (function, optional): A function to call when the movement finishes.

Remarks

The hero is a map entity just like any other. So you can apply a custom movement to him using this function. The usual way to do this is to call hero:freeze() first to properly remove control from the player, and then to start the movement. When you have finished, you can restore the control with hero:unfreeze(). Indeed, changing the movement while the hero is in a state other than `"frozen"` might give surprising results. Your movement will be applied, replacing any built-in movement of the state, but whatever was happening in the state will still continue. Furthermore, your movement will disappear as soon as the state changes. So don't start a movement on the hero during an arbitrary state unless you know what you are doing.

2.10.2.2 movement:stop()

Stops this movement and detaches it from the object that was moved.

2.10.2.3 movement:get_xy()

Returns the coordinates of the object controlled by this movement.

The object controlled by this movement may be a map entity, a drawable object or a point.

- Return value 1 (number): X coordinate.
- Return value 2 (number): Y coordinate.

2.10.2.4 movement:set_xy(x, y)

Sets the coordinates of the object controlled by this movement.

The object controlled by this movement may be a map entity, a drawable object or a point.

- `x` (number): X coordinate to set.
- `y` (number): Y coordinate to set.

2.10.2.5 movement:get_ignore_obstacles()

Returns whether this movement ignores obstacles of the map.

If the movement is not attached to a map entity yet, it is not an error to call this function: the result will have an effect when the movement gets attached to a map entity.

- Return value (boolean): `true` if this movement ignores obstacles.

2.10.2.6 movement:set_ignore_obstacles([ignore_obstacles])

Sets whether a map entity controlled by this movement should ignore obstacles of the map.

If the movement is not attached to a map entity yet, it is not an error to call this function: your choice will have an effect when the movement gets attached to a map entity.

- `ignore_obstacles` (boolean, optional): `true` to make this movement ignore obstacles of the map (no value means `true`).

2.10.2.7 movement:get_direction4()

From the four main directions, returns the closest one to the current trajectory.

East is 0, North is 1, West is 2, South is 3. As the real trajectory does not necessarily follows one of the four main directions, it will be converted to the closest one.

If you use this movement to control a sprite (or a map entity that has a sprite), you can use this function to make the sprite face the direction of the movement.

- Return value (number): The closest direction corresponding to the angle of this movement.

Example of use:

```
-- Example of code from an enemy script.

-- This function is called when the enemy should start or restart its movement.
function enemy:on_restarted()

  -- Create a movement that makes random straight trajectories.
  local movement = sol.movement.create("random")

  -- This function is called when the trajectory has changed.
  function movement:on_movement_changed()
    -- The angle of the movement has changed: update the sprite accordingly.
    local direction = movement:get_direction4()
    enemy:get_sprite():set_direction(direction)
  end

  movement:start(enemy)
end
```

2.10.3 Events of all movement types

Events are callback methods automatically called by the engine if you define them.

The following events are common to all movement types.

Remarks

All movement events are here (it turns out that no specific movement type define additional events).

2.10.3.1 movement:on_position_changed()

Called when the coordinates controlled by this movement have just changed.

2.10.3.2 movement:on_obstacle_reached()

During a game, called when the coordinates controlled by this movement have just failed to change because they would lead the map entity controlled into an obstacle of the map.

When an obstacle is reached, this event is called instead of movement:on_position_changed().

This event can only be called when all of these conditions are met:

- A game is currently running.
- The movement is attached to a map entity (like an enemy, an NPC, etc.).
- The movement does not ignore obstacles (i.e. movement:get_ignore_obstacles() returns `false`).

Remarks

When the movement attempts to change the coordinates, one of movement:on_position_changed() or movement:on_obstacle_reached() is guaranteed to be called.

2.10.3.3 movement:on_changed()

Called when the characteristics of this movement (like speed or angle) have just changed.

2.10.3.4 movement:on_finished()

Called when this movement has just finished (if there is an end).

2.10.4 Straight movement

A straight movement follows a rectilinear trajectory. You can define the trajectory as an angle and a speed.

2.10.4.1 Methods inherited from movement

Straight movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.4.2 Methods of the type straight movement

The following methods are specific to straight movements.

2.10.4.2.1 straight_movement:get_speed()

Returns the speed of this movement.

- Return value (number): The speed in pixels per second.

2.10.4.2.2 straight_movement:set_speed(speed)

Sets the speed of this movement.

- `speed` (number): The new speed in pixels per second.

2.10.4.2.3 straight_movement:get_angle()

Returns the angle of the trajectory in radians.

East is 0, North is `math.pi / 2`, West is `math.pi`, South is `3 * math.pi / 2` and any intermediate value is possible.

- Return value (number): The angle in radians.

Remarks

If you prefer a value in a 4-direction system, see movement:get_direction4().

2.10.4.2.4 straight_movement:set_angle(angle)

Sets the angle of the trajectory in radians.

East is 0, North is `math.pi / 2`, West is `math.pi`, South is `3 * math.pi / 2` and any intermediate value is possible. Negative values and values greater to `2 * math.pi` are also accepted.

- `angle` (number): The new angle in radians.

2.10.4.2.5 straight_movement:get_max_distance()

Returns the maximum distance of this movement.

The movement will stop when this distance is reached.

- Return value (number): The maximum distance in pixels (`0` means no limit).

2.10.4.2.6 straight_movement:set_max_distance(max_distance)

Sets the maximum distance of this movement.

The movement will stop when this distance is reached.

- `max_distance` (number): The maximum distance in pixels (`0` means no limit).

2.10.4.2.7 straight_movement:is_smooth()

Returns whether this movement adjusts its trajectory when an obstacle of the map is reached. This property has no effect if the movement is not attached to a map entity or if the movement ignores obstacles.

- Return value (boolean): `true` if this movement is smooth.

2.10.4.2.8 straight_movement:set_smooth([smooth])

Sets whether this movement should adjust its trajectory when an obstacle of the map is reached. This property has no effect if the movement is not attached to a map entity or if the movement ignores obstacles.

- `smooth` (boolean, optional): `true` to make this movement smooth. No value means `true`.

2.10.4.3 Events inherited from movement

Straight movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.10.5 Random movement

This type of movement is a rectilinear movement whose trajectory changes randomly over time. It can be seen as a particular case of the straight movement type, where the angle is automatically changed after random delays.

2.10.5.1 Methods inherited from movement

Random movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.5.2 Methods of the type random movement

The following methods are specific to random movements.

2.10.5.2.1 random_movement:get_speed()

Returns the speed applied to this movement when it is started.

- Return value (number): The speed in pixels per second.

2.10.5.2.2 random_movement:set_speed(speed)

Sets the speed applied to this movement when it is started.

- `speed` (number): The new speed in pixels per second.

2.10.5.2.3 random_movement:get_angle()

Returns the angle of the current trajectory in radians.

East is 0, North is `math.pi / 2`, West is `math.pi`, South is `3 * math.pi / 2` and any intermediate value is possible.

- Return value (number): The angle in radians.

Remarks

If you prefer a value in a 4-direction system, see movement:get_direction4().

2.10.5.2.4 random_movement:get_max_distance()

Returns the maximum distance of this movement.

If the movement goes further than this distance, it automatically comes back towards the initial position.

- Return value (number): The maximum distance in pixels (`0` means no limit).

2.10.5.2.5 random_movement:set_max_distance(max_distance)

Sets the maximum distance of this movement.

If the movement goes further than this distance, it automatically comes back towards the initial position.

- `max_distance` (number): The maximum distance in pixels (`0` means no limit).

2.10.5.2.6 random_movement:is_smooth()

Returns whether this movement adjusts its trajectory when an obstacle of the map is reached. This property has no effect if the movement is not attached to a map entity or if the movement ignores obstacles.

- Return value (boolean): `true` if this movement is smooth.

2.10.5.2.7 random_movement:set_smooth([smooth])

Sets whether this movement should adjust its trajectory when an obstacle of the map is reached. This property has no effect if the movement is not attached to a map entity or if the movement ignores obstacles.

- `smooth` (boolean, optional): `true` to make this movement smooth. No value means `true`.

2.10.5.3 Events inherited from movement

Random movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.10.6 Target movement

A target movement goes towards a target point. The target point can be a fixed point of an entity of the map. If the target is a moving map entity, the movement updates its angle to continue to go towards the entity. By default, the target is the hero when a game is running.

This type of movement can be seen as a particular case of the straight movement type, where the angle is set automatically to go towards the target.

Remarks

This type of movement goes straight towards the target. If you set the `smooth` property to `true`, it will try to avoid simple obstacles by moving to a side. This is usually enough for simple enemies that target the hero. If you want a more complex technique that calculates an intelligent path to the target, see the path finding movement type.

2.10.6.1 Methods inherited from movement

Target movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.6.2 Methods of the type target movement

The following methods are specific to target movements.

2.10.6.2.1 target_movement:set_target(x, y), target_movement:set_target(entity, [x, y])

Sets the target of this movement as a fixed point or a map entity.

To target a fixed point:

- `x` (number): X coordinate of the target.
- `y` (number): Y coordinate of the target.

To target a map entity (only during a game):

- `entity` (entity): The entity to target.
- `x` (number, optional): X offset to add to the target entity's coordinates. Default is `0`.
- `y` (number, optional): Y offset to add to the target entity's coordinates. Default is `0`.

2.10.6.2.2 target_movement:get_speed()

Returns the speed of this movement.

- Return value (number): The speed in pixels per second.

2.10.6.2.3 target_movement:set_speed(speed)

Sets the speed of this movement.

- `speed` (number): The new speed in pixels per second.

2.10.6.2.4 target_movement:get_angle()

Returns the angle of the trajectory in radians.

East is 0, North is `math.pi / 2`, West is `math.pi`, South is `3 * math.pi / 2` and any intermediate value is possible.

- Return value (number): The angle in radians.

Remarks

If you prefer a value in a 4-direction system, see movement:get_direction4().

2.10.6.2.5 target_movement:is_smooth()

Returns whether this movement adjusts its trajectory when an obstacle of the map is reached. This property has no effect if the movement is not attached to a map entity or if the movement ignores obstacles.

- Return value (boolean): `true` if this movement is smooth.

2.10.6.2.6 target_movement:set_smooth([smooth])

Sets whether this movement should adjust its trajectory when an obstacle of the map is reached. This property has no effect if the movement is not attached to a map entity or if the movement ignores obstacles.

- `smooth` (boolean, optional): `true` to make this movement smooth. No value means `true`.

2.10.6.3 Events inherited from movement

Target movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.10.7 Path movement

A path movement follows a specified path on an 8∗8 pixels grid, in an 8-direction system. A path is a succession of steps of 8 pixels in one of the 8 main directions. You can define each step of the path and make it repeated if you want.

2.10.7.1 Methods inherited from movement

Path movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.7.2 Methods of the type path movement

The following methods are specific to path movements.

2.10.7.2.1 path_movement:get_path()

Returns the path of this movement.

- Return value (table): The path as an array of integers. Each value is a number between 0 and 7 that represents a step (move of 8 pixels) in the path. 0 is East, 1 is North-East, etc.

2.10.7.2.2 path_movement:set_path(path)

Sets the path of this movement.

- `path` (table): The path as an array of integers. Each value is a number between 0 and 7 that represents a step (move of 8 pixels) in the path. 0 is East, 1 is North-East, etc.

Example of use:

```
-- Example of code from a map script.
-- Assume that there is an NPC called "scared_cat" on this map.
function scared_cat:on_interaction()
  -- The hero is talking to me: run away!
  sol.audio.play_sound("meow")
  local movement = sol.movement.create("path")
  -- This path is arbitrary, it's just an example (North and then West).
  movement:set_path{2,2,2,2,2,2,4,4,4,4}
  movement:set_speed(80)
  self:start_movement(movement)
end
```

2.10.7.2.3 path_movement:get_speed()

Returns the speed of this movement.

- Return value (number): The speed in pixels per second.

2.10.7.2.4 path_movement:set_speed(speed)

Sets the speed of this movement.

- `speed` (number): The new speed in pixels per second.

2.10.7.2.5 path_movement:get_loop()

Returns whether this movement repeats itself once the end of the path is reached.

- Return value (boolean): `true` if the path repeats itself.

2.10.7.2.6 path_movement:set_loop([loop])

Sets whether this movement repeats itself once the end of the path is reached.

- `loop` (boolean, optional): `true` to make the path repeat itself. No value means `true`.

2.10.7.2.7 path_movement:get_snap_to_grid()

Returns whether this movement automatically snaps to the map grid the map entity that it controls.

The map grid is composed of squares of 8∗8 pixels. All tiles are aligned to the grid. This property has no effect if there is no current map of if this movement is not attached to a map entity.

- Return value (boolean): `true` if this movement automatically snaps its map entity to the map grid.

2.10.7.2.8 path_movement:set_snap_to_grid([snap])

Sets whether this movement should automatically snap to the map grid the map entity that it controls.

The map grid is composed of squares of 8∗8 pixels. All tiles are aligned to the grid. This property has no effect if there is no current map of if this movement is not attached to a map entity.

- `snap` (boolean, optional): `true` to make this movement automatically snap its map entity to the map grid. No value means `true`.

2.10.7.3 Events inherited from movement

Path movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.10.8 Random path movement

The random path movement is a particular case of path movement, where the path is chosen automatically. The resulting movement is composed of repeated random steps in the four main directions only and with a length that is a multiple of 8 pixels.

Remarks

This movement is a typical random walk movement. You will probably use it for NPCs.

Example of use:

```
-- Example of code from a map script.
-- Assume that there is an NPC called "bob" on this map.
function sol.map:on_started()
  -- The map has just started: make bob walk.
  bob:start_movement(sol.movement.create("random_path"))
end
```

2.10.8.1 Methods inherited from movement

Random path movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.8.2 Methods of the type random path movement

The following methods are specific to random path movements.

2.10.8.2.1 random_path_movement:get_speed()

Returns the speed of this movement.

- Return value (number): The speed in pixels per second.

2.10.8.2.2 random_path_movement:set_speed(speed)

Sets the speed of this movement.

- `speed` (number): The new speed in pixels per second.

2.10.8.3 Events inherited from movement

Random path movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.10.9 Path finding movement

A path finding movement is a particular path movement where the path is calculated to reach a target. The target is a map entity (by default the hero). The movement calculates repeatedly the shortest path towards the target entity, taking into account obstacles of the map. With this type of movement, an entity is capable of finding its way in a maze.

Remarks

This type of movement computes a precise path on the map grid and avoids complex obstacles by using a sophisticated A.I. algorithm (A*). If you just need to go straight towards a target, which may be more natural for basic enemies, see the target movement type.

2.10.9.1 Methods inherited from movement

Path finding movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.9.2 Methods of the type path finding movement

The following methods are specific to path finding movements.

2.10.9.2.1 path_finding_movement:set_target(entity)

Sets the target entity of this movement.

- `entity` (entity): The entity to target.

2.10.9.2.2 path_finding_movement:get_speed()

Returns the speed of this movement.

- Return value (number): The speed in pixels per second.

2.10.9.2.3 path_finding_movement:set_speed(speed)

Sets the speed of this movement.

- `speed` (number): The new speed in pixels per second.

2.10.9.3 Events inherited from movement

Path finding movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.10.10 Circle movement

A circle movement makes a circular trajectory around a center point or a map entity.

2.10.10.1 Methods inherited from movement

Circle movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.10.2 Methods of the type circle movement

The following methods are specific to circle movements.

2.10.10.2.1 circle_movement:set_center(x, y), circle_movement:set_center(entity, [dx, dy])

Sets the center of this movement as a fixed point or a map entity.

To make circles around a fixed point:

- `x` (number): X coordinate of the center.
- `y` (number): Y coordinate of the center.

To make circles around a map entity (only during a game):

- `entity` (entity): The center entity.
- `dx` (number, optional): X offset to add to the center entity's coordinates (default 0).
- `dy` (number, optional): Y offset to add to the center entity's coordinates (default 0).

2.10.10.2.2 circle_movement:get_radius()

Returns the radius of circles to make.

If circle_movement:get_radius_speed() is not 0, radius changes are made gradually.

- Return value (number): The wanted radius in pixels.

2.10.10.2.3 circle_movement:set_radius(radius)

Sets the radius of circles to make.

If circle_movement:get_radius_speed() is not 0, the radius will be updated gradually.

- `radius` (number): The new wanted radius in pixels.

2.10.10.2.4 circle_movement:get_radius_speed()

Returns the speed of radius changes.

- Return value (number): The speed in pixels per second, or 0 if radius changes are immediate.

2.10.10.2.5 circle_movement:set_radius_speed(radius_speed)

Sets the radius to be updated immediately of gradually (at the specified speed) towards its wanted value.

- `radius_speed` (number): The speed of radius changes in pixels per second, or 0 to make radius changes immediate.

2.10.10.2.6 circle_movement:is_clockwise()

Returns whether circles are made clockwise or counter-clockwise.

- Return value (boolean): `true` if circles are clockwise.

2.10.10.2.7 circle_movement:set_clockwise([clockwise])

Sets whether circles are made clockwise or counter-clockwise.

- `clockwise` (boolean, optional): `true` to make circles clockwise. No value means `true`.

2.10.10.2.8 circle_movement:get_initial_angle()

Returns the angle from where the first circle starts.

- Return value (number): The initial angle in degrees.

2.10.10.2.9 circle_movement:set_initial_angle(initial_angle)

Sets the angle from where the first circle should start.

- `initial_angle` (number): The initial angle in degrees.

2.10.10.2.10 circle_movement:get_angle_speed()

Returns the speed of the angle variation.

- Return value (number): The angle speed in degrees per second.

2.10.10.2.11 circle_movement:set_angle_speed(angle_speed)

Sets the speed of the angle variation.

- `angle_speed` (number): The new angle speed in degrees per second.

2.10.10.2.12 circle_movement:get_max_rotations()

Returns the maximum number of rotations of this movement.

When this number of rotations is reached, the movement stops.

- Return value (number): The maximum number of rotations to make (`0` means infinite).

2.10.10.2.13 circle_movement:set_max_rotations(max_rotations)

Sets the maximum number of rotations of this movement.

When this number of rotations is reached, the movement stops.

- `max_rotations` (number): The maximum number of rotations to make (`0` means infinite).

Remarks

The movement stops itself by setting its radius to 0. Therefore, if the radius is set to change gradually (see circle_movement:get_radius_speed()), the movement will continue for a while until the radius reaches 0.
When the movement has stopped, it restarts later if it was set to loop (see circle_movement:get_loop_delay()), and again, possibly gradually.

2.10.10.2.14 circle_movement:get_duration()

Returns the maximum duration of this movement.

When this delay is reached, the movement stops.

- Return value (number): The duration of the movement in milliseconds (`0` means infinite).

2.10.10.2.15 circle_movement:set_duration(duration)

Sets the maximum duration of this movement.

When this delay is reached, the movement stops.

- `duration` (number): The duration of the movement in milliseconds (`0` means infinite).

Remarks

The movement is stopped by automatically setting its radius to 0. Therefore, if the radius is set to change gradually (see circle_movement:get_radius_speed()), the movement will continue for a while until the radius reaches 0.
When the movement has stopped, it will then restart if it was set to loop (see circle_movement:set_loop_↩ delay()), and again, possibly gradually.

2.10.10.2.16 circle_movement:get_loop_delay()

Returns the delay after which this movement restarts.

- Return value (number): The restart delay in milliseconds (`0` means no restart).

2.10.10.2.17 circle_movement:set_loop_delay(loop_delay)

Sets the delay after which this movement restarts.

- `loop_delay` (number): The restart delay in milliseconds (`0` means no restart).

Remarks

This delay is applied if the movement get stopped by reaching the maximum number of rotations or the maximum duration.
When the movement restarts, the radius starts from 0 and gets back to its previous value, possibly gradually (see circle_movement:set_radius_speed()).

2.10.10.3 Events inherited from movement

Circle movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.10.11 Jump movement

This type of movement makes a jump above a specified rectilinear trajectory. We call this rectilinear trajectory the baseline. To use a jump movement, you typically specify the baseline (direction and distance), and the movement will jump above this baseline. The speed is adjusted automatically depending on the distance, but you can change it if you want.

For now, the baseline can only have one of the 8 main directions. This restriction may be removed in the future.

2.10.11.1 Methods inherited from movement

Jump movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.11.2 Methods of the type jump movement

The following methods are specific to jump movements.

2.10.11.2.1 jump_movement:get_direction8()

Returns the direction of the baseline of this jump. The baseline always has one of the 8 main directions. 0 is East, 1 is North-East, etc.

- Return value (number): The direction (`0` to `7`).

2.10.11.2.2 jump_movement:set_direction8(direction8)

Sets the direction of the baseline of this jump. The baseline always has one of the 8 main directions. 0 is East, 1 is North-East, etc.

- `direction8` (number): The direction (`0` to `7`).

2.10.11.2.3 jump_movement:get_distance()

Returns the distance of the baseline of this jump.

- Return value (number): The distance of the jump in pixels.

2.10.11.2.4 jump_movement:set_distance(distance)

Sets the distance of the baseline of this jump.

- `distance` (number): The new distance of the jump in pixels.

2.10.11.2.5 jump_movement:get_speed()

Returns the speed of this movement.

- Return value (number): The speed in pixels per second.

2.10.11.2.6 jump_movement:set_speed(speed)

Sets the speed of this movement.

- `speed` (number): The new speed in pixels per second. `0` means to set automatically a speed based on the distance of the jump.

2.10.11.3 Events inherited from movement

Jump movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.10.12 Pixel movement

A pixel movement makes a succession of basic translations, where each translation is a pixel-precise specified move (for example `(+2,-1)`). Each translation is immediate.

Unlike most other types of movements, there is no notion of speed in pixels per seconds. That's because a translation can transport the object instantly to another place - the movement is not necessarily continuous. Instead, you can set the delay between each translation.

Remarks

Pixel movements are not often needed. Most of the time, you don't want to specify pixel-by-pixel trajectories. Higher-level types of movements like path movement or target movement usually fit your needs when you move NPCs or enemies.

2.10.12.1 Methods inherited from movement

Pixel movements are particular movement objects. Therefore, they inherit all methods from the type movement.

See Methods of all movement types to know these methods.

2.10.12.2 Methods of the type pixel movement

The following methods are specific to pixel movements.

2.10.12.2.1 pixel_movement:get_trajectory()

Returns the trajectory of this movement.

- Return value (table): An array of all successive translations. Each translation is itself an array of two integers (x and y).

2.10.12.2.2 pixel_movement:set_trajectory(trajectory)

Sets the trajectory of this movement.

Any previous trajectory is removed and the movement starts at the beginning of the new trajectory.

- trajectory (table): An array of all successive translations to make. Each translation should be an array of two integers (x and y).

2.10.12.2.3 pixel_movement:get_loop()

Returns whether this movement restarts automatically when the trajectory is finished.

- Return value (boolean): `true` if the movement loops.

2.10.12.2.4 pixel_movement:set_loop([loop])

Sets whether this movement should restart automatically when the trajectory is finished.

- `loop` (boolean, optional): `true` to make the movement loop. No value means `true`.

2.10.12.2.5 pixel_movement:get_delay()

Returns the delay between two steps of the trajectory.

- Return value (number): The delay between two steps in milliseconds.

2.10.12.2.6 pixel_movement:set_delay(delay)

Sets the delay between two steps of the trajectory.

- `delay` (number): The delay between two steps in milliseconds.

2.10.12.3 Events inherited from movement

Pixel movements are particular movement objects. Therefore, they inherit all events from the type movement.

See Events of all movement types to know these events.

2.11 Game

This module provides a datatype "game" that represents a savegame.

2.11.1 Overview

2.11.1.1 Saved data

On a game object, you can access and modify everything that is saved. The data saved and restored by the engine are the following:

- starting location of the player (map and destination on this map),
- life (maximum and current),
- money (maximum and current),
- magic (maximum and current),
- built-in ability levels (like attacking, swimming, running, etc.).
- possession state of each saved equipment item,
- equipment item assigned to each item command slot,
- keyboard and joypad associations to game commands,
- any key-value pair that you need to store for your quest (see game:get_value() and game:set_value()).

When a game is running, more features are available (like pausing the game, handling game commands, etc.). Only one game can be running at a time.

2.11.1.2 Game commands

An important concept that comes with the game is the notion of game commands. Game commands are built-in game actions that can be mapped to a low-level keyboard or joypad input. They can be seen as an abstraction of the keyboard and the joypad. Game commands are like a virtual game device that provides the the following buttons:

- `"action"`: Contextual action such as talking, swimming, throwing, etc.
- `"attack"`: Main attack (using the sword).
- `"pause"`: Pausing or unpausing the game.
- `"item_1"`: Using the equipment item placed in slot 1 (see game:get_item_assigned())
- `"item_2"`: Using the equipment item placed in slot 2 (see game:get_item_assigned())

- `"right"`: Moving to the right.
- `"left"`: Moving to the left.
- `"up"`: Moving to the top.
- `"down"`: Moving to the bottom.

Of course, these virtual commands are mapped to real, low-level inputs from the keyboard and/or the joypad. You script can control which keyboard and joypad inputs are associated to each game command.

When a game command is pressed, no matter if the command came from the keyboard or the joypad, the engine performs some built-in behavior by default (like pausing the game or moving the hero to the right). But you can also extend or override this behavior, because the engine notifies you first (see game:on_command_pressed()). Therefore, you usually don't have to worry about the underlying keyboard or joypad input (but if you want to, you can, and the callbacks are exactly the sames as in sol.main and in menus).

2.11.1.3 Accessing the game like tables

Objects of the game type are userdata, but like most Solarus Lua types, they can also be accessed like tables. This is especially useful for the game type to add, next to the built-in game features, your own quest-specific functions and data (like your HUD and your pause menu).

2.11.2 Functions of sol.game

2.11.2.1 sol.game.exists(file_name)

Returns whether the specified savegame file exists.

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise savegames cannot be used and this function generates a Lua error.

- `file_name` (string): Name of the file to test, relative to the quest write directory.
- Return value (boolean): `true` if there exists a file with this name in the quest write directory.

2.11.2.2 sol.game.delete(file_name)

Deletes a savegame file.

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise savegames cannot be used and this function generates a Lua error.

- `file_name` (string): Name of the file to delete, relative to the quest write directory.

2.11.2.3 sol.game.load(file_name)

Loads an existing savegame, or initializes a new one if it does not exist (but does not save it).

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise savegames cannot be used and this function generates a Lua error.

- `file_name` (string): Name of a savegame file, relative to the to the quest write directory.
- Return value (game): The loaded (or created) game.

Remarks

This function does not start the game, it just loads the savegame file and initializes all equipment item scripts. Then you can access the data saved in the savegame file and use the API of equipment items. To actually run the game, call game:start().

2.11.3 Methods of the type game

2.11.3.1 game:save()

Saves this game into its savegame file.

A valid quest write directory must be set (in your quest.dat file or by calling sol.main.set_quest_write_dir()), otherwise savegames cannot be used and this function generates a Lua error.

2.11.3.2 game:start()

Runs this game.

This function is typically called from your savegame menu, when the player chooses its savegame file.

If another game was running, it is stopped automatically because only one game can be running at a time.

You can also call this function to restart the current game itself, even if it was not saved recently (saved data will not be reset). This may be useful to restart the game after the game-over sequence.

2.11.3.3 game:is_started()

Returns whether this game is currently running.

Only one game can be running at a time.

- Return value (boolean): `true` if this game is running.

2.11.3.4 game:is_suspended()

Returns whether this game is currently suspended.

The game is suspended when at least one of the following conditions is true:

- the game is paused,
- or a dialog is active,
- or the game-over sequence is active,
- or a transition between two maps is playing,
- or you explicitly called game:set_suspended(true).

- Return value (boolean): `true` if this game is currently suspended. Only possible when the game is running.

2.11.3.5 game:set_suspended([suspended])

Suspends or unsuspends the game.

Note that the game is also automatically suspended by the engine in the following situations:

- when the game is paused,
- or when a dialog is active,
- or when the game-over sequence is active,
- or when a transition between two maps is playing.

Therefore, if you call `game:set_suspended(false)` during one of these sequences, it will only take effect at the end of it.

- `suspended` (boolean, optional): `true` to suspend the game, `false` to resume it. No value means `true`.

Note

When the hero goes to another map, the game is automatically unsuspended after the map opening transition.

2.11.3.6 game:is_paused()

Returns whether this game is currently paused.

- Return value (boolean): `true` if this game is paused. Only possible when the game is running.

2.11.3.7 game:set_paused([paused])

Pauses or resumes the game explictly.

Note that by default, a built-in game command already exists to pause and unpause the game.

- `paused` (boolean, optional): `true` to pause the game, `false` to unpause it. Only possible when the game is running. No value means `true`.

2.11.3.8 game:is_pause_allowed()

Returns whether the player can pause or unpause the game.

- Return value (boolean): `true` if the player is allowed to pause the game.

2.11.3.9 game:set_pause_allowed([pause_allowed])

Sets whether the player can pause or unpause the game.

- `pause_allowed` (boolean, optional): `true` to allow the player to pause the game. No value means `true`.

Remarks

This function applies to the built-in pause command. Your script can still pause the game explicitly by calling game:set_paused().

2.11.3.10 game:is_dialog_enabled()

Returns whether this game is currently showing a dialog.

It does not matter whether the dialog is shown with the built-in, minimal dialog box or with your custom dialog box (see game:on_dialog_started()).

Only possible when the game is running.

- Return value (boolean): `true` if a dialog is being shown.

2.11.3.11 game:start_dialog(dialog_id, [info], [callback])

Starts showing a dialog.

A dialog must not be already active. This function returns immediately, but you can provide a callback that will be executed when the dialog finishes. The game is suspended during the dialog, like when it is paused.

If the event game:on_dialog_started() is not defined, then the engine will show a default, minimal dialog system without decoration. The user will be able to close the dialog by pressing the action command (you don't need to call game:stop_dialog()).

On the contrary, if the event game:on_dialog_started() is defined, the engine calls it and does nothing else. This is the recommended way, because you can make your custom dialog box implementation with any feature you need. The game will be suspended until you call game:stop_dialog() explicitly.

- `dialog_id` (string): Id of the dialog to show. The corresponding dialog must exist in the dialogs.dat file of the current language.
- `info` (any type, optional): Any information you want to pass to the game:on_dialog_started() event. You can use this parameter to include in the dialog any information that is only known at runtime, for example the name of the player, the best score of a mini-game or the time spent so far in the game. See the examples below.
- `callback` (function, optional): A function to be called when the dialog finishes. A status parameter (possibly `nil`) is passed to your function: its value is the argument of game:stop_dialog() and it represents the result of the dialog. This feature may be used by your dialog box system to return any useful information to the map script, like the answer chosen by the player if the dialog was a question, or whether the dialog was skipped.

Example of a small map script with an NPC that shows a simple dialog:

```
local map = ...

function some_npc:on_interaction()
  -- Remember that you specify a dialog id, not directly the text to show.
  -- The text is defined in the dialogs.dat file of the current language.
  map:get_game():start_dialog("welcome_to_my_house")
end
```

Here is a more complex example, with an NPC that asks a question. This example assumes that your dialog box system can ask questions to the player, and returns the answer as a boolean value passed to game:stop_dialog().

```
local map = ...
local game = map:get_game()
function another_npc:on_interaction()
  game:start_dialog("give_me_100_rupees_please", function(answer)
    if answer then
      if game:get_money() >= 100 then
        game:remove_money(100)
        game:start_dialog("thanks")
      else
        sol.audio.play_sound("wrong")
        game:start_dialog("not_enough_money")
      end
    else
      game:start_dialog("not_happy")
    end
  end)
end
```

Finally, to illustrate the use of the `info` parameter, let's modify the previous example to make the amount of money only determined at runtime. In other words, we want an NPC that can say "Please give me 100 rupees", but also "Please give me 25 rupees" or any number. Since the number is only known at runtime, it can no longer be hardcoded in the text of the dialog. So let's assume that the text of the dialog contains instead a special sequence (like `"$v"`) to be substituted by the final value. (Note that shop treasures use a very similar convention for their dialogs.)

```
local map = ...
local game = map:get_game()

function another_npc:on_interaction()
  local how_much = math.random(100)
  game:start_dialog("give_me_x_rupees_please", how_much, function(answer)
    if answer then
      if game:get_money() >= how_much then
        game:remove_money(how_much)
        -- ... The rest is unchanged.
```

To make this example work, you need a dialog box system that performs the substitution when `info` is set. See game:on_dialog_started().

Note

The `info` parameter of game:start_dialog() and the status parameter of the callback are a flexible way to make the map script communicate with the dialog box system in both directions. They can been seen as the parameter and the result (respectively) of the dialog being displayed. They can both be any value, like a table with many information.

2.11.3.12 game:stop_dialog([status])

Stops the current dialog.

A dialog must be active when you call this function. The dialog stops being displayed and the game can resume. This function is typically called by your dialog box system when it wants to close the dialog.

The game:on_dialog_finished() event is first called (if it is defined). Then, the callback that was passed to game:start_dialog() is called (if it was defined) with the `status` argument.

- `status` (any type, optional): Some information to return to the script that started the dialog. For example, you can pass the result of the dialog if it was a question, or whether it was skipped by the user before the end. See the examples above.

2.11.3.13 game:is_game_over_enabled()

Returns whether this game is currently showing a game-over sequence.

Only possible when the game is running.

The game-over sequence automatically starts when the player's life gets to zero, or when you call game:start_game_over() explicitly. Define the event game:on_game_over_started() to show your game-over menu. If you don't define this event, by default, there is no game-over sequence and the engine immediately restarts the game (but does not save it).

- Return value (boolean): `true` if a game-over sequence is running.

2.11.3.14 game:start_game_over()

Starts the game-over sequence manually.

Only possible when the game is running.

This function is seldom needed since the game-over sequence automatically starts when the player's life reaches zero. But you can use it if you want to start a game-over sequence even when the player's life is greater than zero.

2.11.3.15 game:stop_game_over()

Finishes the current game-over sequence.

Only possible during a game-over sequence.

The game is suspended during the whole game-over sequence. Call this function to resume it. If the life is still zero at this point, then the engine automatically restores full life.

2.11.3.16 game:get_map()

Returns the current map.

- Return value (map): The current map of this game (`nil` if this game is not running).

2.11.3.17 game:get_hero()

Returns the hero.

The hero is a map entity that always exists while the game is running, and that persists when the map changes. For this reason, he can be seen as belonging to the game more than to the current map. That's why this function exists.

- Return value (hero): The hero, or `nil` if the game is not running.

Remarks

Equivalent to `game:get_map():get_entity("hero")`.

2.11.3.18 game:get_value(savegame_variable)

Returns a value saved.

- `savegame_variable` (string): Name of the value to get from the savegame.
- Return value (string, number or boolean): The corresponding value (`nil` if no value is defined with this key).

2.11.3.19 game:set_value(savegame_variable, value)

Sets a value in the savegame.

This function allows to store key-value pairs in the savegame. Values can be strings, integers or booleans.

- `savegame_variable` (string): Name of the value to save (must contain alphanumeric characters or `'_'` only, and must start with a letter).
- `value` (string, number or boolean): The value to set, or `nil` to unset this value.

Remarks

This method changes a value, but remember that the change will be saved in the savegame file only when you call game:save().

2.11.3.20 game:get_starting_location()

Returns the location where the hero is placed when this game is started or restarted.

- Return value 1 (string): Id of the starting map. `nil` means that it was not set: in this case, the first map declared in project_db.dat will be used.
- Return value 2 (string): Name of the destination where the hero will be placed on that map. `nil` means that it was not set: in this case, the default destination entity of that map will be used.

2.11.3.21 game:set_starting_location([map_id, [destination_name]])

Sets the location where the hero should be placed when this game is started or restarted.

- `map_id` (string, optional): Id of the starting map. By default, the first map declared in project_db.dat is used.
- `destination_name` (string, optional): Name of the destination where the hero should be placed on that map. By default, the default destination of the map is used.

Remarks

When the hero moves from a map to another map that belongs to a different world (for example, from a dungeon to the outside world) using a destination entity, by default, the starting location is automatically set to this point. If this behavior is okay for your quest, you never need to call this function except the first time: when initializing a new savegame file. This behavior can be changed by setting the "Save starting location" property of destinations, from the quest editor or from a script (with destination:set_starting_location_mode()).

2.11.3.22 game:get_life()

Returns the current level of life of the player.

- Return value (number): The current life.

2.11.3.23 game:set_life(life)

Sets the level of life of the player.

A negative value will be replaced by zero. A value greater than than the maximum level of life will be replaced by the maximum value.

- `life` (number): Number of life points to set.

2.11.3.24 game:add_life(life)

Adds some life to the player.

- `life` (number): Number of life points to add. Must be a positive number or `0`.

Remarks

Equivalent to `game:set_life(game:get_life() + life)`.

2.11.3.25 game:remove_life(life)

Removes some life from the player.

- `life` (number): Number of life points to remove. Must be a positive number or `0`.

Remarks

Equivalent to `game:set_life(game:get_life() - life)`.

2.11.3.26 game:get_max_life()

Returns the maximum level of life of the player.

- Return value (number): The maximum number of life points.

2.11.3.27 game:set_max_life(life)

Sets the maximum level of life of the player.

- `life` (number): Maximum number of life points to set. Must be a positive number.

2.11.3.28 game:add_max_life(life)

Increases the maximum level of life of the player.

- `life` (number): Maximum number of life points to add to the maximum. Must be a positive number or `0`.

Remarks

Equivalent to `game:set_max_life(game:get_max_life() + life)`.

2.11.3.29 game:get_money()

Returns the amount of money of the player.

- Return value (number): The current amount of money.

2.11.3.30 game:set_money(money)

Sets the amount of money of the player.

A negative value will be replaced by zero. A value greater than than the maximum amount of money will be replaced by the maximum amount.

- `money` (number): The amount of money to set.

2.11.3.31 game:add_money(money)

Adds some money to the player.

- `money` (number): Amount of money to add. Must be a positive number or `0`.

Remarks

Equivalent to `game:set_money(game:get_money() + money)`.

2.11.3.32 game:remove_money(money)

Removes some money from the player.

- `money` (number): Amount of money to remove. Must be a positive number or `0`.

Remarks

Equivalent to `game:set_money(game:get_money() - money)`.

2.11.3.33 game:get_max_money()

Returns the maximum amount of money of the player.

- Return value (number): The maximum money.

2.11.3.34 game:set_max_money(money)

Sets the maximum amount of money of the player.

- `money` (number): Maximum money to set. Must be a positive number or `0`.

2.11.3.35 game:get_magic()

Returns the current number of magic points.

- Return value (number): The current number of magic points.

2.11.3.36 game:set_magic(magic)

Sets the amount of magic points of the player.

A negative value will be replaced by zero. A value greater than than the maximum number of magic points will be replaced by that maximum.

- `magic` (number): The number of magic points to set.

2.11.3.37 game:add_magic(magic)

Adds some magic points to the player.

- `magic` (number): Number of magic points to add. Must be a positive number or `0`.

Remarks

Equivalent to `game:set_magic(game:get_magic() + magic)`.

2.11.3.38 game:remove_magic(magic)

Removes some magic points from the player.

- `magic` (number): Number of magic points to remove. Must be a positive number or `0`.

Remarks

Equivalent to `game:set_magic(game:get_magic() - magic)`.

2.11.3.39 game:get_max_magic()

Returns the maximum number of magic points.

- Return value (number): The maximum number of magic points.

2.11.3.40 game:set_max_magic(magic)

Sets the maximum number of magic points.

- `magic` (number): The maximum number of magic points to set. Must be a positive number or `0`.

2.11.3.41 game:has_ability(ability_name)

Returns whether the player has a built-in ability.

- `ability_name`: Name of the ability to get (see game:get_ability() for the list of valid ability names).
- Return value (boolean): `true` if the player has this ability.

Remarks

Equivalent to `game:get_ability(ability_name) > 0`.

2.11.3.42 game:get_ability(ability_name)

Returns the level of a built-in ability.

Built-in ability levels indicate whether the hero can perform some built-in actions like attacking, swimming or running.

- `ability_name` (string): Name of the ability to get. Valid ability names are:
  - `"sword"`: Ability to use the sword and with which sprite.
  - `"sword_knowledge"`: Ability to make the super spin-attack.
  - `"tunic"`: Tunic (determines the sprite used for the hero's body).
  - `"shield"`: Protection against enemies. Determines whether the hero can avoid some kinds of attacks.
  - `"lift"`: Ability to lift heavy objects.
  - `"jump_over_water"`: Automatically jump when arriving into water without the `"swim"` ability.
  - `"swim"`: Ability to swim in deep water.
  - `"run"`: Ability to run when pressing the action command.
  - `"detect_weak_walls"`: Notifies the player with a sound when a weak wall is nearby.
- Return value (number): Level of this ability (`0` means not having this ability yet).

2.11.3.43 game:set_ability(ability_name, level)

Sets the level of an ability.

- `ability_name` (string): Name of the ability to set (see game:get_ability() for the list of valid ability names).
- `level` (number): Level of this ability to set (`0` removes the ability).

2.11.3.44 game:get_item(item_name)

Returns an equipment item.

- `item_name` (string): Name of the item to get.
- Return value (item): The corresponding equipment item.

2.11.3.45 game:has_item(item_name)

Returns whether the player has the specified equipment item (only for a saved item).

- `item_name` (string): Name of the item to check.
- Return value (boolean): `true` if the player has at least the first variant of this item.

Remarks

Equivalent to `game:get_item(item_name):get_variant() > 0`.

2.11.3.46 game:get_item_assigned(slot)

Returns the equipment item assigned to a slot.

- `slot` (number): The slot to get (`1` or `2`).
- Return value (item): The equipment item associated to this slot (`nil` means none).

2.11.3.47 game:set_item_assigned(slot, item)

Assigns an equipment item to a slot.

- `slot` (number): The slot to set (`1` or `2`).
- `item` (item): The equipment item to associate to this slot, or `nil` to make the slot empty.

2.11.3.48 game:get_command_effect(command)

Returns the current built-in effect of a game command.

This function is useful if you want to show a HUD that indicates to the player the current effect of pressing a game command, especially for command `"action"` whose effect changes a lot depending on the context.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- Return value (string): A string describing the current built-in effect of this game command. `nil` means that this command has currently no built-in effect (for example because the game is paused). Possible values are:
  - For command `"action"`: `"next"`, `"look"`, `"open"`, `"lift"`, `"throw"`, `"grab"`, `"speak"`, `"swim"`, `"run"` or `nil`.
  - For command `"attack"`: `"sword"` or `nil`.
  - For command `"pause"`: `"pause"`, `"return"` or `nil`.
  - For command `"item_1"`: `"use_item_1"` or `nil`.
  - For command `"item_2"`: `"use_item_2"` or `nil`.
  - For command `"right"`: `"move_right"` or `nil`.
  - For command `"left"`: `"move_left"` or `nil`.
  - For command `"up"`: `"move_up"` or `nil`.
  - For command `"down"`: `"move_down"` or `nil`.

Remarks

All these built-in game commands are initially mapped to some default keyboard and joypad inputs. You can use game:set_command_keyboard_binding(), game:set_command_joypad_binding() and game:capture_↩ command_binding() to change or even disable these mappings.
It is also possible to override the behavior of game commands by intercepting the events game:on_↩ command_pressed() game:on_command_released().

2.11.3.49 game:get_command_keyboard_binding(command)

Returns the keyboard key that triggers the specified game command.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- Return value (string): Name of the keyboard key that triggers this game command, or `nil` if no keyboard key is mapped to this game command.

2.11.3.50 game:set_command_keyboard_binding(command, key)

Sets the keyboard key that triggers a game command.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- key (string): Name of the keyboard key that should trigger this game command (`nil` means none).

Note

If this keyboard key was already mapped to a command, keyboard keys of both commands are switched.

2.11.3.51 game:get_command_joypad_binding(command)

Returns the joypad input that triggers the specified game command.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- Return value (string): A string describing what joypad input triggers this game command, or `nil` if no joypad input is mapped to this game command. This string can have one of the following forms:
  - `"button X"` where X is the index of a joypad button (first is `0`),
  - `"axis X +"` where X is the index of a joypad axis (first is `0`),
  - `"axis X -"` where X is the index of a joypad axis (first is `0`),
  - `"hat X Y"` where X is the index of a joypad hat (first is `0`) and Y is a direction (`0` to `7`).

2.11.3.52 game:set_command_joypad_binding(command, joypad_string)

Sets the joypad input that should trigger the specified game command.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- joypad_string (string): A string describing what joypad input should trigger this game command (`nil` means none). This string must have one of the following forms:
  - `"button X"` where X is the index of a joypad button (first is `0`),
  - `"axis X +"` where X is the index of a joypad axis (first is `0`),
  - `"axis X -"` where X is the index of a joypad axis (first is `0`),
  - `"hat X Y"` where X is the index of a joypad hat (first is `0`) and Y is a direction (`0` to `7`).

Note

If this joypad input was already mapped to a command, joypad inputs of both commands are switched.

2.11.3.53 game:capture_command_binding(command, [callback])

Makes the next keyboard or joypad input become the new binding for the specified game command.

This function returns immediately. After you call it, the next time the player presses a keyboard key or performs a joypad input, this input is treated differently: instead of being forwarded to your script or handled by the engine as usual, it automatically becomes the new keyboard or joypad binding for a game command.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- `callback` (function, optional): A function to call when the new input occurs.

Note

If the keyboard (or joypad) input was already mapped to a command, keyboard (or joypad) inputs of both commands are switched.

2.11.3.54 game:is_command_pressed(command)

Returns whether a built-in game command is currently pressed.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- Return value (boolean): `true` if this game command is currently pressed by the player.

2.11.3.55 game:get_commands_direction()

Returns the direction (in an 8-direction system) formed by the combination of directional game commands currently pressed by the player.

- Return value (number): The direction wanted by the player (`0` to 7), or `nil` for no direction. No direction means that no directional command is pressed, or that contradictory directional commands are pressed, like left and right at the same time (impossible with most joypads, but easy with a keyboard).

Remarks

This function is provided for convenience. Its result can also be computed by calling game:is_command_↩ pressed() four times (with the four directional game commands).

2.11.3.56 game:simulate_command_pressed(command)

Creates a command pressed input event.

Everything acts like if the player had just pressed an input mapped to this game command.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.

2.11.3.57 game:simulate_command_released(command)

Creates a command released input event.

Everything acts like if the player had just released an input mapped to this game command.

- `command` (string): Name of a game command. Valid commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.

2.11.4 Events of a game

Events are callback methods automatically called by the engine if you define them. In the case of a game, they are only called on the game currently running, if any.

2.11.4.1 game:on_started()

Called when this game starts running (including when you restart the same game).

2.11.4.2 game:on_finished()

Called when this game stops running (including when you restart the same game).

2.11.4.3 game:on_update()

Called at each cycle of the main loop while this game is running.

Remarks

As this function is called at each cycle, it is recommended to use other solutions when possible, like timers and other events.

2.11.4.4 game:on_draw(dst_surface)

Called when the game has just been redrawn by the engine.

The engine has already drawn the current map, since the map is always drawn before the game. If the game has menus, these menu are not drawn yet at this point. Use this event if you want to draw some additional content before the menus.

- `dst_surface` (surface): The surface where the game is drawn.

2.11.4.5 game:on_map_changed(map)

Called when the player has just entered a map.

The new map is already started at this point. For example, you may use this event if some parts of your HUD needs to be changed on particular maps.

- `map` (map): The new active map.

Remarks

This event is also called for the first map (when your game starts).

2.11.4.6 game:on_paused()

Called when the game has just been paused.

The game may have been paused by the player (by pressing the `"pause"` game command) or by you (by calling game:set_paused(true)).

This function is typically the place where you should start your pause menu.

2.11.4.7 game:on_unpaused()

Called when the game is being resumed.

The game may have been unpaused by the player (by pressing the "pause" game command) or by you (by calling game:set_paused(false)).

This is probably a good place to stop your pause menu.

2.11.4.8 game:on_dialog_started(dialog, [info])

Called when a dialog starts.

The dialog may be triggered by a Lua script (by calling game:start_dialog()) or by the engine in various situations (for example when finding a treasure).

If this event is not defined, the engine shows a minimal dialog box without decoration and you have nothing else to do.

If this event is defined, the engine does nothing and your script is responsible to show the dialog in any way you want, and to close it later by calling game:stop_dialog(). It is recommended to implement your dialog system as a menu: if you do so, you will automatically get called by the engine when a command is pressed, when you need to draw the dialog box on the screen, etc.

- dialog (table): All properties of the dialog to show. This table is identical to the one returned by sol.language.get_dialog(). It is a table with at least the following two entries:
  - dialog_id (string): Id of the dialog.
  - text (string): Text of the dialog in the current language. It may have several lines. When it is not empty, it always ends with a newline character.

  The table also contains all custom entries defined in text/dialogs.dat for this dialog. These custom entries always have string keys and string values. Values that were defined as numbers in "text/dialogs.dat" are replaced in this table by their string representation, and values that were defined as booleans are replaced by the string "1" for true and "0" for false.
- info (any value, optional): Some additional information for this particular dialog. You can get here some data that is only known at runtime. See the examples of game:start_dialog().

2.11.4.9 game:on_dialog_finished(dialog)

Called when the current dialog stops.

- dialog (table): All properties of the dialog that was shown. See game:on_dialog_started() for a description of this table.

2.11.4.10 game:on_game_over_started()

Called when a game-over sequence starts.

This event is called when the player's life reaches zero, as soon as the hero is in a state that allows game-over. It is also called if you started a game-over sequence manually with game:start_game_over().

If this event is not defined, there is no game-over sequence: the game restarts immediately, like if you called game:start(), and the full life of the player is restored.

If this event is defined, the engine does nothing except suspending the game. Your script is then responsible to show a game-over sequence in any way you want, and to call game:stop_game_over() once you have finished.

For instance, you may create a dialog that lets the player restart the game or save and quit, or a menu with more options.

Actually, it is not even required to restart or quit the game after your game-over sequence (even if this is the most common case). Indeed, you can also just resume the game. In this case, the game continues normally like if nothing happened.

2.11.4.11 game:on_game_over_finished()

Called when the current game-over sequence stops.

This event is also called if you did not define a game-over sequence.

2.11.4.12 game:on_key_pressed(key, modifiers)

Called when the user presses a keyboard key while your game is running.

- `key` (string): Name of the raw key that was pressed.
- `modifiers` (table): A table whose keys indicate what modifiers were down during the event. Possible table keys are `"shift"`, `"control"` and `"alt"`. Table values don't matter.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects. If you return `false` or nothing, the event will continue its propagation in this order: to the game menus if any, to the current map (including its own menus if any), and then to the game commands.

If you handle the event, you should return `true` to make the event stop being propagated. The menus of your game (if any) and the current map won't be not notified in this case. On the contrary, if neither your game, its menus nor the current map handle the event, then the engine handles it with a built-in behavior. This built-in behavior is to check whether a game command is mapped to the keyboard key that was pressed. If yes, the keyboard pressed event will be transformed into a game command pressed event (see game:on_command_pressed()).

Remarks

This event indicates the raw keyboard key pressed. If you want the corresponding character instead (if any), see game:on_character_pressed(). If you want the corresponding higher-level game command (if any), see game:on_command_pressed().

2.11.4.13 game:on_key_released(key, modifiers)

Called when the user releases a keyboard key while your game is running.

- `key` (string): Name of the raw key that was released.
- `modifiers` (table): A table whose keys indicate what modifiers were down during the event. Possible table keys are `"shift"`, `"control"` and `"alt"`. Table values don't matter.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects. If you return `false` or nothing, the event will continue its propagation in this order: to the game menus if any, to the current map (including its own menus if any), and then to the game commands.

If you handle the event, you should return `true` to make the event stop being propagated. The menus of your game (if any) and the current map won't be not notified in this case. On the contrary, if neither your game, its menus nor the current map handle the event, then the engine handles it with a built-in behavior. This built-in behavior is to check whether a game command is mapped to the keyboard key that was released. If yes, the "keyboard released" event will be transformed into a "game command released" event (see game:on_command_released()).

2.11.4.14 game:on_character_pressed(character)

Called when the user enters text while your game is running.

- `character` (string): A utf-8 string representing the character that was entered.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects. If you return `false` or nothing, the event will continue its propagation in this order: to the game menus if any and then to the current map (including its own menus if any).

Remarks

When a character key is pressed, two events are called: game:on_key_pressed() (indicating the raw key) and game:on_character_pressed() (indicating the utf-8 character). If your game needs to input text from the user, game:on_character_pressed() is what you want because it considers the keyboard's layout and gives you international utf-8 strings.

2.11.4.15 game:on_joypad_button_pressed(button)

Called when the user presses a joypad button while your game is running.

- `button` (number): Index of the button that was pressed.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.11.4.16 game:on_joypad_button_released(button)

Called when the user releases a joypad button while your game is running.

- `button` (number): Index of the button that was released.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.11.4.17 game:on_joypad_axis_moved(axis, state)

Called when the user moves a joypad axis while your game is running.

- `axis` (number): Index of the axis that was moved. Usually, `0` is an horizontal axis and `1` is a vertical axis.
- `state` (number): The new state of the axis that was moved. `-1` means left or up, `0` means centered and `1` means right or down.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.11.4.18 game:on_joypad_hat_moved(hat, direction8)

Called when the user moves a joypad hat while your game is running.

- `hat` (number): Index of the hat that was moved.
- `direction8` (number): The new direction of the hat. `-1` means that the hat is centered. `0` to `7` indicates that the hat is in one of the eight main directions.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.11.4.19 game:on_command_pressed(command)

Called when the player presses a game command (a keyboard key or a joypad action mapped to a built-in game behavior) while this game is running. You can use this event to override the normal built-in behavior of the game command.

- `command` (string): Name of the built-in game command that was pressed. Possible commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (you are overriding the built-in behavior of pressing this game command).

Remarks

This event is not triggered if you already handled its underlying low-level keyboard or joypad event.

2.11.4.20 game:on_command_released(command)

Called when the player released a game command (a keyboard key or a joypad action mapped to a built-in game behavior). while this game is running. You can use this event to override the normal built-in behavior of the game command.

- `command` (string): Name of the built-in game command that was released. Possible commands are
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (you are overriding the built-in behavior of releasing this game command).

Remarks

This event is not triggered if you already handled its underlying low-level keyboard or joypad event.

2.11.4.21 game:on_mouse_pressed(button, x, y)

Called when the user presses a mouse button while the game is running.

- `button` (string): Name of the mouse button that was pressed. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- `x` (integer): The x position of the mouse in quest size coordinates.
- `y` (integer): The y position of the mouse in quest size coordinates.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.11.4.22 game:on_mouse_released(button, x, y)

Called when the user releases a mouse button while the game is running.

- `button` (string): Name of the mouse button that was released. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- `x` (integer): The x position of the mouse in quest size coordinates.
- `y` (integer): The y position of the mouse in quest size coordinates.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.12 Equipment items

An equipment item represents something that the player can obtain (one or more times, in various forms and with several variants) and possibly keep. The Lua item type described in this page provides various functions to

- get and the possession state of a equipment item,
- set its properties, like whether the item is saved and has an amount,
- control its dynamic behavior, like what happens when the player uses this item (for items that can be used).

A Lua item object represents a kind of treasure, and not a particular instance of treasures. Individual treasures may then be represented as pickable treasures, chests, shop treasures, and may be brandished by the hero. For example, multiple treasures of the kind `"rupee"` may exist at the same time during the game, but only one Lua item object manages them.

The script file `items/XXXX.lua` defines the item named `XXXX`. The corresponding Lua item object is passed as parameter of that script. Use the Lua notation `"..."` to get this parameter and store it into a regular variable.

Here is a basic example of script for the `rupee` item, an item whose only role is to increase the money of the player when he obtains a rupee.

```
-- First, we put the parameter into a variable called "rupee".
-- (In Lua, the notation "..." refers to the parameter(s) of the script.)
local rupee = ...

-- Event called when the hero obtains a rupee treasure.
function rupee:on_obtaining()
  self:get_game():add_money(1)  -- Increase the money of 1 unit.
end
```

Remarks

All item scripts are loaded when you create a savegame object. Indeed, equipment items only exist in the context of a particular savegame. As shown in the example above, you can retrieve that savegame with item:get_game().
A sprite animation named `XXXX` must also exist in the sprite `entities/items`: it is used by the engine whenever it needs to draw your item on the map (for example, when a pickable treasure of this kind is created on the map).

2.12.1 Methods of the type item

2.12.1.1 item:get_name()

Returns the name of this item.

- Return value (string): The name that identifies this item. It is also the file name of the item script (without the extension).

2.12.1.2 item:get_game()

Returns the game where this item belongs.

- Return value (game): The game that contains this item.

Remarks

Items only exist in the context of a game, but the game is not necessarily running.

2.12.1.3 item:get_map()

Returns the current map.

- Return value (map): The current map, or `nil` if the game is not running.

2.12.1.4 item:get_savegame_variable()

Returns the name of the integer savegame value that stores the possession state of this item.

- Return value (string): The savegame variable that stores the possessed variant of this item, or `nil` if this item is not saved.

2.12.1.5 item:set_savegame_variable(savegame_variable)

Sets the name of the integer savegame value that stores the possession state of this item.

You should call this function at initialization time if you want your item to be saved.

- `savegame_variable` (string): The savegame variable that should stored the possessed variant of this item, or `nil` to make this item unsaved.

2.12.1.6 item:get_amount_savegame_variable()

Returns the name of the integer savegame value that stores the amount associated to this item.

- Return value (string): The savegame variable that stores the possessed amount of this item, or `nil` if this item has no associated amount.

2.12.1.7 item:set_amount_savegame_variable()

Returns the name of the integer savegame value that stores the amount of this item.

You should call this function at initialization time if you want your item to store an amount (in addition to its possessed variant). This is typically used for items like the bow and the counter of bombs.

- Return value (string): The savegame variable that should store the possessed amount of this item, or `nil` to make this item have no associated amount.

2.12.1.8 item:is_obtainable()

Returns whether the player is allowed to obtain this item.

If not, any treasure representing this item is automatically replaced by an empty treasure.

- Return value (boolean): `true` if this item is obtainable.

2.12.1.9 item:set_obtainable([obtainable])

Sets whether the player is allowed to obtain this item.

If not, any treasure representing this item is automatically replaced by an empty treasure. There is no risk that the player can obtain it or even see it during the game. You can use this feature to hide some items while the player has not the necessary equipment. For example, you can make arrows unobtainable until the player has the bow. You can also make magic jars unobtainable until the player has a magic bar.

- Return value (boolean, optional): `true` if this item is obtainable (no value means `true`).

2.12.1.10 item:is_assignable()

Returns whether this item can be assigned to an item slot.

When the item is assigned to a slot, the player can use it by pressing the game command of that slot. Some items are meant to be used by pressing a command (like the bow), other are not supposed to (like a key or a rupee). When the player uses your item, the event item:on_using() is triggered.

- Return value (boolean): `true` if this item is assignable.

2.12.1.11 item:set_assignable([assignable])

Sets whether this item should be assignable to an item slot.

When the item is assigned to a slot, the player can use it by pressing the game command of this slot. Some items are meant to be used by pressing a command (like the bow), other are not supposed to (like a key or a rupee). When the player uses your item, the event item:on_using() is triggered.

By default, an item is not assignable. Call this function at initialization time if you want your item to be assignable.

- `assignable` (boolean, optional): `true` if this item is assignable (no value means `true`).

2.12.1.12 item:get_can_disappear()

Returns whether pickable treasures of this kind disappears after a few seconds when they are dropped by an enemy or a destructible entity.

- Return value (boolean): `true` if pickable treasures of this kind can disappear.

2.12.1.13 item:set_can_disappear([can_disappear])

Sets whether pickable treasures of this kind should disappear after a few seconds when they are dropped by an enemy or a destructible entity.

By default, an item cannot disappear. Call this function at initialization time if you want your item to be ephemeral.

- `can_disappear` (boolean, optional): `true` to make such pickable treasures disappear after a few seconds (no value means `true`).

Remarks

This property only applies to pickable treasures dropped dynamically (by enemies and destructible entities). Pickable treasures already present on the map when the map starts don't disappear with time.

2.12.1.14 item:get_brandish_when_picked()

Returns whether the hero brandishes treasures of this kind when he picks them on the ground.

- Return value (boolean): `true` if the hero brandish such treasures.

2.12.1.15 item:set_brandish_when_picked([brandish_when_picked])

Sets whether the hero should brandish treasures of this kind when he picks them on the ground.

Treasures coming from a chest are always brandished, even the most basic ones like simple rupees. However, when treasures are picked on the ground (like rupees dropped by an enemy), you may want the hero not to brandish them.

By default, this property is `true`. Call this function if you don't want your item to be brandished when it is picked on the ground.

- `brandish_when_picked` (boolean, optional): `true` if the hero should brandish such treasures (no value means `true`).

2.12.1.16 item:get_shadow()

Returns the name of the animation representing the shadow of this item in the sprite `"entities/shadow"`.

- Return value (string): Name of the shadow animation adapted to this item in the sprite `"entities/shadow"`. `nil` means no shadow displayed.

2.12.1.17 item:set_shadow(shadow_animation)

Sets the name of the animation that should represent the shadow of this item in the sprite `"entities/shadow"`.

When the engine needs to show a treasure representing your item, it sometimes also wants to display a shadow (in addition of the treasure's main sprite). For example, pickable treasures dropped by enemies normally have a shadow.

The default shadow animation is `"big"`. You should call this function at initialization time if your item sprite is larger or smaller than usual.

- `shadow_animation` (string): Name of the shadow animation in the sprite `"entities/shadow"` to set for this item. `nil` means that no shadow will be displayed.

Remarks

To draw a treasure, the engine relies on two sprites: the treasure's main sprite (`entities/item`) and the shadow's sprite (`entities/shadow`). Both sprites and their appropriate animations must exist so that treasures can be displayed correctly.

2.12.1.18 item:get_sound_when_picked()

Returns the sound played when the hero picks a treasure of this kind.

- Return value (string): Name of the sound played when the hero picks a treasure of this kind (`nil` means no sound).

2.12.1.19 item:set_sound_when_picked(sound_when_picked)

Sets the sound to play when the hero picks a treasure of this kind.

The default sound is `"picked_item"`.

- `sound_when_picked` (string): Name of the sound to play (as in sol.audio.play_sound()) when the hero picks a treasure of this kind (`nil` means no sound).

Remarks

This sound is always played, even if the treasure is also brandished then (i.e. if item:get_brandish_when_↩ picked() returns `true`).

2.12.1.20 item:get_sound_when_brandished()

Returns the sound played when the hero brandishes a treasure of this kind.

- Return value (string): Name of the sound played when the hero brandishes a treasure of this kind (`nil` means no sound).

2.12.1.21 item:set_sound_when_brandished(sound_when_brandished)

Sets the sound to play when the hero brandishes a treasure of this kind.

The hero can brandish treasures in various situations: when opening a chest, when buying a shop treasure, when picking up a pickable treasure (unless you called item:set_brandish_when_picked(false)), and also when you call hero:start_treasure() directly.

The default sound is `"treasure"`.

- `sound_when_brandished` (string): Name of the sound to play (as in sol.audio.play_sound()) when the hero brandishes a treasure of this kind (`nil` means no sound).

2.12.1.22 item:has_variant([variant])

Returns whether the player owns at least the specified variant of this item (only for a saved item).

- `variant` (number, optional): The variant to check (default `1`).
- Return value (boolean): `true` if the player has at least this variant.

Remarks

Equivalent to `item:get_variant() >= variant`.

2.12.1.23 item:get_variant()

Returns the possession state of this item (only for a saved item).

- Return value (number): The possession state (`0`: not possessed, `1`: first variant, `2`: second variant, etc.).

Remarks

Equivalent to `item:get_game():get_value(item:get_savegame_variable())`.

2.12.1.24 item:set_variant(variant)

Sets the possession state of this item (only for a saved item).

- `variant` (number): The new possession state to set (`0`: not possessed, `1`: first variant, `2`: second variant, etc.)

Remarks

Equivalent to `item:get_game():set_value(item:get_savegame_variable(), variant)`.

2.12.1.25 item:has_amount([amount])

Returns whether this item has an associated amount, or whether the player has at least the specified value.

- `amount` (number, optional): The amount to check. If this parameter is not set, this function only tests whether an amount value exists for the item.
- Return value (boolean): If an amount is specified, return `true` if the player has at least that amount. Otherwise, returns `true` if the item has an amount value.

Remarks

If an amount is specified, this method is equivalent to `item:get_amount() >= amount`. Otherwise, this method is equivalent to `item:get_amount_savegame_variable() ~= nil`.

2.12.1.26 item:get_amount()

Returns the amount associated to this item (only for an item with an amount value).

- Return value (number): The associated amount.

Remarks

Equivalent to `item:get_game():get_value(item:get_amount_savegame_variable())`.

2.12.1.27 item:set_amount(amount)

Sets the amount associated to this item (only for an item with an amount value). A negative amount will be replaced by `0`. An amount greater than item:get_max_amount() will be replaced by that maximum value.

- `amount` (number): The amount to set.

2.12.1.28 item:add_amount(amount)

Increases the amount associated to this item (only for an item with an amount value), without exceeding the maximum amount.

- `amount` (number): The amount to add.

2.12.1.29 item:remove_amount(amount)

Decreases the amount associated to this item (only for an item with an amount value), without going below `0`.

- `amount` (number): The amount to remove.

2.12.1.30 item:get_max_amount()

Returns the maximum amount associated to this item (only for an item with an amount value).

- Return value (number): The maximum amount.

2.12.1.31 item:set_max_amount(max_amount)

Sets the maximum amount associated to this item (only for an item with an amount value). This maximum value is used in item:set_amount() and item:add_amount() to make a limit. The default value is `1000`.

- `max_amount` (number): The maximum amount to set.

Remarks

The maximum amount of an item is not saved automatically. Only the current variant (see item:set_savegame_variable()) and the current amount (see item:set_amount_savegame_variable()) are saved by the engine. Therefore, you have to set the maximum amount of appropriate items when they are loaded (typically from event item:on_started()).

2.12.1.32 item:set_finished()

Notifies the engine that using this item is finished and that the hero can get back to a normal state.

When the player uses this item (by pressing an item game command), your item script takes full control of the hero (event item:on_using() is called) and you have to program the item's behavior. When it is finished, call this function to restore normal control to the player.

This method should only be called when the hero is using this item.

2.12.2 Events of an item

Events are callback methods automatically called by the engine if you define them. In the case of a game, they are only called on the game currently running, if any.

2.12.2.1 item:on_created()

Called when your item script is being created, that is, when the game object is being created.

In this event, you should initialize your item.

Remarks

All items scripts are not loaded yet at this time. Your item is being loaded, but you don't know the status of the other ones. Unless your item is the last one to be initialized, other items are not ready. If your item needs to get some properties from other items in its initialization, use item:on_started() instead.

2.12.2.2 item:on_started()

Called when your item starts, that is, once all items are created.

At this time, all items scripts are loaded and their on_created() events have been called.

You can use this event to make initializations that depend on other items. Such initializations could not be done earlier from item:on_created(), because all items did not exist at that time.

2.12.2.3 item:on_finished()

Called when the game stops running.

2.12.2.4 item:on_update()

Called at each cycle of the main loop while the game is running.

Remarks

As this function is called at each cycle, it is recommended to use other solutions when possible, like timers and other events.

2.12.2.5 item:on_suspended(suspended)

Called when the map is being suspended or resumed. Only possible when the game is running.

The map is suspended by the engine in a few cases, like when the game is paused or when the camera is being moved by a script. When this happens, all map entities stop moving and most sprites stop their animation. Your item may need to be notified.

- `suspended` (boolean): `true` if the map is being suspended, `false` if it is being resumed.

2.12.2.6 item:on_map_changed(map)

Called when the player has just entered a map. Only possible when the game is running.

The new map is already started at this point.

- `map` (map): The new active map.

Remarks

This event is also called for the first map (when your game starts).

2.12.2.7 item:on_pickable_created(pickable)

Called when a pickable treasure of your item's kind is created on the map. Only possible when the game is running.

If you need to do something special when a pickable treasure of this kind appears on the map, like setting a particular movement, you can use this event.

- `pickable` (pickable treasure): The pickable treasure just created.

2.12.2.8 item:on_obtaining(variant, savegame_variable)

Called when the hero is obtaining a treasure of this kind of item. Only possible when the game is running.

- `variant` (number): The variant of item (because some items may have several variants). `1` is the first variant, `2` is the second variant, etc.
- `savegame_variable` (string): Name of the boolean saved value that stores the state of the treasure being obtained. `nil` means that the treasure is not saved.

Remarks

If the treasure is brandished, this function is called immediately (before the dialog starts). If you want to do something after the treasure's dialog, see item:on_obtained().

2.12.2.9 item:on_obtained(variant, savegame_variable).

Like item:on_obtaining(), but called right after the treasure is obtained. Only possible when the game is running.

In the case of a brandished treasure, this event is called once the treasure's dialog is finished. Otherwise, it is called immediately after item:on_obtaining().

- `variant` (number): The variant of item (because some items may have several variants). `1` is the first variant, `2` is the second variant, etc.
- `savegame_variable` (string): Name of the boolean saved value that stores the state of the treasure just obtained. `nil` means that the treasure is not saved.

2.12.2.10 item:on_variant_changed(variant)

Called when the possessed variant of this item has just changed. Only for saved items.

This event is triggered whenever the variant changes: it can be when the hero obtains a treasure of this item's kind, or when you call item:set_variant() explicitly.

- `variant` (number): The new variant possessed (possibly `0`).

2.12.2.11 item:on_amount_changed(amount)

Called when the possessed amount of this item has just changed. Only for items with an associated amount.

This event is triggered when you call item:set_amount(), item:add_amount(), item:remove_amount(),

- `amount` (number): The new amount possessed (possibly `0`).

2.12.2.12 item:on_using()

Called when the player is using this item. Only possible when the game is running, and only for assignable items.

The player is using your item (by pressing an item game command). You now have full control of the hero. From this event, you have to program the item's behavior. For example, your item can remove some magic points and perform an special attack that kills all enemies nearby. When you have finished, call item:set_finished() to restore normal control to the player.

Remarks

There is another event for the special case of giving an item to an non-playing character. If the hero uses an item in front of an NPC whose property is to notify your item, then event item:on_npc_interaction_item() is triggered first. If that event is defined and returns `true`, then item:on_using() is not called (the interaction is considered done).

2.12.2.13 item:on_ability_used(ability_name)

Called when the player has just performed a built-in ability. Only possible when the game is running.

Built-in abilities indicate whether the hero can perform some built-in actions like attacking, swimming or running. See game:get_ability() for more details.

- `ability_name` (string): Name of the ability that was used.

2.12.2.14 item:on_npc_interaction(npc)

Called when the hero interacts (the player pressed the action command) in front of an NPC whose property is to notify your item. Only possible when the game is running.

- `npc` (NPC): A non-playing character.

2.12.2.15 item:on_npc_interaction_item(npc, item_used)

Called when the hero uses any item (the player pressed an item command) with an NPC whose property is to notify your item. Only possible when the game is running.

- `npc` (NPC): A non-playing character.
- `item_used` (item): The item currently used by the player. This is not necessarily your item. The reason why your item gets notified is because the NPC is related to your item. But the player may be using another item.
- Return value (boolean): `true` if an interaction happened. If you return `false` or nothing, then item_used:on_using() will be called (just like if there was no NPC in front of the hero).

2.12.2.16 item:on_npc_collision_fire(npc)

Called when an NPC whose property is to notify your item touches fire. Only possible when the game is running.

- `npc` (npc): A non-playing character that touches fire and has the property to notify your item.

2.13 Map

Maps are areas where the game takes place. They may be rooms, houses, entire dungeon floors, parts of the outside world or any place. The active map contains many objects called map entities (or just "entities" to be short). Map entities include everything that has a position on the map: the hero, the tiles, the enemies, the pickable treasures, etc. See the entity API for more details.

2.13.1 Overview

2.13.1.1 Coordinates and layer

A map has a rectangular size in pixels. The width and the height are always multiples of 8 pixels, and most map entities usually stay aligned on a grid of squares of 8x8 pixels (except when they are moving).

Thus, each entity has some coordinates `X, Y` on the map. But its position is also defined by a third value: its layer. The map has a number of distinct layers that are stacked. Each layer has its own set of entities. Layers are identified by a number. There is always a layer `0`, and maps can add additional layers above or below `0`.

Layers allow to implement maps with multi-level content, like a bridge between two higher platforms. The hero (as well as enemies and any other map entity) is then able to walk either above or under the bridge, depending on his layer. Entities like stairs and jumpers can change the layer of the hero automatically, and you can also do that from your Lua scripts.

2.13.1.2 Tileset

The graphic skin of a map is called a tileset. The tileset defines the small patterns used to draw tiles and also some other entities that may depend on the skin. For example, you can have a forest tileset, a castle tileset, etc. Each map has only one tileset, but tilesets have no special size limitation.

2.13.1.3 Map files

A map can contain many types of entities. Map entities can be either declared in the map data file or created dynamically. Thus, a map with id `XXXX` is managed by two distinct files:

- In the data file maps/XXXX.dat are declared all entities that initially compose your map (most importantly: the tiles). You normally don't need to edit this file by hand: you can use the quest editor (but if you want to, here is the syntax).
- In the script file `"maps/XXXX.lua"` (which is optional), you define the dynamic behavior of your map, using the features of the map Lua type described on this page (and the rest of the Solarus API will also be very useful).

When the player enters a map, the engine first creates the entities declared in the map data file maps/XXXX.dat, and then it runs your script "maps/XXXX.lua". The Lua map object is passed as parameter of your script (remember that any Lua script is implicitly a function and can have parameters). Use the Lua notation "..." to get this parameter and store it into a regular variable.

Here is a basic example of script for a map that does nothing special except playing a special music and showing an evil welcome dialog when the hero enters it.

```
-- First, we put the parameter into a variable called "my_map".
-- (In Lua, the notation "..." refers to the parameter(s) of the script.)
local my_map = ...

-- Event called when the player enters the map, at initialization time.
function my_map:on_started()

  -- Play an evil music.
  sol.audio.play_music("evil")
end

-- Event called when the player enters the map, after the opening transition.
function my_map:on_opening_transition_finished()

  -- Show an evil welcome dialog.
  my_map:get_game():start_dialog("welcome_to_darkness")
end
```

In practice, many maps have short scripts like this or even no script at all. Sometimes, everything that you define in the data file (with the help of the quest editor) is enough. You need a script when there is something dynamic to program on your map: opening a door when a pressure plate is pressed, making a treasure chest appear when some enemies are killed, showing a dialog with an NPC that depends on whether the player has accomplished a particular action before, etc.

2.13.1.4 Lifetime of maps

At any moment, during the game, only one map is active: the one drawn on the game screen. That map is also the one where the hero is currently located. Only the active map has its entities living in it. Other maps are not loaded. If the player leaves the map and comes back later, your Lua map object will be a new one the second time. Your entities will be reloaded as specified in the map data file and your map script will be executed again.

If you want some data or behavior to persist when the player comes back, like the position of an NPC or whether a puzzle was solved, you can store the information into the savegame. Alternatively, if you don't want these data to be saved, but you want to remember then during the current game session, you can store the information as a field of the game object. The game object persists until the player stops playing and comes back to the title screen or restarts the game.

Remarks

Some entities save their state automatically if you want, like whether a treasure chest is open, whether a door is open or whether an enemy is killed. Therefore, you have nothing to do for such entities: they keep their state.

2.13.1.5 Accessing maps like tables

Like other essential Solarus types (including game, items and map entities), a fundamental property of maps is that even if they are userdata, they can also be used like Lua tables.

This property is actually what allows you to define callbacks on your map. But you can store any data in your map object, including new functions specific to your particular map. Here is an example that uses this feature:

```
-- Example of a map with 3 switches to activate in the correct order.
-- We assume that 3 switches exist with names "puzzle_switch_1", "puzzle_switch_2" and "puzzle_switch_3".
local map = ...

function map:on_started()
  map.next_switch_index = 1
  -- Equivalent to: map["next_switch_index"] = 1
end

-- Called when the player walks on the switch named "puzzle_switch_1".
function puzzle_switch_1:on_activated()
  -- Check that puzzle_switch_1 is the correct switch to activate.
  map:check_switch(self)
end

function puzzle_switch_2:on_activated()
  map:check_switch(self)
end

function puzzle_switch_3:on_activated()
  map:check_switch(self)
end

function map:check_switch(switch)
  if switch:get_name() == "puzzle_switch_" .. map.next_switch_index then
    -- Okay so far.
    map.next_switch_index = map.next_switch_index + 1
    if map.next_switch_index > 3 then
      -- Finished!
      sol.audio.play_sound("secret")
      map:get_game():start_dialog("congratulations")
    end
  else
    -- Wrong switch: reset the puzzle.
    sol.audio.play_sound("wrong")
    puzzle_switch_1:set_activated(false)
    puzzle_switch_2:set_activated(false)
    puzzle_switch_3:set_activated(false)
    map.next_switch_index = 1
  end
end
```

In this example, you see that we add three values to the map object, like it was a table: `next_switch_index` (a number), `on_started` (a function) and `check_switch` (a function). Actually, we also use this feature on the three switches: we add a value `on_activated` on them.

Remarks

You may wonder how we can access `puzzle_switch_1`, `puzzle_switch_2` and `puzzle_switch_3` without declaring them. There is a mechanism that makes all named map entities directly accessible in the environment of the map script. See map:get_entity() for more details.

But this example has some issues. The three switches are managed by duplicated code. This is error-prone because one day, we will probably want to make a similar puzzle with 50 entities instead of just 3. To make your map easier to maintain, the following and equivalent version is much more preferred:

```
local map = ...

function map:on_started()
  map.next_switch_index = 1
  map.nb_switches = map:get_entities_count("puzzle_switch_")
end

local function puzzle_switch_activated(switch)
  map:check_switch(switch)
end

-- Define the same on_activated() method for all entities whose name
```

```
-- starts with "puzzle_switch_".
for switch in map:get_entities("puzzle_switch_") do
  switch.on_activated = puzzle_switch_activated
end

function map:check_switch(switch)
  if switch:get_name() == "puzzle_switch_" .. map.next_switch_index then
    -- Okay so far.
    map.next_switch_index = map.next_switch_index + 1
    if map.next_switch_index > map.nb_switches then
      -- Finished!
      sol.audio.play_sound("secret")
      map:start_dialog("congratulations")
    end
  else
    -- Wrong switch: reset the puzzle.
    sol.audio.play_sound("wrong")
    for s in map:get_entities("puzzle_switch_") do
      s:set_activated(false)
    end
    map.next_switch_index = 1
  end
end
```

This improved version is more evolutive: it does not even hardcode the number of switches in the puzzle. Thus, if you add a switch called `"puzzle_switch_4"` from the editor one day, the script will be directly take it into account in the puzzle.

2.13.2 Methods of the type map

2.13.2.1 map:get_id()

Returns the id of this map.

- Return value (string): Id of the map.

Remarks

This id appears in the name of map files.

2.13.2.2 map:get_game()

Returns the current game.

- Return value (game): The game that is currently running the map.

2.13.2.3 map:get_world()

Returns the world name that was set on this map.

The world name is an optional property defined in the map data file. Worlds allow to group maps together. The world can be any arbitrary name. Maps that have the same world name are considered to be part of the same environment. For example, your map can be in a world named `"outside_world"`, `"dungeon_1"` or `"some_scary_cave"`. A map that has no world is always considered to be alone in its own environment.

The world property is used to decide when to set the starting location of the player (the place where he starts when loading his savegame). By default, the starting location is automatically set by the engine when the world changes (not when the map changes). This can be changed by defining the "Save starting location" property of destinations, from the quest editor or from a script (with destination:set_starting_location_mode()).

Some other features may also rely on the world property, like the state of crystal blocks. Their state persists between all maps of the current world and is reset when entering a map whose world is different.

- Return value (string): Name of the world of the current map. `nil` means no world.

2.13.2.4 map:set_world(world)

Changes the world of this map.

The world property remains until the map is destroyed: If you reload the same map again later, the world is reset to the one defined in the map data file.

- `world` (string): The new world name to set, or `nil` to set no world.

2.13.2.5 map:get_floor()

Returns the floor of the current map if any.

The floor is an optional property defined in the map data file.

The engine does not do anything particular with this floor property. But you can use it in scripts, for example to show the current floor on the HUD when it changes or to make a minimap menu.

- Return value (number): The current floor. `0` is the first floor, `1` is the second floor, `-1` is the first basement floor, etc. `nil` means that this map is not part of a floor system.

2.13.2.6 map:set_floor(floor)

Changes the floor of this map.

The floor property remains until the map is destroyed: If you reload the same map again later, the floor is reset to the one defined in the map data file.

- `floor` (number): The new floor number to set, or `nil` to set no floor.

2.13.2.7 map:get_min_layer()

Returns the index of the lowest layer of this map.

- Return value (number): The lowest layer (0 or less).

2.13.2.8 map:get_max_layer()

Returns the index of the highest layer of this map.

- Return value (number): The highest layer (0 or more).

2.13.2.9 map:get_size()

Returns the size of this map in pixels.

- Return value 1 (number): The width in pixels (always a multiple of 8).
- Return value 2 (number): The height in pixels (always a multiple of 8).

2.13.2.10 map:get_location()

Returns the x,y location of this map in its world.

The engine uses this information to implement scrolling between two adjacent maps.

For example, you can also use this property in scripts if you want to show the position of the hero on the minimap menu of your outside world. Indeed, your outside world is probably not a single map, but it is usually composed of several adjacent maps.

- Return value 1 (number): X position of the top-left corner of this map relative to its world.
- Return value 2 (number): Y position of the top-left corner of this map relative to its world.

2.13.2.11 map:get_tileset()

Returns the name of the tileset of the current map.

- Return value (string): Id of the current tileset.

2.13.2.12 map:set_tileset(tileset_id)

Changes the tileset of the current map.

It is your responsability to make sure that the new tileset is be compatible with the previous one: every tile of the previous tileset must exist in the new one and have the exact same properties, and only the images can differ. Internally, this function actually keeps the data of the previous tileset and only loads the image of the new tileset.

- `tileset_id` (string): Id of the new tileset to set.

Remarks

If the new tileset is not compatible with the previous one, tiles will be displayed with wrong graphics.

2.13.2.13 map:get_music()

Returns the name of the music associated to this map.

This is the music to play when the map starts, as specified in the map file. It may be different from the music currently being played. To get the music currently being played, see sol.audio.get_music().

- Return value (string): Name of the music of this map, relative to the `musics` directory and without extension. It can also be the special value `"same"` if the map specifies to keep the music unchanged, or `nil` if the map specifies to play no music.

2.13.2.14 map:get_camera()

Returns the camera entity of the map.

- Return value (camera): The camera.

2.13.2.15 map:get_ground(x, y, layer)

Returns the kind of ground (terrain) of a point.

The ground is defined by tiles (and other entities that may change it like dynamic tiles) that overlap this point.

- `x` (number): X coordinate of a point of the map.
- `y` (number): Y coordinate of a point of the map.
- `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
- Return value (string): The kind of ground. The possible values are the same as the `ground` property of the tileset file: `"empty"`, `"traversable"`, `"wall"`, `"low_wall"`, `"wall_top_right"`, `"wall_top_left"`, `"wall_bottom_left"`, `"wall_bottom_right"`, `"wall_top_right_water"`, `"wall_top_left_water"`, `"wall_bottom_left_water"`, `"wall_bottom_right_water"`, `"deep_water"`, `"shallow_water"`, `"grass"`, `"hole"`, `"ice"`, `"ladder"`, `"prickles"` or `"lava"`.

2.13.2.16 map:draw_visual(drawable, x, y)

Draws a drawable object (surface, text surface or sprite) on the camera at the given map coordinates.

This function can be used as an alternative to drawable:draw() in order to draw the object relative to the map (instead of relative to the screen).

If the object to draw is a sprite, its origin point will be displayed at the given location, relative to the the upper left corner of the map.

This function should only be called during the drawing phase of the map, for example from map:on_draw() or from custom_entity:on_pre_draw().

- `drawable` (drawable): The visual object to draw on the map.
- `x` (number): X coordinate of where to draw the object, in map coordinates.
- `y` (number): Y coordinate of where to draw the object, in map coordinates.

2.13.2.17 map:draw_sprite(sprite, x, y)

Draws a sprite on the screen at the given map coordinates.

Warning

This method is deprecated since Solarus 1.5.

Use map:draw_visual() instead.

2.13.2.18 map:get_crystal_state()

Returns the configuration of crystal blocks.

- Return value (boolean): `false` initially (orange blocks lowered), `true` otherwise (blue blocks lowered).

2.13.2.19 map:set_crystal_state(state)

Sets the configuration of crystal blocks.

This state persists accross maps of the same world. It is reset when the world changes and when the savegame is reloaded.

- `state` (boolean): `false` to set the initial configuration (orange blocks lowered), `true` to the modified one (blue blocks lowered).

2.13.2.20 map:change_crystal_state()

Inverts the configuration of crystal blocks.

Remarks

Equivalent to `map:set_crystal_state(not map:get_crystal_state())`.

2.13.2.21 map:open_doors(prefix)

Opens the doors whose name starts with the specified prefix, enables or disables relative dynamic tiles accordingly and plays the `"door_open"` sound.

Opening a door may be more complex than just modifying a single door entity. Indeed, there is often a corresponding door is the adjacent room that you also want to open (that corresponding door is another entity). Name both doors with the same prefix, and you can use this function to open both of them.

Furthermore, you sometimes want dynamic tiles to be shown or hidden depending on the state of a door. When a door is open, all dynamic tiles whose prefix is the door's name followed by `_open` or `_closed` are automatically enabled or disabled, respectively.

- `prefix` (string): Prefix of the name of doors to open.

Remarks

The doors will be really closed once the opening animation of their sprite is finished. However, they immediately become obstalces.

2.13.2.22 map:close_doors(prefix)

Closes the doors whose name starts with the specified prefix, enables or disables relative dynamic tiles accordingly and plays the `"door_closed"` sound.

Closing a door may be more complex than just modifying a single door entity. Indeed, there is often a corresponding door is the adjacent room that you also want to open (that corresponding door is another entity). Name both doors with the same prefix, and you can use this function to close both of them.

Furthermore, you sometimes want dynamic tiles to be shown or hidden depending on the state of a door. When a door is closed, all dynamic tiles whose prefix is the door's name followed by `_open` or `_closed` are automatically disabled or enabled, respectively.

- `prefix` (string): Prefix of the name of doors to close.

2.13.2.23 map:set_doors_open(prefix, [open])

Like map:open_doors() or map:close_doors(), but does not play any sound or any sprite animation.

This function is intended to be called when you don't want the player to notice the change, typically when your map starts (i.e. from the map:on_started() event).

- `prefix` (string): Prefix of the name of doors to set.
- `open` (boolean, optional): `true` to open the doors, `false` to close them (no value means `true`).

2.13.2.24 map:get_entity(name)

Returns the map entity with the specified name if it exists on this map. Entity names are unique (two entities cannot exist on the map with the same name at the same time). The name is optional: some entities may have no name. In this case, you cannot access them from this function.

As a convenient feature, map entities can also be accessed directly through the environment of the map script. In other words, you can just write `bob:get_position()` as an equivalent to `map:get_entity("bob")↩ :get_position()`.

- `name` (string): Name of the map entity to get.
- Return value (entity): The corresponding entity, or `nil` if there exists no entity with this name on the map.

Remarks

Technical note for curious Lua experts: the mechanism that makes map entities directly accessible in the map script environment is lazy (it is implemented as an `__index` metamethod). Entities are imported to the Lua side only when your script requests them. If you have thousands of named entities in your map, you won't have thousands of useless objects living in Lua. Only the ones your script tries to access are imported.

2.13.2.25 map:has_entity(name)

Returns whether there currently exists a map entity with the specified name on the map.

- `name` (string): Name of the map entity to check.
- Return value (boolean): `true` if such an entity exists.

Remarks

Equivalent to `map:get_entity(name) ~= nil` (but a bit lighter because it avoids to export the entity to Lua).

2.13.2.26 map:get_entities([prefix])

Returns an iterator to all map entities whose name has the specified prefix.

The typical usage of this function is:

```
for entity in map:get_entities("your_prefix") do
  -- some code related to the entity
end
```

- `prefix` (string, optional): Prefix of the entities to get. No value means all entities (equivalent to an empty prefix).
- Return value (function): An iterator to all entities with this prefix. Entities are returned in Z order (insertion order).

2.13.2.27 map:get_entities_count(prefix)

Returns the number of map entities having the specified prefix.

- `prefix` (string): Prefix of the entities to count.
- Return value (number): The number of entities having this prefix on the map.

2.13.2.28 map:has_entities(prefix)

Returns whether there exists at least one map entity having the specified prefix.

This function can be used for example to checker whether a group of enemies is dead.

- `prefix` (string): Prefix of the entities to check.
- Return value (boolean): `true` if at least one entity with this prefix exists on the map.

Remarks

Equivalent to `map:get_entities_count(prefix) > 0` but faster when there are a lot of entities (because it stops searching as soon as there is a match).

2.13.2.29 map:get_entities_by_type(type)

Returns an iterator to all map entities of the given type on the map.

The typical usage of this function is:

```
for entity in map:get_entities_by_type(type) do
  -- some code related to the entity
end
```

- `type` (string): Name of an entity type. See entity:get_type() for the possible values.

2.13.2.30 map:get_entities_in_rectangle(x, y, width, height)

Returns an iterator to all map entities whose maximum bounding box intersects the given rectangle. The maximum bounding box is the union of the entity's own bounding box and of the bounding boxes from its sprites.

The typical usage of this function is:

```
for entity in map:get_entities_in_rectangle(x, y, width, height) do
  -- some code related to the entity
end
```

- `x` (number): X coordinate of the rectangle's upper-left corner.
- `y` (number): Y coordinate of the rectangle's upper-left corner.
- `width` (number): Width of the rectangle.
- `height` (number): Height of the rectangle.
- Return value (function): An iterator to all entities intersecting the rectangle. Entities are returned in Z order (insertion order).

2.13.2.31 map:get_entities_in_region(x, y), map:get_entities_in_region(entity)

Returns an iterator to all map entities that are in a region. Regions of the map are defined by the position of separators and map limits. The region of an entity is the one of its center point.

Regions should be rectangular. Non-convex regions, for example with an "L" shape, are not supported by this function.

The typical usage of this function is:

```
for entity in map:get_entities_in_region(my_entity) do
  -- some code related to the entity
end
```

To get entities in the same region as a point:

- `x` (number): X coordinate of the region to get.
- `y` (number): Y coordinate of the region to get.
- Return value (function): An iterator to all entities in the same region as the point.

To get entities in the same region as another entity:

- `entity` (entity): An entity.
- Return value (function): An iterator to all other entities in the same region.

2.13.2.32 map:get_hero()

Returns the hero.

- Return value (hero): The hero.

Remarks

Equivalent to `map:get_entity("hero")` but shorter to write. This function is provided for convenience as getting the hero is often needed.

2.13.2.33 map:set_entities_enabled(prefix, [enabled])

Enables or disables all map entities having the specified prefix.

Disabled entities are not displayed and are not updated. Therefore, they don't move and their collisions are no longer detected. But they still exist and can be enabled back later.

- `prefix` (string): Prefix of the entities to change.
- `enable` (boolean, optional): `true` to enable them, `false` to disable them. No value means `true`.

Remarks

Equivalent to calling entity:set_enabled() on a group of entities.

2.13.2.34 map:remove_entities(prefix)

Removes and destroys all map entities having the specified prefix.

Once an entity is removed, it is destroyed and it no longer exists on the map. A good practice is to avoid keeping references to destroyed entities in your scripts so that they can be garbage-collected by Lua.

- `prefix` (string): Prefix of the entities to remove from the map.

Remarks

Equivalent to calling entity:remove() on a group of entities.

2.13.2.35 map:create_destination(properties)

Creates an entity of type destination on the map.

- `properties` (table): A table that describles all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `direction` (number): Direction that the hero should take when arriving on the destination, between `0` (East) and `3` (South), or `-1` to keep his direction unchanged.
  - `sprite` (string, optional): Id of the animation set of a sprite to create for the destination. No value means no sprite (the destination will then be invisible).
  - `save_location` (string, optional): Whether to update the starting location of the player when arriving to this destination. If yes, when the player restarts his game, he will restart at this destination. Must be one of:
    - `"when_world_changes"` (default): Updates the starting location if the current world has just changed when arriving to this destination.
    - `"yes"`: Updates the starting location.
    - `"no"`: Does not update the starting location.
  - `default` (boolean, optional): Sets this destination as the default one when teletransporting the hero to this map without destination specified. No value means `false`. Only one destination can be the default one on a map. If no default destination is set, then the first one declared becomes the default one.
- Return value (destination): The destination created.

2.13.2.36 map:create_teletransporter(properties)

Creates an entity of type teletransporter on the map.

- `properties` (table): A table that describles all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `width` (number): Width of the entity in pixels.
  - `height` (number): Height of the entity in pixels.
  - `sprite` (string, optional): Id of the animation set of a sprite to create for the teletransporter. No value means no sprite (the teletransporter will then be invisible).
  - `sound` (string, optional): Sound to play when the hero uses the teletransporter. No value means no sound.

  - `transition` (string, optional): Style of transition to play when the hero uses the teletransporter. Must be one of:
    * `"immediate"`: No transition.
    * `"fade"`: Fade-out and fade-in effect.
    * `"scrolling"`: Scrolling between maps. The default value is `"fade"`.
  - `destination_map` (string): Id of the map to transport to (can be the id of the current map).
  - `destination` (string, optional): Location on the destination map. Can be the name of a destination entity, the special value `"_same"` to keep the hero's coordinates, or the special value `"_side"` to place on hero on the corresponding side of an adjacent map (normally used with the scrolling transition style). No value means the default destination entity of the map.
- Return value (teletransporter): The teletransporter created.

2.13.2.37 map:create_pickable(properties)

Creates an entity of type pickable treasure on the map.

- `properties` (table): A table that describles all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `treasure_name` (string, optional): Kind of treasure to create (the name of an equipment item). If this value is not set, or corresponds to a non-obtainable item, then no entity is created and `nil` is returned.
  - `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
  - `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this pickable treasure was found. No value means that the treasure is not saved. If the treasure is saved and the player already has it, then no entity is be created and `nil` is returned.
- Return value (pickable treasure): The pickable treasure created, or `nil` if the item is not set, not obtainable, or if the pickable treasure is already found (for a saved one).

2.13.2.38 map:create_destructible(properties)

Creates an entity of type destructible object on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.

  - `treasure_name` (string, optional): Kind of pickable treasure to hide in the destructible object (the name of an equipment item). If this value is not set, then no treasure is placed in the destructible object. If the treasure is not obtainable when the object is destroyed, no pickable treasure is created.
  - `treasure_variant` (number, optional): Variant of the treasure if any (because some equipment items may have several variants). The default value is `1` (the first variant).
  - `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the pickable treasure hidden in the destructible object was found. No value means that the treasure (if any) is not saved. If the treasure is saved and the player already has it, then no treasure is put in the destructible object.
  - `sprite` (string): Name of the animation set of a sprite to create for the destructible object.
  - `destruction_sound` (string, optional): Sound to play when the destructible object is cut or broken after being thrown. No value means no sound.
  - `weight` (number, optional): Level of `"lift"` ability required to lift the object. `0` allows the player to lift the object unconditionally. The special value `-1` means that the object can never be lifted. The default value is `0`.
  - `can_be_cut` (boolean, optional): Whether the hero can cut the object with the sword. No value means `false`.
  - `can_explode` (boolean, optional): Whether the object should explode when it is cut, hit by a weapon and after a delay when the hero lifts it. The default value is `false`.
  - `can_regenerate` (boolean, optional): Whether the object should automatically regenerate after a delay when it is destroyed. The default value is `false`.
  - `damage_on_enemies` (number, optional): Number of life points to remove from an enemy that gets hit by this object after the hero throws it. If the value is `0`, enemies will ignore the object. The default value is `1`.
  - `ground` (string, optional): Ground defined by this entity. The ground is usually `"wall"`, but you may set `"traversable"` to make the object traversable, or for example `"grass"` to make it traversable too but with an additional grass sprite below the hero. The default value is `"wall"`.

- Return value (destructible object): The destructible object created.

Remarks

The state of the pickable treasure placed in the destructible object (obtained or not) and the possessed variant of the item (a number) are two independent values that have different meanings and are saved separately.

2.13.2.39 map:create_chest(properties)

Creates an entity of type treasure chest on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `treasure_name` (string, optional): Kind of treasure to place in the chest (the name of an equipment item). If this value is not set, then the chest will be empty. If the treasure is not obtainable when the hero opens the chest, it becomes empty.

  - `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
  - `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this chest is open. No value means that the state of the treasure is not saved. If the treasure is saved and the player already has it, then no treasure is placed in the chest (the chest will appear open).
  - `sprite` (string): Name of the animation set of the sprite to create for the chest. The sprite must have animations `"open"` and `"closed"`.
  - `opening_method` (string, optional): Specifies the permissions for the hero to open the chest. Must be one of:
    * `"interaction"` (default): Can be opened by pressing the action command in front of it.
    * `"interaction_if_savegame_variable"`: Can be opened by pressing the action command in front of it, provided that a specific savegame variable is set.
    * `"interaction_if_item"`: Can be opened by pressing the action command in front of it, provided that the player has a specific equipment item.
  - `opening_condition` (string, optional): The condition required to open the chest. Only for opening methods `"interaction_if_savegame_variable"` and `"interaction_if_item"`.
    * For opening method `"interaction_if_savegame_variable"`, it must be the name of a savegame variable. The hero will be allowed to open the chest if this saved value is either `true`, an integer greater than zero or a non-empty string.
    * For opening method `"interaction_if_item"`, it must be the name of an equipment item. The hero will be allowed to open the chest if he has that item and, for items with an amount, if the amount is greater than zero.
    * For the default opening method (`"interaction"`), this setting has no effect.
  - `opening_condition_consumed` (boolean, optional): Whether opening the chest should consume the savegame variable or the equipment item that was required. The default setting is `false`. If you set it to `true`, the following rules are applied when the hero successfully opens the chest:
    * For opening method `"interaction_if_savegame_variable"`, the savegame variable that was required is reset to `false`, `0` or `""` (depending on its type).
    * For opening method is `"interaction_if_item"`, the equipment item that was required is removed. This means setting its possessed variant to `0`, unless it has an associated amount: in this case, the amount is decremented.
  - `cannot_open_dialog` (string, optional): Id of the dialog to show if the hero fails to open the chest. If you don't set this value, no dialog is shown.

- Return value (chest): The treasure chest created.

Remarks

The state of the treasure chest (obtained or not) and the possessed variant of its item are two independent values that have different meanings and are saved separately.

2.13.2.40 map:create_jumper(properties)

Creates an entity of type jumper on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.

  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `width` (number): Width of the entity in pixels.
  - `height` (number): Height of the entity in pixels.
  - `direction` (number): Direction of the jump, between `0` (East) and `7` (South-East). If the direction is horizontal, the width must be `8` pixels. If the direction is vertical, the height must be `8` pixels. If the direction is diagonal, the size must be square.
  - `jump_length` (number): Length of the baseline of the jump in pixels (see the jump movement page for details).
- Return value (jumper): The jumper created.

2.13.2.41 map:create_enemy(properties)

Creates an entity of type enemy on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `direction` (number): Initial direction of the enemy, between `0` (East) and `3` (South).
  - `breed` (string): Model of enemy to create.
  - `savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this enemy is dead. No value means that the enemy is not saved. If the enemy is saved and was already killed, then no enemy is created. Instead, its pickable treasure is created if it is a saved one.
  - `treasure_name` (string, optional): Kind of pickable treasure to drop when the enemy is killed (the name of an equipment item). If this value is not set, then the enemy won't drop anything. If the treasure is not obtainable when the enemy is killed, then nothing is dropped either.
  - `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
  - `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the pickable treasure of this enemy was obtained. No value means that the state of the treasure is not saved. If the treasure is saved and the player already has it, then the enemy won't drop anything.
- Return value (enemy or pickable treasure): The enemy created, except when it is a saved enemy that is already dead. In this case, if the enemy dropped a saved treasure that is not obtained yet, this pickable treasure is created and returned. Otherwise, `nil` is returned.

Remarks

The state of the enemy (alive or dead), the state of its treasure dropped (obtained or not) and the possessed variant of the item dropped (a number) are three independent values that have different meanings and are saved separately.

2.13.2.42 map:create_npc(properties)

Creates an entity of type non-playing character (NPC) on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `direction` (number): Initial direction of the NPC's sprite, between `0` (East) and `3` (South).
  - `subtype` (number): Kind of NPC to create: `1` for a usual NPC who the player can talk to, `0` for a generalized NPC (not necessarily a person). See the NPC documentation for more details.
  - `sprite` (string, optional): Name of the animation set of a sprite to create for the NPC. No value means no sprite (the NPC will then be invisible).
  - `behavior` (string, optional): What to do when there is an interaction with the NPC.
    - `"dialog#XXXX"`: Starts the dialog with id `XXXX` when the player talks to this NPC.
    - `"map"` (default): Forwards events to the map script (for example, calls the on_interaction() event of the NPC).
    - `"item#XXXX"`: Forwards events to an equipment item script (for example, calls the on_interaction() event of the equipment item with id `XXXX`).
- Return value (NPC): the NPC created.

2.13.2.43 map:create_block(properties)

Creates an entity of type block on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `direction` (number, optional): The only direction where the block can be moved, between `0` (East) and `3` (South). No value means no restriction and allows the block to be moved in any of the four main directions.
  - `sprite` (string): Name of the animation set of a sprite to create for the block.
  - `pushable` (boolean): `true` to allow the block to be pushed.
  - `pullable` (boolean): `true` to allow the block to be pulled.
  - `maximum_moves` (number): Indicates how many times the block can be moved (`0`: none, `1`: once, `2`: infinite).
- Return value (block): the block created.

2.13.2.44 map:create_dynamic_tile(properties)

Creates an entity of type dynamic tile on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate of the top-left corner of the dynamic tile on the map.
  - `y` (number): Y coordinate of the top-left corner of the dynamic tile on the map.
  - `width` (number): Width of the dynamic tile in pixels. The tile pattern will be repeated horizontally to fit to this width.
  - `height` (number): Height of the entity in pixels. The tile pattern will be repeated vertically to fit to this height.
  - `tile_pattern_id` (string): Id of the tile pattern to use from the tileset.
  - `enabled_at_start` (boolean): `true` to make the dynamic tile initially enabled, `false` to make it initially disabled.
- Return value (dynamic tile): the dynamic tile created.

2.13.2.45 map:create_switch(properties)

Creates an entity of type switch on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `subtype` (string): Kind of switch to create:
    - `"walkable"`: A traversable pressure plate that gets activated when the hero walks on it.
    - `"solid"`: A non-traversable, solid switch that can be activated in various conditions: by the sword, by an explosion or by a projectile (a thrown object, an arrow, the boomerang or the hookshot).
    - `"arrow_target"` A switch that can be only activated by shooting an arrow on it.
  - `sprite` (string): Name of the animation set of a sprite to create for the switch. The animation set must at least contain animations `"activated"` and `"inactivated"`. No value means no sprite.
  - `sound` (string, optional): Sound to play when the switch is activated. No value means no sound.
  - `inactivate_when_leaving` (boolean): If `true`, the switch becomes inactivated when the hero or the block leaves it (only for a walkable switch).
- Return value (switch): the switch created.

2.13.2.46 map:create_wall(properties)

Creates an entity of type wall on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `width` (number): Width of the entity in pixels.
  - `height` (number): Height of the entity in pixels.
  - `stops_hero` (boolean, optional): `true` to make the wall stop the hero. No value means `false`.
  - `stops_npcs` (boolean, optional): `true` to make the wall stop non-playing characters. No value means `false`.
  - `stops_enemies` (boolean, optional): `true` to make the wall stop enemies. No value means `false`.
  - `stops_blocks` (boolean, optional): `true` to make the wall stop blocks. No value means `false`.
  - `stops_projectiles` (boolean, optional): `true` to make the wall stop projectiles: thrown objects, arrows, the hookshot and the boomerang. No value means `false`.
- Return value (wall): the wall created.

2.13.2.47 map:create_sensor(properties)

Creates an entity of type sensor on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `width` (number): Width of the entity in pixels.
  - `height` (number): Height of the entity in pixels.
- Return value (sensor): the sensor created.

2.13.2.48 map:create_crystal(properties)

Creates an entity of type crystal on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
- Return value (crystal): the crystal created.

2.13.2.49 map:create_crystal_block(properties)

Creates an entity of type crystal block on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `width` (number): Width of the entity in pixels.
  - `height` (number): Height of the entity in pixels.
  - `subtype` (number): Kind of crystal block to create: `0` for a block initially lowered (orange), `1` for a block initially raised (blue).
- Return value (crystal block): the crystal block created.

2.13.2.50 map:create_shop_treasure(properties)

Creates an entity of type shop treasure on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `price` (number): Money amount required to buy the treasure.
  - `font` (string, optional): Id of the font to use to display to price. The default value is the first one in alphabetical order.
  - `dialog` (string): Id of the dialog to show when the hero asks for information about the treasure.
  - `treasure_name` (string): Kind of treasure to sell (the name of an equipment item). If this value or corresponds to a non-obtainable item, then the shop treasure is not created and `nil` is returned.
  - `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
  - `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the player has purchased this treasure. No value means that the state of the treasure is not saved. If the treasure is saved and the player already has it, then the shop treasure is not created and `nil` is returned.
- Return value (shop treasure): The shop treasure created, or `nil` if the item is not obtainable, or if the shop treasure was already purchased (for a saved one).

Remarks

The state of the shop treasure (purchased or not) and the possessed variant of its item (a number) are two independent values that have different meanings and are saved separately.

2.13.2.51 map:create_stream(properties)

Creates an entity of type stream on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `direction` (number): Direction where the stream moves the hero, between `0` (East) and `7` (South-East).
  - `sprite` (string, optional): Id of the animation set of a sprite to create for the stream. No value means no sprite (the stream will then be invisible).
  - `speed` (number, optional): Speed of the movement applied to the hero by the stream, in pixels per second. The default value is `64`.
  - `allow_movement` (boolean, optional): Whether the player can still move the hero when he is on the stream. The default value is `true`.
  - `allow_attack` (boolean, optional): Whether the player can use the sword when he is on the stream. The default value is `true`.
  - `allow_item` (boolean, optional): Whether the player can use equipment items when he is on the stream. The default value is `true`.
- Return value (stream): the stream created.

2.13.2.52 map:create_door(properties)

Creates an entity of type door on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `direction` (number): Direction of the door, between `0` (East of the room) and `3` (South of the room).
  - `sprite` (string): Name of the animation set of the sprite to create for the door. The sprite must have an animation `"closed"`, that will be shown while the door is closed. When the door is open, no sprite is displayed. Optionally, the sprite can also have animations `"opening"` and `"closing"`, that will be shown (if they exist) while the door is being opened or closed, respectively. If they don't exist, the door will open close instantly.
  - `savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this door is open. No value means that the door is not saved. If the door is saved as open, then it appears open.

  - `opening_method` (string, optional): How the door is supposed to be opened by the player. Must be one of:
    * `"none"` (default): Cannot be opened by the player. You can only open it from Lua.
    * `"interaction"`: Can be opened by pressing the action command in front of it.
    * `"interaction_if_savegame_variable"`: Can be opened by pressing the action command in front of it, provided that a specific savegame variable is set.
    * `"interaction_if_item"`: Can be opened by pressing the action command in front of it, provided that the player has a specific equipment item.
    * `"explosion"`: Can be opened by an explosion.
  - `opening_condition` (string, optional): The condition required to open the door. Only for opening methods `"interaction_if_savegame_variable"` and `"interaction_if_item"`.
    * For opening method `"interaction_if_savegame_variable"`, it must be the name of a savegame variable. The hero will be allowed to open the door if this saved value is either `true`, an integer greater than zero or a non-empty string.
    * For opening method `"interaction_if_item"`, it must be the name of an equipment item. The hero will be allowed to open the door if he has that item and, for items with an amount, if the amount is greater than zero.
    * For other opening methods, this setting has no effect.
  - `opening_condition_consumed` (boolean, optional): Whether opening the door should consume the savegame variable or the equipment item that was required. The default setting is `false`. If you set it to `true`, the following rules are applied when the hero successfully opens the door:
    * For opening method `"interaction_if_savegame_variable"`, the savegame variable that was required is reset to `false`, `0` or `""` (depending on its type).
    * For opening method is `"interaction_if_item"`, the equipment item that was required is removed. This means setting its possessed variant to `0`, unless it has an associated amount: in this case, the amount is decremented.
    * With other opening methods, this setting has no effect.
  - `cannot_open_dialog` (string, optional): Id of the dialog to show if the hero fails to open the door. If you don't set this value, no dialog is shown.

- Return value (door): The door created.

2.13.2.53 map:create_stairs(properties)

Creates an entity of type stairs on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `direction` (number): Direction where the stairs should be turned between `0` (East of the room) and `3` (South of the room). For stairs inside a single floor, this is the direction of going upstairs.
  - `subtype` (number): Kind of stairs to create:
    * `0`: Spiral staircase going upstairs.
    * `1`: Spiral staircase going downstairs.
    * `2`: Straight staircase going upstairs.
    * `3`: Straight staircase going downstairs.
    * `4`: Small stairs inside a single floor (change the layer of the hero).

- Return value (stairs): the stairs created.

2.13.2.54 map:create_bomb(properties)

Creates an entity of type bomb on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
- Return value (bomb): the bomb created.

2.13.2.55 map:create_explosion(properties)

Creates an entity of type explosion on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
- Return value (explosion): the explosion created.

2.13.2.56 map:create_fire(properties)

Creates an entity of type fire on the map.

- `properties` (table): A table that describes all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
- Return value (fire): the fire created.

2.13.2.57 map:create_separator(properties)

Creates an entity of type separator on the map.

- `properties` (table): A table that describles all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `width` (number): Width of the entity in pixels.
  - `height` (number): Height of the entity in pixels. One of `width` or `height` must be 16 pixels.
- Return value (separator): The separator created.

2.13.2.58 map:create_custom_entity(properties)

Creates an entity of type custom entity on the map.

- `properties` (table): A table that describles all properties of the entity to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity or `nil`. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique.
  - `direction` (number): Direction of the custom entity, between `0` (East) and `3` (South). This direction will be applied to the entity's sprites if possible.
  - `layer` (number): The layer, between map:get_min_layer() and map:get_max_layer().
  - `x` (number): X coordinate on the map.
  - `y` (number): Y coordinate on the map.
  - `width` (number): Width of the entity in pixels (default `16`).
  - `height` (number): Height of the entity in pixels (default `16`).
  - `sprite` (string, optional): Name of the animation set of a sprite to create for the custom entity.
  - `model` (string, optional): Model of custom entity or `nil`. The model is the name of a Lua script in the `"entities"` directory of your quest. It will define the behavior of your entity. This script will be called with the entity as parameter. Models are useful when you need to create lots of similar entities, especially in different maps. `nil` means no model: in this case, no particular script will be called but you can still define the behavior of your entity in the map script.
- Return value (custom entity): The custom entity created.

2.13.3 Events of a map

Events are callback methods automatically called by the engine if you define them. In the case of maps, they are only called on the current map.

2.13.3.1 map:on_started(destination)

Called when this map starts (when the player enters it).

- `destination` (destination): The destination entity from where the hero arrives on the map, or `nil` if he used another way than a destination entity (like the side of the map or direct coordinates).

2.13.3.2 map:on_finished()

Called when this map stops (when the player leaves it).

2.13.3.3 map:on_update()

Called at each cycle of the main loop while this map is the current one.

Remarks

As this function is called at each cycle, it is recommended to use other solutions when possible, like timers and other events.

2.13.3.4 map:on_draw(dst_surface)

Called when the map has just been redrawn by the engine.

The engine has already drawn the map, but not the menus of this map if any. Use this event if you want to draw some additional content on the map before the menus, for example an overlay.

- `dst_surface` (surface): The surface where the map is drawn. This surface represents the visible part of the screen, not the whole map.

2.13.3.5 map:on_suspended(suspended)

Called when the map has just been suspended or resumed.

The map is suspended by the engine in a few cases, like when the game is paused or when the camera is being moved by a script. When this happens, all map entities stop moving and most sprites stop their animation.

- `suspended` (boolean): `true` if the map was just suspended, `false` if it was resumed.

2.13.3.6 map:on_opening_transition_finished(destination)

When the map begins, called when the opening transition effect finishes. After that moment, the player has the control of the hero.

- `destination` (destination): The destination entity from where the hero arrived on the map, or `nil` if he used another way than a destination entity (like the side of the map or direct coordinates).

2.13.3.7 map:on_obtaining_treasure(treasure_item, treasure_variant, treasure_savegame_variable)

Called when the hero is obtaining a treasure on this map, before the treasure's dialog (if any).

- `treasure_item` (item): Equipment item being obtained.
- `treasure_variant` (number): Variant of the treasure (because some equipment items may have several variants).
- `treasure_savegame_variable` (string): Name of the boolean value that stores in the savegame whether this treasure is found, or `nil` if this treasure is not saved.

2.13.3.8 map:on_obtained_treasure(treasure_item, treasure_variant, treasure_savegame_variable)

Called after the hero has obtained a treasure on this map.

In the case of a brandished treasure, this event is called once the treasure's dialog is finished. Otherwise, it is called immediately after map:on_obtaining_treasure().

- `treasure_item` (item): Equipment item being obtained.
- `treasure_variant` (number): Variant of the treasure (because some equipment items may have several variants).
- `treasure_savegame_variable` (string): Name of the boolean value that stores in the savegame whether this treasure is found, or `nil` if this treasure is not saved.

2.13.3.9 map:on_key_pressed(key, modifiers)

Called when the user presses a keyboard key while your map is active.

- `key` (string): Name of the raw key that was pressed.
- `modifiers` (table): A table whose keys indicate what modifiers were down during the event. Possible table keys are `"shift"`, `"control"` and `"alt"`. Table values don't matter.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects. If you return `false` or nothing, the event will continue its propagation to the commands.

Remarks

This event indicates the raw keyboard key pressed. If you want the corresponding character instead (if any), see map:on_character_pressed(). If you want the corresponding higher-level game command (if any), see map:on_command_pressed().

2.13.3.10 map:on_key_released(key, modifiers)

Called when the user releases a keyboard key while your map is active.

- `key` (string): Name of the raw key that was released.
- `modifiers` (table): A table whose keys indicate what modifiers were down during the event. Possible table keys are `"shift"`, `"control"` and `"alt"`. Table values don't matter.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects. If you return `false` or nothing, the event will continue its propagation to the commands.

Remarks

This event indicates the raw keyboard key pressed. If you want the corresponding character instead (if any), see map:on_character_pressed(). If you want the corresponding higher-level game command (if any), see map:on_command_pressed().

2.13.3.11 map:on_character_pressed(character)

Called when the user enters text while your map is active.

- `character` (string): A utf-8 string representing the character that was pressed.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects. If you return `false` or nothing, the event will continue its propagation to the commands.

Remarks

When a character key is pressed, two events are called: map:on_key_pressed() (indicating the raw key) and map:on_character_pressed() (indicating the utf-8 character). If your script needs to input text from the user, map:on_character_pressed() is what you want because it considers the keyboard's layout and gives you international utf-8 strings.

2.13.3.12 map:on_joypad_button_pressed(button)

Called when the user presses a joypad button while your map is active.

- `button` (number): Index of the button that was pressed.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.13.3.13 map:on_joypad_button_released(button)

Called when the user releases a joypad button while your map is active.

- `button` (number): Index of the button that was released.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.13.3.14 map:on_joypad_axis_moved(axis, state)

Called when the user moves a joypad axis while your map is active.

- `axis` (number): Index of the axis that was moved. Usually, `0` is an horizontal axis and `1` is a vertical axis.
- `state` (number): The new state of the axis that was moved. `-1` means left or up, `0` means centered and `1` means right or down.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.13.3.15 map:on_joypad_hat_moved(hat, direction8)

Called when the user moves a joypad hat while your map is active.

- `hat` (number): Index of the hat that was moved.
- `direction8` (number): The new direction of the hat. `-1` means that the hat is centered. `0` to `7` indicates that the hat is in one of the eight main directions.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.13.3.16 map:on_command_pressed(command)

Called when the player presses a game command (a keyboard key or a joypad action mapped to a built-in game behavior) while this map is active. You can use this event to override the normal built-in behavior of the game command.

- `command` (string): Name of the built-in game command that was pressed. Possible commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (you are overriding the built-in behavior of pressing this game command).

Remarks

This event is not triggered if you already handled its underlying low-level keyboard or joypad event.

2.13.3.17 map:on_command_released(command)

Called when the player released a game command (a keyboard key or a joypad action mapped to a built-in game behavior). while this map is active. You can use this event to override the normal built-in behavior of the game command.

- `command` (string): Name of the built-in game command that was released. Possible commands are `"action"`, `"attack"`, `"pause"`, `"item_1"`, `"item_2"`, `"right"`, `"up"`, `"left"` and `"down"`.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects (you are overriding the built-in behavior of releasing this game command).

Remarks

This event is not triggered if you already handled its underlying low-level keyboard or joypad event.

2.13.3.18 map:on_mouse_pressed(button, x, y)

Called when the user presses a mouse button while this map is active.

- `button` (string): Name of the mouse button that was pressed. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- `x` (integer): The x position of the mouse in quest size coordinates.
- `y` (integer): The y position of the mouse in quest size coordinates.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.13.3.19 map:on_mouse_released(button, x, y)

Called when the user releases a mouse button while this map is active.

- `button` (string): Name of the mouse button that was released. Possible values are `"left"`, `"middle"`, `"right"`, `"x1"` and `"x2"`.
- `x` (integer): The x position of the mouse in quest size coordinates.
- `y` (integer): The y position of the mouse in quest size coordinates.
- Return value (boolean): Indicates whether the event was handled. If you return `true`, the event won't be propagated to other objects.

2.13.4 Deprecated methods of the type map

The following methods are deprecated and may be removed it future releases.

2.13.4.1 map:get_camera_position()

Returns the currently visible area of the map.

Warning

This method is deprecated because since Solarus 1.5, the camera is now an entity. Therefore, you can get its coordinates and size like any other entity.

Use map:get_camera():get_bounding_box() instead.

2.13.4.2 map:move_camera(x, y, speed, callback, [delay_before], [delay_after])

Starts a camera moving sequence.

Warning

This method is deprecated because since Solarus 1.5, the camera is now an entity. Therefore, you can move the camera like any other entity and have much more customization possibilities.

It can be rewritten in pure Lua as follows.

```
function map:move_camera(x, y, speed, callback, delay_before, delay_after)

  local camera = map:get_camera()
  local game = map:get_game()
  local hero = map:get_hero()

  delay_before = delay_before or 1000
  delay_after = delay_after or 1000

  local back_x, back_y = camera:get_position_to_track(hero)
  game:set_suspended(true)
  camera:start_manual()

  local movement = sol.movement.create("target")
  movement:set_target(camera:get_position_to_track(x, y))
  movement:set_ignore_obstacles(true)
  movement:set_speed(speed)
  movement:start(camera, function()
    local timer_1 = sol.timer.start(map, delay_before, function()
      callback()
      local timer_2 = sol.timer.start(map, delay_after, function()
        local movement = sol.movement.create("target")
        movement:set_target(back_x, back_y)
        movement:set_ignore_obstacles(true)
        movement:set_speed(speed)
        movement:start(camera, function()
          game:set_suspended(false)
          camera:start_tracking(hero)
          if map.on_camera_back ~= nil then
            map:on_camera_back()
          end
        end)
      end)
      timer_2:set_suspended_with_map(false)
    end)
    timer_1:set_suspended_with_map(false)
  end)
end
```

This function temporarily moves the camera somewhere else, like to a place where a chest or an enemy will appear.

The camera first moves towards a target point. When the target is reached, after a first delay, your callback function is called. Then, after a second delay, the camera moves back towards the hero. The game is suspended during the whole sequence.

- `x` (number): X coordinate of the target, relative to the map's top-left corner.
- `y` (number): Y coordinate of the target, relative to the map's top-left corner.
- `speed` (number): Speed of the camera movement in pixels per second.
- `callback` (function): A function to be called when the camera reaches the target (after `delay_before`).
- `delay_before` (number, optional): A delay in milliseconds before calling your function once the target is reached (default `1000`).
- `delay_after` (number, optional): A delay in milliseconds after calling your function, before the camera goes back (default `1000`).

2.13.5 Deprecated events of a map

The following events are deprecated and may be removed it future releases.

2.13.5.1 map:on_camera_back()

After a camera sequence initiated by map:move_camera(), this event is triggered when the camera gets back to the hero.

Warning

> This event is deprecated because since Solarus 1.5, camera:move() is deprecated. The camera is now an entity and now has much more customization possibilities.

Use camera:on_state_changed() instead, or the callback parameter of movement:start()

2.14 Map entities

Objects placed on the map are called map entities (or just entities).

There exists many types of entities. They can be either declared in the map data file, or created dynamically by using the `map:create_*` methods of the map API.

2.14.1 Overview

All entities have a position on the map (X, Y and layer) and a size. Depending on their type, they can be visible or not. When they are visible, they are usually represented by one or several sprites. Some entities are fixed, others move according to a movement object.

Entities can also have a name that uniquely identifies them on the map. This is useful to access them from the map API. The name is optional, but if an entity has a name, it must be unique on the map.

Here are the existing types of entities.

- Hero: The character controlled by the player.
- Tile: A small brick that composes a piece of the map, with a pattern picked from the tileset.
- Dynamic tile: A special tile that can be enabled or disabled dynamically (usual tiles are optimized away at runtime).
- Teletransporter: When walking on it, the hero is transported somewhere, possibly on the same map or another map.
- Destination: A possible destination place for teletransporters.
- Pickable treasure: A treasure placed on the ground and that the hero can pick up.
- Destructible object: An entity that can be cut or lifted by hero, and that may hide a pickable treasure.
- Carried object: A destructible object lifted and carried by the hero.
- Chest: A chest that contains a treasure.

- Shop treasure: A treasure that the hero can buy for a price.
- Enemy: A bad guy (possibly a boss) who may also drop a pickable treasure when killed.
- Non-playing character (NPC): Somebody or something the hero can interact with.
- Block: An entity that the hero can push or pull.
- Jumper: When walking on it, the hero jumps into a direction.
- Switch: A button or another mechanism that the hero can activate.
- Sensor: An invisible detector that detects the presence of the hero.
- Separator: An horizontal or vertical separation between two parts of the map.
- Wall: An invisible object that stops some kinds of entities.
- Crystal: A switch that lowers or raises crystal blocks.
- Crystal block: A low wall that can be lowered (travserable) or raised (obstacle) using a crystal.
- Stream: When walking on it, the hero automatically moves into a direction.
- Door: A door to open with an equipment item or another condition.
- Stairs: Stairs between two maps or to a platform of a single map.
- Bomb: A bomb that will explode after a few seconds and that may be lifted by the hero.
- Explosion: An explosion that can hurt the hero and the enemies.
- Fire: A flame that can hurt enemies and interact with other entities.
- Arrow: An arrow shot by the bow.
- Hookshot: A hookshot shot by the hero.
- Boomerang: A boomerang shot by the hero.
- Camera: A rectangle that determines the visible area of the map.
- Custom entity: An entity fully controlled by your Lua scripts.

Remarks

Note that sprites and movements are not map entities, but they can be attached to map entities (to display them and to move them, respectively). Sprites and movements can also be used outside a map, for example in your title screen or in other menus.

2.14.2 Methods of all entity types

These methods exist in all entity types.

2.14.2.1 entity:get_type()

Returns the type of entity.

- Return value (string): The type of this entity. Can be one of: `"hero"`, `"dynamic_tile"`, `"teletransporter"`, `"destination"`, `"pickable"`, `"destructible"`, `"carried_object"`, `"chest"`, `"shop_treasure"`, `"enemy"`, `"npc"`, `"block"`, `"jumper"`, `"switch"`, `"sensor"`, `"separator"`, `"wall"`, `"crystal"`, `"crystal_block"`, `"stream"`, `"door"`, `"stairs"`, `"bomb"`, `"explosion"`, `"fire"`, `"arrow"`, `"hookshot"`, `"boomerang"` or `"custom_entity"`.

Remarks

The type `"tile"` is not is this list because tiles don't exist at runtime for optimization reasons.

2.14.2.2 entity:get_map()

Returns the map this entity belongs to.

- Return value (map): The map that contains this entity.

2.14.2.3 entity:get_game()

Returns the game that is running the map this entity belongs to.

- Return value (game): The current game.

2.14.2.4 entity:get_name()

Returns the name of this map entity.

The name uniquely identifies the entity on the map.

- Return value (string): The name of this entity, or `nil` if the entity has no name (because the name is optional).

2.14.2.5 entity:exists()

Returns whether this entity still exists on the map.

An entity gets destroyed when you call entity:remove() or when the engine removes it (for example an enemy that gets killed or a pickable treasure that gets picked up). If you refer from Lua to an entity that no longer exists in the C++ side, this method returns `false`.

- Return value (boolean): `true` if the entity exists, `false` if it was destroyed.

2.14.2.6 entity:remove()

Removes this entity from the map and destroys it.

After the entity is destroyed, entity:exists() returns `false` and there is no reason to keep a reference to it in the Lua side (though it is harmless).

2.14.2.7 entity:is_enabled()

Returns whether this entity is enabled.

When an entity is disabled, it is not displayed on the map, it does not move and does not detect collisions. But it still exists, it still has a position and it can be enabled again later.

- Return value (boolean): `true` if this entity is enabled.

2.14.2.8 entity:set_enabled([enabled])

Enables or disables this entity.

When an entity is disabled, it is not displayed on the map, it does not move and does not detect collisions. Its movement, its sprites and its timers if any are suspended and will be resumed when the entity gets enabled again. While the entity is disabled, it still exists, it still has a position and it can be enabled again later.

- `enabled` (boolean, optional): `true` to enable the entity, `false` to disable it. No value means `true`.

2.14.2.9 entity:get_size()

Returns the size of the bounding box of this entity.

The bounding box determines the position of the entity on the map. It is a rectangle whose width and height are multiples of 8 pixels. The bounding box is used to detect whether the entity overlaps obstacles or other entities.

- Return value 1 (number): Width of the entity in pixels.
- Return value 2 (number): Height of the entity in pixels.

Remarks

Note that the sprites of an entity may have a different size than the entity itself. See sprite:get_size() to know it.

2.14.2.10 entity:get_origin()

Returns the origin point of this entity, relative to the upper left corner of its bounding box.

When an entity is located at some coordinates on the map, the origin points determines what exact point of the entity's bounding box is at those coordinates. It is not necessarily the upper left corner of the entity's bounding box.

The origin point is usually be the central point of contact between the entity and the ground. For example, the origin point of the hero, enemies, non-playing characters and most entities is the center of their shadow. Thus, for entities of size `16,16`, the origin point is often `8,13`.

This origin point property allows entities of different sizes to have comparable reference points that can be used by the engine. Indeed, when two enemies overlap, the engine needs to determine which one has to be displayed first (it is always the one with the lowest Y coordinate). Sometimes, the engine also needs to compute an angle between two entities, for example to push away an enemy that was just hit. Using the upper left corner of their bounding box would not give the correct angle (unless both entities had the same size).

The origin point is also the point of synchronization of an entity with its sprites (because again, an entity that has a given size may have sprites with different sizes).

- Return value 1 (number): X coordinate of the origin point in pixels, relative to the upper left corner of the entity's bounding box.
- Return value 2 (number): Y coordinate of the origin point in pixels, relative to the upper left corner of the entity's bounding box.

2.14.2.11 entity:get_position()

Returns the position of this entity on the map (coordinates and layer).

- Return value 1 (number): X coordinate of the origin point of the entity, relative to the upper left corner of the map.
- Return value 2 (number): Y coordinate of the origin point of the entity, relative to the upper left corner of the map.
- Return value 3 (number): Layer where the entity is on the map, between map:get_min_layer() and map:get_max_layer().

2.14.2.12 entity:set_position(x, y, [layer]):

Changes instantly the position of this entity on the map (coordinates and layer). The origin point of the entity gets placed at these coordinates, relative to the map's upper left corner. Any previous movement or other action performed by the entity continues normally.

- `x` (number): X coordinate to set.
- `y` (number): Y coordinate to set.
- `layer` (number, optional): Layer to set, between map:get_min_layer() and map:get_max_layer(). By default, the layer is unchanged.

Remarks

Be careful: this function does not check collisions with obstacles.

2.14.2.13 entity:get_center_position()

Returns the coordinates of the center point of this entity on the map.

- Return value 1 (number): X coordinate of the center of this entity's bounding box, relative to the upper left corner of the map.
- Return value 2 (number): Y coordinate of the center of this entity's bounding box, relative to the upper left corner of the map.
- Return value 3 (number): Layer of the entity.

2.14.2.14 entity:get_facing_position()

Returns the coordinates of the point this entity is looking at. This point depends on the direction of the main sprite if any. If the entity has no sprite, or if the main sprite has not 4 directions, then the movement is considered. If there is no movement either, the entity is assumed to look to the North.

- Return value 1 (number): X coordinate of the facing point of this entity, relative to the upper left corner of the map.
- Return value 2 (number): Y coordinate of the facing point of this entity, relative to the upper left corner of the map.
- Return value 3 (number): Layer of the entity.

2.14.2.15 entity:get_facing_entity()

Returns the entity this entity is looking at, if any.

This is an entity overlapping the facing position of this entity. If several entities are overlapping the facing position, the first one in Z order is returned.

- Return value (entity): The facing entity, or `nil` if there is no entity in front of this entity.

2.14.2.16 entity:get_ground_position()

Returns the coordinates of the point used for ground detection for this entity on the map.

The ground position is the point tested by all features related to the ground, like all effects of various grounds on the hero, the result of entity:get_ground_below() and the event custom_entity:on_ground_below_changed().

- Return value 1 (number): X coordinate of the ground point of this entity, relative to the upper left corner of the map.
- Return value 2 (number): Y coordinate of the ground point of this entity, relative to the upper left corner of the map.
- Return value 3 (number): Layer of the entity.

Remarks

The ground point of an entity is slightly (2 pixels) above its origin point as returned by entity:get_position().

2.14.2.17 entity:get_ground_below()

Returns the map's ground below this entity.

The ground is defined by the topmost tile below this entity, plus potential dynamic entities that may affect the ground, like dynamic tiles, destructibles and custom entities.

The exact point tested is the one returned by entity:get_ground_position(), and it is slightly different from entity:get_position().

- Return value (string): The ground below this entity. See map:get_ground() for the list of possible grounds.

2.14.2.18 entity:get_bounding_box()

Returns the rectangle representing the coordinates and size of this entity on the map.

The bounding box determines the position of the entity on the map. It is a rectangle whose width and height are multiples of 8 pixels. The bounding box is used to detect whether the entity overlaps obstacles or other entities.

- Return value 1 (number): X coordinate of the upper left corner of the bounding box.
- Return value 2 (number): Y coordinate of the upper left corner of the bounding box.
- Return value 3 (number): Width of the bounding box.
- Return value 4 (number): Height of the bounding box.

Remarks

The sprites of this entity (if any) may exceed the bounding box. See entity:get_max_bounding_box().

2.14.2.19 entity:get_max_bounding_box()

Returns the rectangle surrounding the bounding box of this entity plus the bounding boxes of its sprites in all their possible animations and directions.

This is usually larger than entity:get_bounding_box(), because the sprite of an entity often exceeds its bounding box.

- Return value 1 (number): X coordinate of the upper left corner of the sprites bounding box.
- Return value 2 (number): Y coordinate of the upper left corner of the sprites bounding box.
- Return value 3 (number): Width of the sprites bounding box.
- Return value 4 (number): Height of the sprites bounding box.

2.14.2.20 entity:overlaps(x, y, [width, height])

Returns whether the bounding box of this entity overlaps the specified rectangle or point.

To test if this entity overlaps a rectangle or a point (a point is a rectangle of size 1x1):

- `x` (number): X coordinate of the upper left corner of the rectangle to check.
- `y` (number): Y coordinate of the upper left corner of the rectangle to check.
- `width` (number, optional): Width of the rectangle (default `1`).
- `height` (number, optional): Height of the rectangle (default `1`).
- Return value (boolean): `true` if the bounding box of this entity overlaps the rectangle.

2.14.2.21 entity:overlaps(other_entity, [collision_mode])

Returns whether another entity collides with this entity according to the specified collision test.

- `entity` (entity): Another entity.
- `collision_mode` (string, optional): Specifies what kind of collision you want to test. This may be one of:
  - `"overlapping"`: Collision if the bounding box of both entities overlap. This is the default value.
  - `"containing"`: Collision if the bounding box of the other entity is fully inside the bounding box of this entity.
  - `"origin"`: Collision if the origin point or the other entity is inside the bounding box of this entity.
  - `"center"`: Collision if the center point of the other entity is inside the bounding box of this entity.
  - `"facing"`: Collision if the facing position of the other entity's bounding box is touching this entity's bounding box. Bounding boxes don't necessarily overlap, but they are in contact: there is no space between them. When you consider the bounding box of an entity, which is a rectangle with four sides, the facing point is the middle point of the side the entity is oriented to. This `"facing"` collision test is useful when the other entity cannot traverse your custom entity. For instance, if the other entity has direction "east", there is a collision if the middle of the east side of its bounding box touches (but does not necessarily overlap) this entity's bounding box. This is typically what you need to let the hero interact with this entity when he is looking at it.
  - `"touching"`: Like `"facing"`, but accepts all four sides of the other entity's bounding box, no matter its direction.
  - `"sprite"`: Collision if a sprite of the other entity overlaps a sprite of this entity. The collision test is pixel precise.
- Return value (boolean): `true` if a collision is detected with this collision test.

Remarks

For custom entities, see also custom_entity:add_collision_test() to be automatically notified when a collision is detected.

2.14.2.22 entity:get_distance(x, y), entity:get_distance(other_entity)

Returns the distance in pixels between this map entity and a point or another map entity.

To compute the distance to a specified point:

- `x` (number): X coordinate of the point.
- `y` (number): Y coordinate of the point.
- Return value (number): The distance in pixels between the origin point of this entity and the point.

To compute the distance to another map entity:

- `other_entity` (entity): The entity to compute the distance to.
- Return value (number): The distance in pixels between the origin point of this entity and the origin point of the other entity.

2.14.2.23 entity:get_angle(x, y), entity:get_angle(other_entity)

Returns the angle between the X axis and the vector that joins this entity to a point.

To compute the angle to a specified point:

- `x` (number): X coordinate of the point.
- `y` (number): Y coordinate of the point.
- Return value (number): The angle in radians between the origin point of this entity and the specified point. The angle is between `0` and `2 * math.pi`.

To compute the angle to another map entity:

- `other_entity` (entity): The entity to compute the angle to.
- Return value (number): The angle in radians between the origin point of this entity and the origin point of the other entity. The angle is between `0` and `2 * math.pi`.

2.14.2.24 entity:get_direction4_to(x, y), entity:get_direction4_to(other_entity)

Like entity:get_angle(), but instead of an angle in radians, returns the closest direction among the 4 main directions.

This is a utility function that essentially rounds the result of entity:get_angle().

To compute the direction to a specified point:

- `x` (number): X coordinate of the point.
- `y` (number): Y coordinate of the point.
- Return value (number): The direction this entity should take to look at this point, between 0 (East) and 3 (South).

To compute the direction to another map entity:

- `other_entity` (entity): An entity to target.
- Return value (number): The direction this entity should take to look at the other entity, between 0 (East) and 3 (South).

2.14.2.25 entity:get_direction8_to(x, y), entity:get_direction8_to(other_entity)

Like entity:get_angle(), but instead of an angle in radians, returns the closest direction among the 8 main directions.

This is a utility function that essentially rounds the result of entity:get_angle().

To compute the direction to a specified point:

- `x` (number): X coordinate of the point.
- `y` (number): Y coordinate of the point.
- Return value (number): The direction this entity should take to look at this point, between 0 (East) and 7 (South-East).

To compute the direction to another map entity:

- `other_entity` (entity): An entity to target.
- Return value (number): The direction this entity should take to look at the other entity, between 0 (East) and 7 (South-East).

2.14.2.26 entity:snap_to_grid()

Makes sure this entity's upper left corner is aligned with the 8∗8 grid of the map.

Remarks

Be careful: this function does not check collisions with obstacles.

2.14.2.27 entity:bring_to_front()

Places this entity in front of all other entities on the same layer.

Since entities that are on the same layer can overlap, you can use this function to change their Z-index.

Remarks

Some entities like NPCs, enemies, the hero and some custom entities have the property to be displayed in Y order. The Z-index of these entities is already managed by the engine to show entities more to the north behind the ones more to the south. Therefore, this function has no effect on their drawing order.

2.14.2.28 entity:bring_to_back()

Places this entity behind all other entities on the same layer.

Since entities that are on the same layer can overlap, you can use this function to change their Z-index.

Remarks

Some entities like NPCs, enemies and the hero have the property to be displayed in Y order. The Z-index of these entities is already managed by the engine to show entities more to the north behind the ones more to the south. Therefore, this function has no effect on their drawing.

2.14.2.29 entity:get_optimization_distance()

Returns the optimization threshold hint of this map entity.

Above this distance from the camera, the engine may decide to skip updates or drawings. This is only a hint: the engine is responsible of the final decision. A value of `0` means an infinite distance (the entity is never optimized away).

- Return value (number): The optimization distance hint in pixels.

2.14.2.30 entity:set_optimization_distance(optimization_distance)

Sets the optimization threshold hint of this map entity.

Above this distance from the camera, the engine may decide to skip updates or drawings. This is only a hint: the engine is responsible of the final decision.

A value of `0` means an infinite distance (the entity is never optimized away). The default value is `0`.

- `optimization_distance` (number): The optimization distance hint to set in pixels.

2.14.2.31 entity:is_in_same_region(other_entity)

Returns whether this entity is in the same region as another one.

Regions of the map are defined by the position of separators and map limits. The region of an entity is the one of its center point.

Regions should be rectangular. Non-convex regions, for example with an "L" shape, are not supported by this function.

You can use this function to make sure that an enemy close to the hero but in the other side of a separator won't attack him.

- `other_entity` (entity): Another entity.
- Return value (boolean): `true` if both entities are in the same region.

2.14.2.32 entity:test_obstacles([dx, dy, [layer]])

Returns whether there would be a collision with obstacles if this map entity was translated.

- `dx` (number, optional): X component of the translation in pixels (`0` means the current X position). No value means `0`.
- `dx` (number, optional): Y component of the translation in pixels (`0` means the current Y position). No value means `0`.
- `layer` (number, optional): Layer to test. No value means the current layer.
- Return value (boolean): `true` if there would be a collision in this position.

2.14.2.33 entity:get_sprite([name])

Returns a sprite representing this entity.

To manage entities with multiple sprites, you can set names with sprite creation methods like enemy:create_sprite() and custom_entity:create_sprite(). However, it is easier to just leave the names blank and simply store the result of these sprite creation methods. The name is more useful for built-in entities that have multiple sprites automatically created by the engine. Such entities are the hero, pickable treasures, carried objects and crystals. See the documentation pages of these entities to know their exact sprites, the name of these sprites and which one is their main sprite. For enemies and custom entities, the main sprite is the first one in Z order, which is the sprite creation order unless you call entity:bring_sprite_to_front() or entity:bring_sprite_to_back().

- `name` (string, optional): Name of the sprite to get. Only useful for entities that have multiple sprites. No value means the main sprite.
- Return value (sprite): The entity sprite with this name, or its main sprite if no name is specified. Returns `nil` if the entity has no such sprite.

2.14.2.34 entity:get_sprites()

Returns an iterator to all sprites of this entity.

At each step, the iterator provides two values: the name of a sprite (which is an empty string if the sprite has no name) and the sprite itself. See entity:get_sprite() for more details about named sprites.

Sprites are returned in their displaying order. Note that this order can be changed with entity:bring_sprite_to_front() and entity:bring_sprite_to_back().

The typical usage of this function is:

```
for sprite_name, sprite in entity:get_sprites() do
  -- some code related to the sprite
end
```

- Return value (function): An iterator to all sprites of this entity.

2.14.2.35 entity:bring_sprite_to_front(sprite)

Reorders a sprite of this entity to be displayed after other sprites (displayed to the front).

This function is only useful for entities that have multiple sprites.

- `sprite` (sprite): The sprite to reorder. It must belong to this entity.

2.14.2.36 entity:bring_sprite_to_back(sprite)

Reorders a sprite of this entity to be displayed before other sprites (displayed to the back).

This function is only useful for entities that have multiple sprites.

- `sprite` (sprite): The sprite to reorder. It must belong to this entity.

2.14.2.37 entity:is_visible()

Returns whether this entity is visible.

When the entity is hidden, its sprites (if any) are not displayed, but everything else continues normally, including collisions.

- Return value (boolean): `true` if the entity is visible.

2.14.2.38 entity:set_visible([visible])

Hides or shows the entity.

When the entity is hidden, its sprites (if any) are not displayed, but everything else continues normally, including collisions.

- `visible` (boolean, optional): `true` to show the entity, `false` to hide it. No value means `true`.

2.14.2.39 entity:get_movement()

Returns the current movement of this map entity.

- Return value (movement): The current movement, or `nil` if the entity is not moving.

2.14.2.40 entity:stop_movement()

Stops the current movement of this map entity if any.

2.14.3 Events of all entity types

Events are callback methods automatically called by the engine if you define them.

2.14.3.1 entity:on_created()

Called when this entity has just been created on the map.

2.14.3.2 entity:on_removed()

Called when this entity is about to be removed from the map (and therefore destroyed).

2.14.3.3 entity:on_position_changed(x, y, layer)

Called when the coordinates of this entity have just changed.

- `x` (number): The new X coordinate of the entity.
- `y` (number): The new Y coordinate of the entity.
- `layer` (number): The new layer of the entity.

2.14.3.4 entity:on_obstacle_reached(movement)

Called when the movement of this entity was stopped because of an obstacle.

When an obstacle is reached, this event is called instead of entity:on_position_changed().

- `movement` (movement): The movement of the entity.

2.14.3.5 entity:on_movement_started(movement)

Called when a movement is started on this entity.

- `movement` (movement): The movement that was just started on this entity.

2.14.3.6 entity:on_movement_changed(movement)

Called when some characteristics of this entity's movement (like the speed or the angle) have just changed.

- `movement` (movement): The movement of the entity.

2.14.3.7 entity:on_movement_finished()

Called when the movement of the entity is finished (if there is an end).

2.14.4 Hero

The hero is the character controlled by the player. There is always exactly one hero on the current map. The hero is automatically created by the engine: you cannot create or remove him.

2.14.4.1 Overview

The name of the hero (as returned by entity:get_name()) is always `"hero"`. Therefore, from a script, you can access the hero with `map:get_entity("hero")` or, if you are in a map script, directly with the `hero` variable (see map:get_entity() for more details). There are also quick accessors map:get_hero() and game:get_hero() to retrieve the hero more easily.

His size is always 16x16 pixels.

The hero entity persists accross map changes. In other words, the effect of functions like hero:set_walking_speed() persists when the player goes to another map. Similarly, events that you define on the hero (like hero:on_taking_damage()) continue to get called on any map while the game is active.

We describe here the Lua API that you can use to change the hero's state.

2.14.4.1.1 Hero sprites

Multiple sprites for the hero are automatically created by the engine. You can access them like for any other entity, specifying their name in entity:get_sprite([name]).

Here is the list of hero sprites created by the engine and their names:

- `"tunic"`: Main sprite of the hero. It always exists. This is the default one in entity:get_sprite([name]).
- `"shield"`: Shield if any.
- `"sword"`: Sword if any.
- `"sword_stars"`: Small stars sparkling near the sword in some states.
- `"shadow"`: Shadow displayed under the hero in some states.
- `"trail"`: Trail of dust displayed under the hero when running.
- `"ground"`: Ground displayed under the hero when walking on special terrain like shallow water or grass.

Keep in mind that depending on the hero's equipment, his state and the ground below him, all of these built-in sprites don't always exist, and some of them may exist at some point but without being currently displayed.

2.14.4.2 Methods inherited from map entity

The hero is a particular map entity. Therefore, he inherits all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.4.3 Methods of the type hero

The following methods are specific to the hero.

2.14.4.3.1 hero:teleport(map_id, [destination_name, [transition_style]])

Teletransports the hero to a different place.

- `map_id` (string): Id of the map to go to (may be the same map or another one). If the map does not exist, the teletransportation fails and this function generates a Lua error. If the map exists, then the teletransportation is always successful even if there is no viable destination (see below).
- `destination_name` (string, optional): Name of the destination entity where to go on that map, or the special keyword `"_same"` to keep the same coordinates. Can also be the special keyword `"_side0"`, `"_side1"`, `"_side2"` or `"_side3"` to arrive near the East, North, West or South frontier of the map respectively. But the hero should be near the corresponding side of the original map for this to look okay. This is usually used in combination with scrolling transitions. No value means the default destination entity of the map. If the destination does not exist, a debugging message is logged and the default destination is used as a fallback. Finally, if there is no destination at all, then no default destination can be used. In this case, another debugging message is logged and the hero is placed at coordinates (0,0).
- `transition_style` (string, optional): `"immediate"` (no transition effect) `"fade"` (fade-out and fade-in effect) or `"scrolling"`. The default value is `"fade"`.

Remarks

The transportation will occur at the next cycle of the engine's main loop. Therefore, your map is not unloaded immediately and your map script continues to work.

2.14.4.3.2 hero:get_direction()

Returns the direction of the hero's sprites.

- Return value (number): The direction of the hero's sprites, between `0` (East) and 3 (South).

Remarks

The direction of the hero's sprites may be different from both the direction pressed by the player's commands" and from the actual direction of the hero's \ref lua_api_movement "movement".

2.14.4.3.3 hero:set_direction(direction4)

Sets the direction of the hero's sprites.

- `direction4` (number): The direction of the hero's sprites, between `0` (East) and 3 (South).

Remarks

The direction of the hero's sprites may be different from both the direction pressed by the player's commands" and from the actual direction of the hero's \ref lua_api_movement "movement".

2.14.4.3.4 hero:get_walking_speed()

Returns the speed of the normal walking movement of hero.

This speed is automatically reduced when the hero walks on special ground like grass, ladders or holes.

- Return value (number): The speed of normal walk in pixels par second.

2.14.4.3.5 hero:set_walking_speed(walking_speed)

Sets the speed of the normal walking movement of hero.

The default walking speed is 88 pixels per second. This speed is automatically reduced when the hero walks on special ground like grass, ladders or holes.

- `walking` speed (number): The speed of normal walk in pixels par second.

2.14.4.3.6 hero:save_solid_ground([x, y, layer]), hero:save_solid_ground(callback)

Sets a position to go back to if the hero falls into a hole or other bad ground.

This replaces the usual behavior which is going back to the last solid position before the fall.

The position can be specified either as coordinates and a layer, or as a function that returns coordinates and a layer. Using a function allows to decide the position at the last moment.

To memorize a position directly:

- `x` (number, optional): X coordinate to memorize (no value means the current position).
- `y` (number, optional): Y coordinate to memorize (no value means the current position).
- `layer` (number, optional): Layer to memorize (no value means the current position).

To set a function that indicates the position to go back to:

- `callback` (function): A function to be called whenever the hero falls into bad ground. The function should return 2 or 3 values: x, y and optionally the layer (no layer value means keeping the layer unchanged). A `nil` value unsets any position or function that was previously set.

2.14.4.3.7 hero:reset_solid_ground()

Forgets a position that was previously memorized by hero:save_solid_ground() (if any).

The initial behavior is restored: the hero will now get back to where he was just before falling, instead going to of a memorized position.

This is equivalent to `hero:save_solid_ground(nil)`.

2.14.4.3.8 hero:get_solid_ground_position()

Returns the position where the hero gets back if he falls into a hole or other bad ground now.

This is the position that was previously memorized by the last call to hero:save_solid_ground(), if any. If the position was passed to hero:save_solid_ground() as a function, then this function is called to get a position.

Otherwise, this is the position of the hero the last time he was on solid ground.

- Return value 1 (number): X coordinate where to get back to solid ground, or `nil` if the hero never went on solid ground on this map yet.
- Return value 2 (number): Y coordinate where to get back to solid ground, or no value if the hero never went on solid ground on this map yet.
- Return value 3 (number): Layer where to get back to solid ground, or no value if the hero never went on solid ground on this map yet.

2.14.4.3.9 hero:get_animation()

Returns the current animation of the hero's sprites.

The hero may have several sprites (see hero:set_animation(). This function always returns the animation of the tunic sprite.

- Return value (string): The animation name of the tunic sprite.

2.14.4.3.10 hero:set_animation(animation, [callback])

Changes the animation of the hero's sprites.

The hero has several sprites, that are normally displayed or not depending on his state and his abilities:

- the body (the tunic),
- the shield,
- the sword,
- the sword stars,
- the trail of dust (by default, only shown when running).

The animation of all these sprites is usually managed by the engine to represent the current state of the hero. You can use this function to customize the animation. This allows you to implement custom actions that are not provided by the built-in states. Make sure that you don't call this function when the hero is not already doing a particular action (like pushing something or attacking), unless you know what you are doing.

All sprites of the hero that have an animation with the specified name take the animation. The ones that don't have such an animation are not displayed.

- `animation` (string): Name of the animation to set to hero sprites.
- `callback` (function, optional): A function to call when the animation ends (on the tunic sprite). If the animation loops, or is replaced by another animation before it ends, then the function will never be called.

2.14.4.3.11 hero:get_tunic_sprite_id()

Returns the name of the sprite representing the hero's body.

- Return value (string): The sprite animation set id of the hero's tunic.

2.14.4.3.12 hero:set_tunic_sprite_id(sprite_id)

Changes the sprite representing the hero's body.

By default, the sprite used for the body is `"hero/tunicX"`, where X is the tunic level.

You can use this function if you want to use another sprite.

- `sprite_id` (string): The sprite animation set id of the hero's tunic.

2.14.4.3.13 hero:get_sword_sprite_id()

Returns the name of the sprite representing the hero's sword.

- Return value (string): The sprite animation set id of the hero's sword.

2.14.4.3.14 hero:set_sword_sprite_id(sprite_id)

Changes the sprite representing the hero's sword.

By default, the sprite used for the sword is `"hero/swordX"`, where X is the sword level, or no sprite if the sword level is `0`.

You can use this function if you want to use another sprite.

- `sprite_id` (string): The sprite animation set id of the hero's sword. An empty string means no sword sprite.

2.14.4.3.15 hero:get_sword_sound_id()

Returns the name of the sound played when the hero uses the sword.

- Return value (string): The sound id of the hero's sword.

2.14.4.3.16 hero:set_sword_sound_id(sound_id)

Changes the sound to play when the hero uses the sword.

By default, the sound used for the sword is `"swordX"`, where X is the sword level, or no sound if the sword level is `0`.

You can use this function if you want another sound to be played.

- `sound_id` (string): The sound id of the hero's sword. An empty string means no sword sound.

2.14.4.3.17 hero:get_shield_sprite_id()

Returns the name of the sprite representing the hero's shield.

- Return value (string): The sprite animation set id of the hero's shield.

2.14.4.3.18 hero:set_shield_sprite_id(sprite_id)

Changes the sprite representing the hero's shield.

By default, the sprite used for the shield is `"hero/shieldX"`, where X is the shield level, or no sprite if the shield level is `0`.

You can use this function if you want to use another sprite.

- `sprite_id` (string): The sprite animation set id of the hero's shield. An empty string means no shield sprite.

2.14.4.3.19 hero:is_invincible()

Returns whether the hero is currently invincible.

The hero is temporarily invincible after being hurt or after you called hero:set_invincible(). In this situation, enemies cannot attack the hero, but you can still hurt him manually with hero:start_hurt().

- Return value (boolean): `true` if the hero is currently invincible.

2.14.4.3.20 hero:set_invincible([invincible, [duration]])

Sets or unsets the hero temporarily invincible.

When the hero is invincible, enemies cannot attack him, but you can still hurt him manually with hero:start_hurt().

- `invincible` (boolean, optional): `true` to make the hero invincible, or `false` to stop the invincibility. No value means `true`.
- `duration` (number, optional): Duration of the invincibility in milliseconds. Only possible when you set `invincible` to `true`. No value means unlimited.

2.14.4.3.21 hero:is_blinking()

Returns whether the hero's sprites are currently blinking.

The sprites are temporarily blinking after the hero was hurt or after you called hero:set_blinking().

- Return value (boolean): `true` if the hero's sprite are currently blinking.

Remarks

The visibility property of the hero is independent from this. Even when the sprites are blinking, the result of hero:is_visible() is unchanged.

2.14.4.3.22 hero:set_blinking([blinking, [duration]])

Makes the hero's sprites temporarily blink or stop blinking.

This only affects displaying: see hero:set_invincible() if you also want to make the hero invincible.

- `blinking` (boolean, optional): `true` to to make the sprites blink, or `false` to stop the blinking. No value means `true`.
- `duration` (number, optional): Duration in milliseconds before stopping the blinking. Only possible when you set `blinking` to `true`. No value means unlimited.

2.14.4.3.23 hero:get_state()

Returns the name of the current built-in state of the hero.

- Return value (`string`): The current state. Can be one of: `"back to solid ground"`, `"boomerang"`, `"bow"`, `"carrying"`, `"falling"`, `"forced walking"`, `"free"`, `"frozen"`, `"grabbing"`, `"hookshot"`, `"hurt"`, `"jumping"`, `"lifting"`, `"plunging"`, `"pulling"`, `"pushing"`, `"running"`, `"stairs"`, `"stream"`, `"swimming"`, `"sword loading"`, `"sword spin attack"`, `"sword swinging"`, `"sword tapping"`, `"treasure"`, `"using item"` or `"victory"`.

2.14.4.3.24 hero:freeze()

Prevents the player from moving the hero until you call hero:unfreeze().

2.14.4.3.25 hero:unfreeze()

Restores the control to the player. The control may have been lost for example by a call to hero:freeze() or to some_movement:start(hero).

2.14.4.3.26 hero:walk(path, [loop, [ignore_obstacles]])

Makes the hero move with the specified path and a walking animation. The player cannot control him during the movement.

- `path` (string): The path as a string sequence of integers. Each value is a number between 0 and 7 that represents a step (move of 8 pixels) in the path. 0 is East, 1 is North-East, etc.
- `loop` (boolean, optional): `true` to repeat the path once it is done (default `false`).
- `ignore_obstacles` (boolean, optional): `true` to allow the hero to traverse obstacles during this movement (default `false`). Make sure the movement does not end inside an obstacle.

2.14.4.3.27 hero:start_jumping(direction8, distance, [ignore_obstacles])

Makes the hero jump towards the specified direction.

- `direction8` (number): Direction of the jump, between `0` and `7` (see jump_movement:set_direction8(direction8) jump_movement:set_direction8()).
- `distance` (number): Distance of the jump in pixels (see jump_movement:set_distance(distance) jump_movement:set_distance()).
- `ignore_obstacles` (boolean, optional): `true` to allow the hero to traverse obstacles during this movement (default `false`). Make sure the movement does not end inside an obstacle.

2.14.4.3.28 hero:start_attack()

Makes the hero perform his main attack (swinging his sword).

This function does the same as what happens when the player presses the `"attack"` game command. You can use it to trigger the attack from your script instead of from a game command.

If the player is not allowed to perform the attack now (because he does not have the sword ability or because the hero is currently busy in another state), then nothing happens.

2.14.4.3.29 hero:start_item(item)

Makes the hero use an equipment item.

The item:on_using() event will be called and the player won't be able to control the hero until you call item:set_finished(). See the documentation of equipment items for more information.

This function does the same as what happens when the player presses a game command corresponding to this equipment item. You can use it to trigger the item from your script instead of from a game command.

If the player is not allowed to use the item now (because he does not have it, because the item cannot be used explicitly, or because the hero is currently busy in another state), then nothing happens.

- `item` (item): The equipment item to start using.

2.14.4.3.30 hero:start_treasure(treasure_item, [treasure_variant, [treasure_savegame_variable, [callback]]])

Gives a treasure to the player. The hero will brandish the treasure.

- `treasure_name` (string, optional): Kind of treasure to give (the name of an equipment item). The treasure must be an obtainable item.
- `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
- `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this treasure is found. No value means that the state of the treasure is not saved. It is allowed (though strange) to give the same saved treasure twice.
- `callback` (function, optional): A function to call when the treasure's dialog finishes.

2.14.4.3.31 hero:start_victory([callback])

Makes the hero brandish his sword for a victory.

- `callback` (function, optional): A function to call when the victory sequence finishes. If you don't define it, the default behavior is to restore control to the player. If you define it, you can do other things, like teletransporting the hero somewhere else. To restore the control to the player, call hero:unfreeze().

2.14.4.3.32 hero:start_boomerang(max_distance, speed, tunic_preparing_animation, sprite_name)

Makes the hero shoot a boomerang.

- `max_distance` (number): Maximum distance of the boomerang's movement in pixels.
- `speed` (number): Speed of the boomerang's movement in pixels per second.
- `tunic_preparing_animation` (string): Name of the animation that the hero's tunic sprite should take while preparing the boomerang.
- `sprite_name` (string): Sprite animation set to use to draw the boomerang then.

2.14.4.3.33 hero:start_bow()

Makes the hero shoot an arrow with a bow.

2.14.4.3.34 hero:start_hookshot()

Makes the hero throw a hookshot.

2.14.4.3.35 hero:start_running()

Makes the hero run.

2.14.4.3.36 hero:start_hurt(source_x, source_y, damage)

Hurts the hero, exactly like when he is touched by an enemy. The hurting animations and sounds of the hero are played.

This method hurts the hero even if enemies cannot, including when the hero is temporarily invincible (because he was already hurt recently) or when the hero is in a state where he cannot be hurt (for example when he is brandishing a treasure).

- `source_x` (number): X coordinate of whatever hurts the hero. Used to push the hero away from that source.
- `source_y` (number): Y coordinate of whatever hurts the hero. Used to push the hero away from that source.
- `damage` (number): Base number of life points to remove (possibly `0`). This number will be divided by the tunic level of the player, unless you override this default calculation in hero:on_taking_damage().

Remarks

If you just want to remove some life to the player, without making the hurting animations and sounds of the hero, see game:remove_life().

2.14.4.3.37 hero:start_hurt([source_entity, [source_sprite]], damage)

Same as hero:start_hurt(source_x, source_y, damage), but specifying the source coordinates as an optional entity and possibly its sprite.

- `source_entity` (map entity, optional): Whatever hurts the hero. The coordinates of this source entity are used to push the hero away from that source. No value means that the hero will not be pushed away.
- `source_sprite` (sprite, optional): Which sprite of the source entity is hurting the hero. If you set this value, the hero will be pushed away from the origin of this sprite instead of from the origin of the source entity. Most of the time, you don't need to set this parameter.
- `damage`: Base number of life points to remove (possibly `0`). This number will be divided by the tunic level of the player, unless you override this default calculation in hero:on_taking_damage().

2.14.4.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

The hero is a particular map entity. Therefore, he inherits all events from the type map entity.

See Events of all entity types to know these events.

2.14.4.5 Events of the type hero

The following events are specific to the hero.

Recall that the hero persists when the player goes to another map, and so do the events defined on the hero.

2.14.4.5.1 hero:on_state_changed(state)

Called when the built-in state of the hero changes.

- `state` (string): Name of the new state. See hero:get_state() for the list of possible states.

Remarks

This event is called even for the initial state of the hero, right after game:on_started(). This initial state is always "`free`".

2.14.4.5.2 hero:on_taking_damage(damage)

Called when the hero is hurt and should take damages.

This happens usually after a collision with an enemy or when you call hero:start_hurt().

This event allows you to override what happens when the hero takes damage. By default, if you don't define this event, the hero loses some life as follows. The life lost is the damage inflicted by the attacker divided by the tunic level of the player, with a minimum of `1` (unless the initial damage was already `0`).

You can define this event if you need to change how the hero takes damage, for example if you want the shield level to give better resistance to injuries.

- `damage` (number): Damage inflicted by the attacker, no matter if this was an enemy or a call to hero:start_hurt().

2.14.5 Tile

Tiles are the small fixed bricks that compose the map.

2.14.5.1 Overview

Each tile has a pattern coming from the tileset of the map. A tile has an adjustable size: you can repeat that pattern several times instead of placing many times the same tile on the map. Tiles can also overlap each other.

For some reasons explained below, tiles are not accessible from Lua. When you need to access a tile from Lua (typically, to make it appear or disappear), use a dynamic tile instead.

2.14.5.1.1 Tiles are designed for performance

Tiles are fixed: they are only declared in the map data file. Actually, most of them don't even exist at runtime. You cannot create them dynamically: unlike other types of map entities, there is no method `map:create_tile()`. You cannot access them dynamically either.

Most tiles don't exist individually at runtime, because the engine makes a special performance treatment to them. When the map is loaded, the engine builds once for all, for each layer, a big image of all tiles (except animated tiles that have to be displayed separately). It also builds once for all the obstacle property of every 8∗8 square that compose each layer. Then, drawing the tiles or detecting whether an entity collides with obstacle tiles is very fast. You can declare a huge number of tiles on your map without degrading performances at runtime.

Note

There exists situations where you want to create, remove, enable or disable a tile from your script. Dynamic tiles are made for that. Dynamic tiles are like normal tiles, except that they are not optimized as explained above: they do exist at runtime. You can access them from your script and call methods such as dynamic_tile:set_enabled().

2.14.6 Dynamic tile

Dynamic tiles are tiles that can be hidden, shown, created and deleted at runtime.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_dynamic_tile().

2.14.6.1 Overview

Dynamic tiles exist because normal tiles are optimized away at runtime. See Tiles are designed for performance for more details.

A typical usage of dynamic tiles is to make appear of disappear parts of the map because something happens: a puzzle is solved, an non-playing characters opens a path for you, water disappears, etc. Use dynamic_tile:set_enabled() to this end.

Remarks

Dynamic tiles can get enabled and disabled automatically to reflect the state of a door. If you give the appropriate name to your dynamic tile, the engine will enable or disable it depending on whether the door is open or closed. See the documentation of map:open_doors() for more information.

2.14.6.2 Methods inherited from map entity

Dynamic tiles are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.6.3 Methods of the type dynamic tile

The following methods are specific to dynamic tiles.

2.14.6.3.1 dynamic_tile:get_pattern_id()

Returns the id of the tile pattern of this dynamic tile.

- Return value (string): The tile pattern id in the tileset.

2.14.6.3.2 dynamic_tile:get_modified_ground()

Returns the ground defined by this dynamic tile on the map.

The presence of a dynamic tile can modify the ground of the map. This is determined by the ground property of the tile pattern.

- Return value (string): The ground defined by this dynamic tile. `"empty"` means that this dynamic tile does not modify the ground of the map. See map:get_ground() for the list of possible grounds.

2.14.6.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Dynamic tiles are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.6.5 Events of the type dynamic tile

None.

2.14.7 Teletransporter

A teletransporter is a detector that sends the hero to another place when he walks on it.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_↩ teletransporter().

2.14.7.1 Overview

A teletransporter can send the hero to one of the following kind of places.

- A destination entity, on the same map or on another one.
- The same coordinates on another map.
- The side of an adjacent map (only possible for a teletransporter placed on a side of the current map).

Teletransporters can have any size, but like all entities, their width and height must be multiples of 8 pixels. The minimum size if 16x16 pixels (the size of the hero).

Note

You can also teletransport the hero explicitly with hero:teleport().

2.14.7.2 Methods inherited from map entity

Teletransporters are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.7.3 Methods of the type teletransporter

The following methods are specific to teletransporters.

2.14.7.3.1 teletransporter:get_sound()

Returns the sound to be played when the hero uses this teletransporter.

- Return value (string): Id of the teletransporter's sound. `nil` means that no sound will be played.

2.14.7.3.2 teletransporter:set_sound(sound_id)

Sets the sound to be played when the hero uses this teletransporter.

- `sound_id` (string): Id of the teletransporter's sound. `nil` means that no sound will be played.

2.14.7.3.3 teletransporter:get_transition()

Returns the style of transition to play when the hero uses this teletransporter.

- Return value (string): The transition style. Can be one of:
  - `"immediate"`: No transition.
  - `"fade"`: Fade-out and fade-in effect.
  - `"scrolling"`: Scrolling between maps.

2.14.7.3.4 teletransporter:set_transition(transition_style)

Sets the style of transition to play when the hero uses this teletransporter.

- `transition_style` (string): The transition style. Can be one of:
  - `"immediate"`: No transition.
  - `"fade"`: Fade-out and fade-in effect.
  - `"scrolling"`: Scrolling between maps.

2.14.7.3.5 teletransporter:get_destination_map()

Returns the id of the destination map of this teletransporter.

- Return value (string): Id of the destination map.

2.14.7.3.6 teletransporter:set_destination_map(map_id)

Sets the destination map of this teletransporter.

- `map_id` (string): Id of the destination map to set.

2.14.7.3.7 teletransporter:get_destination_name()

Returns the name of the destination place on the destination map.

- Return value (string): Location on the destination map. Can be the name of a destination entity, the special value `"_same"` to keep the hero's coordinates, or the special value `"_side"` to place on hero on the corresponding side of an adjacent map (normally used with the scrolling transition style). `nil` means the default destination entity of the map.

2.14.7.3.8 teletransporter:set_destination_name(destination_name)

Sets the destination place on the destination map.

- `destination_name` (string): Location on the destination map. Can be the name of a destination entity, the special value `"_same"` to keep the hero's coordinates, or the special value `"_side"` to place on hero on the corresponding side of an adjacent map (normally used with the scrolling transition style). `nil` means the default destination entity of the map.

2.14.7.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Teletransporters are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.7.5 Events of the type teletransporter

The following events are specific to teletransporters.

2.14.7.5.1 teletransporter:on_activated()

Called when the user takes this teletransporter, just before the map closing transition starts.

2.14.8 Destination

A destination is a possible arrival place for teletransporters.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_↩ destination().

2.14.8.1 Overview

Destinations may either have a sprite or be invisible. They may also have a direction: in this case, the hero takes that direction when arriving on the destination. Otherwise, the hero keeps his current direction.

The size of a destination is the one of the hero (16x16 pixels).

2.14.8.2 Methods inherited from map entity

Destinations are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.8.3 Methods of the type destination

2.14.8.3.1 destination:get_starting_location_mode()

Returns whether this destination updates the starting location of the player when arriving on it. If yes, when the player restarts his game, he will restart at this destination. The default value is `"when_world_changes"`.

- Return value (string): The starting location mode. Can be one of:
  - `"when_world_changes"`: Updates the starting location if the current world has just changed when arriving to this destination.
  - `"yes"`: Updates the starting location.
  - `"no"`: Does not update the starting location.

2.14.8.3.2 destination:set_starting_location_mode(mode)

Sets whether this destination updates the starting location of the player when arriving on it. If yes, when the player restarts his game, he will restart at this destination. The default value is `"when world changes"`.

- `mode` (string): The starting location mode. Can be one of:
  - `"when_world_changes"`: Updates the starting location if the current world has just changed when arriving to this destination.
  - `"yes"`: Updates the starting location.
  - `"no"`: Does not update the starting location.

2.14.8.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Destinations are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.8.5 Events of the type destination

The following events are specific to destinations.

2.14.8.5.1 destination:on_activated()

Called when the hero arrives on this destination.

The map opening transition is about to start at this point.

He may come from a teletransporter, from hero:teleport() or from the saved starting location.

2.14.9 Pickable treasure

A pickable treasure is a treasure on the ground and that the hero can pick up.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_pickable().

Pickable treasures may also be dropped by enemies and by destructible entities.

2.14.9.1 Overview

The properties of a pickable treasure (like its sprite and its size) are essentially determined by the equipment item given by the treasure. The script of that equipment item can call methods of this API to dynamically tune the behavior of the pickable treasure.

2.14.9.1.1 Pickable treasure sprites

Two sprites for a pickable treasure are automatically created by the engine. You can access them like for any other entity, specifying their name in entity:get_sprite([name]).

- `"treasure"`: Main sprite representing the treasure. Its animation set is `"entities/items"`, with the animation of the treasure item's name and the direction of the treasure's variant. This is the default one in entity:get_sprite([name]).
- `"shadow"`: Optional shadow displayed under the treasure. Its animation set is `"entities/shadow"`, with the animation specified in item:set_shadow() (`nil` means no shadow sprite).

2.14.9.2 Methods inherited from map entity

Pickable treasures are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.9.3 Methods of the type pickable

2.14.9.3.1 pickable:has_layer_independent_collisions()

Returns whether this pickable treasure can detect collisions with entities even if they are not on the same layer.

By default, pickable treasures can only have collisions with entities on the same layer.

- Return value (boolean): `true` if this pickable treasure can detect collisions even with entities on other layers.

2.14.9.3.2 pickable:set_layer_independent_collisions([independent])

Sets whether this pickable treasure can detect collisions with entities even if they are not on the same layer.

By default, pickable treasures can only have collisions with entities on the same layer. For example, you can call this method if your pickable treasure is a flying object that should be able to be picked by the hero no matter his current layer.

- `independent` (boolean, optional): `true` to make this pickable treasure detect collisions even with entities on other layers. No value means `true`.

2.14.9.3.3 pickable:get_followed_entity()

Returns the entity (if any) followed by this pickable treasure.

Pickable treasures get automatically attached to entities like the boomerang or the hookshot when such entities collide with them. You can use this function to know if it happens.

- Return value (map entity): The entity this pickable treasure is attached to, or `nil` if the pickable treasure is free.

2.14.9.3.4 pickable:get_falling_height()

Indicates how high this pickable treasure falls from.

This depends on how the pickable treasure was created. If is was placed on the map initially, it does not fall at all (`0` is returned). If it appears when the hero lifts a destructible object, it falls from a low height. If it is dropped by an enemy, it falls from higher.

By default, the engine sets a movement that makes the pickable treasure bounce of a few pixels over the ground during a fraction of second. The number of pixels, the duration and the number of bounces of the movement depends on this height. If you want to override that movement, (by calling movement:start(pickable)), you may also want to make it dependent of the falling height.

- Return value (number): An integer indicating how high the pickable treasure falls from at creation time, between `0` (not falling at all) and `3` (falling from some high place).

2.14.9.3.5 pickable:get_treasure()

Returns the kind of treasure represented by this pickable treasure.

- Return value 1 (item): The equipment item of this treasure.
- Return value 2 (number): Variant of this equipment item (`1` means the first variant).
- Return value 3 (string): Name of the boolean value that stores in the savegame whether this pickable treasure is found. `nil` means that the treasure is not saved.

2.14.9.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Pickable treasures are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.9.5 Events of the type pickable

None.

2.14.10 Destructible object

A destructible object is an entity that can be cut or lifted by the hero and that may hide a pickable treasure.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_↩ destructible().

2.14.10.1 Overview

Destructible objects can be customized in various ways. You can allow the hero to lift them or to cut them. The ones that can be lifted may require a minimum level of the `"lift"` ability. Their size is always 16x16 pixels (like the hero).

2.14.10.2 Methods inherited from map entity

Destructible objects are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.10.3 Methods of the type destructible

The following methods are specific to destructible objects.

2.14.10.3.1 destructible:get_treasure()

Returns what pickable treasure this object will drop when being lifted, when being cut or when exploding.

- Return value 1 (string): Name of an equipment item. `nil` means no item (in this case, other return values are `nil` too).
- Return value 2 (number): Variant of this equipment item (`1` means the first variant).
- Return value 3 (string): Name of the boolean value that stores in the savegame whether the treasure dropped is found. `nil` means that the treasure is not saved.

2.14.10.3.2 destructible:set_treasure([item_name, [variant, [savegame_variable]]])

Sets the pickable treasure that this object will drop when being lifted, when being cut or when exploding.

- `item_name` (string, optional): Name of an equipment item. `nil` or no value means no item.
- `variant` (number, optional): Variant of this equipment item (`1` means the first variant). The default value is `1`.
- `savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the treasure dropped is found. `nil` or no value means that the treasure is not saved.

2.14.10.3.3 destructible:get_destruction_sound()

Returns the sound to be played when this object is cut or broken.

- Return value (string): Id of the destruction sound. `nil` means that no sound will be played.

2.14.10.3.4 destructible:set_destruction_sound(destruction_sound_id)

Sets the sound to be played when this object is cut or broken.

- `destruction_sound_id` (string): Id of the destruction sound. `nil` means that no sound will be played.

2.14.10.3.5 destructible:get_weight()

Returns the weight of this destructible object. The weight corresponds to the `"lift"` ability required to lift the object.

- Return value (number): The level of `"lift"` ability required. `0` allows the player to lift the object unconditionally. The special value `-1` means that the object can never be lifted.

2.14.10.3.6 destructible:set_weight()

Sets the weight of this destructible object. The weight corresponds to the `"lift"` ability required to lift the object.

- `weight` (number): The level of `"lift"` ability required. `0` allows the player to lift the object unconditionally. The special value `-1` means that the object can never be lifted.

2.14.10.3.7 destructible:get_can_be_cut()

Returns whether this object can be cut by the sword.

- Return value (boolean): `true` if this object can be cut by the sword.

2.14.10.3.8 destructible:set_can_be_cut(can_be_cut)

Sets whether this object can be cut by the sword.

- `can_be_cut` (boolean, optional): `true` to allow the player to cut this object with the sword. No value means `true`.

2.14.10.3.9 destructible:get_can_explode()

Returns whether this object explodes when it is hit or after a delay when it is lifted.

- Return value (boolean): `true` if this object can explode.

2.14.10.3.10 destructible:set_can_explode(can_explode)

Sets whether this object explodes when it is hit or after a delay when it is lifted.

- `can_explode` (boolean, optional): `true` to make the object able to explode. No value means `true`.

2.14.10.3.11 destructible:get_can_regenerate()

Returns whether this object regenerates after a delay when it is destroyed.

- Return value (boolean): `true` if this object can regenerate.

2.14.10.3.12 destructible:set_can_regenerate(can_regenerate)

Sets whether this object regenerates after a delay when it is destroyed.

- `can_regenerate` (boolean, optional): `true` to make the object able to regenerate. No value means `true`.

2.14.10.3.13 destructible:get_damage_on_enemies()

Returns the number of life points that an enemy loses when the hero throws this object at it.

- Return value (number): The number of life points to remove to an enemy hit by this object. `0` means that enemies will ignore this object.

2.14.10.3.14 destructible:set_damage_on_enemies(damage_on_enemies)

Sets the number of life points that an enemy loses when the hero throws this object at it.

- `damage_on_enemies` (number): The number of life points to remove to an enemy hit by this object. `0` means that enemies will ignore this object.

2.14.10.3.15 destructible:get_modified_ground()

Returns the ground defined by this destructible object on the map.

The presence of a destructible object can modify the ground of the map. The ground is usually `"wall"`, but it may sometimes be `"traversable"`, or for example `"grass"` to make the destructible object traversable too but with an additional grass sprite below the hero.

- Return value (string): The ground defined by this destructible object. See map:get_ground() for the list of possible grounds.

2.14.10.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Destructible objects a reparticular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.10.5 Events of the type destructible

The following events are specific to destructible objects.

2.14.10.5.1 destructible:on_looked()

Called when the hero looks this destructible object, that is, when the player presses the action key but is not allowed to lift the object.

By default, nothing happens in this case. You can for example show a dialog to give the player a hint like "This is too heavy".

Remarks

If you want to do the same action for all destructible objects of your game, use the metatable trick. Just define this event on the metatable of the destructible object type instead of each individual object, and it will be applied to all of them.

2.14.10.5.2 destructible:on_cut()

Called when the hero has just cut this destructible object.

2.14.10.5.3 destructible:on_lifting()

Called when the hero starts lifting this destructible object.

At this point, the hero is in state `"lifting"`. The animation `"lifting"` of his sprites is playing and the player cannot control the hero.

Remarks

This destructible object no longer exists (unless it can regenerate). It is replaced by a carried object with the same sprite.

2.14.10.5.4 destructible:on_exploded()

Called when this destructible object is exploding.

If destructible:get_can_explode() is `true`, the destructible object explodes when there is an explosion nearby or when the hero lifts it, after a delay.

2.14.10.5.5 destructible:on_regenerating()

Called when this destructible object regenerates.

If destructible:get_can_regenerate() is `true`, the destructible object regenerates after a delay when it was lifted or exploded.

2.14.11 Carried object

A carried object is a map entity that the hero is lifting, carrying or throwing.

2.14.11.1 Overview

A carried object is created automatically by the engine when the hero lifts a map entity. Map entities that can be lifted include destructible objects and bombs. The carried object takes the sprite and the features of the lifted entity it is created from.

The hero can then walk with his carried object and throw it. He can even go to another map: the carried object is preserved.

2.14.11.1.1 Carried object sprites

Two sprites for a carried object are automatically created by the engine. You can access them like for any other entity, specifying their name in entity:get_sprite([name]).

- `"main"`: Main sprite representing the carried object. Its animation set is the one of the original entity he was created from (like a destructible object), with the animation `"stopped"` or `"walking"` depending on the hero state, and the same direction as the hero. This is the default sprite in entity:get_sprite([name]).
- `"shadow"`: Shadow displayed under the carried object when thrown. Its animation set is `"entities/shadow"`, with the animation name `"big"`.

2.14.11.2 Methods inherited from map entity

Carried objects are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.11.3 Methods of the type carried object

None.

2.14.11.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Carried objects are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.11.5 Events of the type carried object

None.

2.14.12 Chest

A chest is a box that contains a treasure.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_chest().

2.14.12.1 Overview

A chest may contain a treasure or be empty.

When opening the chest, the following happens by default:

- If the chest contains a treasure, the engine automatically gives it to the player, unless the treasure is not obtainable.
- If the chest is empty or contains a non-obtainable treasure, then nothing happens.

This is the default behavior of opening a chest, and it can be redefined by your script in the chest:on_opened() event.

The state of a chest is either open or closed. When a chest is closed, its treasure (if any) is still inside and the hero can get it.

A chest appears initially open on the map if its state is saved and the corresponding boolean value is `true`. If is possible to save the state of a chest (open or closed) even if it contains no treasure.

2.14.12.2 Methods inherited from map entity

Chests are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.12.3 Methods of the type chest

The following methods are specific to chests.

2.14.12.3.1 chest:is_open()

Returns the state of this chest (open or closed).

- Return value (boolean): `true` if this chest is open, `false` if it is closed.

2.14.12.3.2 chest:set_open([open])

Sets the state of this chest (open or closed). If you close the chest, its treasure (as returned by chest:get_treasure()) is restored and can be obtained again later.

- `open` (boolean, optional): `true` to make the chest open, `false` to make it closed. No value means `true`.

2.14.12.3.3 chest:get_treasure()

Returns the treasure the player will obtain when opening this chest.

If the chest is already open, this function still works: it returns the treasure that was inside the chest before it was open.

- Return value 1 (string): Name of an equipment item. `nil` means that the chest is empty.
- Return value 2 (number): Variant of this equipment item (`1` means the first variant). `nil` means that the chest is empty.
- Return value 3 (string): Name of the boolean value that stores in the savegame whether the chest is open. `nil` means that the chest is not saved.

Remarks

If the treasure is a non-obtainable item, the hero will actually get no treasure when opening the chest.

2.14.12.3.4 chest:set_treasure([item_name, [variant, [savegame_variable]]])

Sets the treasure the player will obtain when opening this chest.

If the chest is already open, this function still works, it sets the treasure that will be put back in case you close the chest later.

- `item_name` (string, optional): Name of an equipment item. `nil` makes the chest empty.
- `variant` (number, optional): Variant of this equipment item (`1` means the first variant). The default value is `1`. Must be `nil` when `item_name` is `nil`.
- `savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the chest is open. `nil` means that the chest is not saved.

Remarks

If the treasure is a non-obtainable item, the hero will actually get no treasure when opening the chest.

2.14.12.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Chests are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.12.5 Events of the type chest

The following events are specific to chests.

2.14.12.5.1 chest:on_opened(treasure_item, treasure_variant, treasure_savegame_variable)

Called when the hero opens this chest.

At this point, if the chest is saved, then the engine has already set the corresponding savegame value to `true` (`treasure_savegame_variable`), no matter if this event is defined.

Then, if you don't define this event, by default, the engine gives the treasure to the player (if there is no treasure, then nothing else happens and the hero is automatically unfrozen).

Your script can define this event to customize what happens. By calling hero:start_treasure(), you can either give the chest's treasure or a treasure decided dynamically. Or you can do something else: show a dialog, play a sound, close the chest again, etc.

The hero is automatically frozen during the whole process of opening a chest. If you don't give him a treasure, then you have to unblock him explicitly by calling hero:unfreeze() when you want to restore control to the player.

- `treasure_item` (item): Equipment item in the chest, or `nil` if the chest is empty or contains a non-obtainable item.
- `treasure_variant` (number): Variant of the treasure or `nil` if the chest is empty or contains a non-obtainable item.
- `treasure_savegame_variable` (string): Name of the boolean value that stores in the savegame whether this chest is open, or `nil` if this chest is not saved.

2.14.13 Shop treasure

A shop treasure is a treasure that can be purchased by the hero for money.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_shop_treasure().

2.14.13.1 Overview

A shop treasure entity shows the sprite of the treasure and its price. Its size is always 32x32 pixels and other entities cannot traverse it.

The hero can interact with this a shop treasure to buy it. A dialog is then automatically shown and the player can choose to buy the item or not. If he buys the treasure, he obtains it and brandishes it (just like when opening a chest). If the treasure is saved, it disappears from the map, otherwise it stays and can be bought again.

Shop treasures allow to make shops very easily from the quest editor. They provide a built-in process that handles entirely the dialogs, the price and the treasure. This API does not provide much control on this process. If you need to implement more customized interactions, you can use a generalized NPC.

2.14.13.2 Dialogs of shop treasures

Dialogs of shop treasures need special care because everything is built-in with shop treasures, but dialogs are entirely customizable. You can skip this section if you are not making your own dialog box system.

Dialogs that are displayed when interacting with a shop treasure are quite elaborate because they have both a parameter (the price: a number) and a result (the decision of the player: a boolean). If you implement your own dialog box in Lua, which is recommended, you need to respect the mechanism described in this section for the particular dialog of shop treasures.

First, when the hero interacts with a shop treasure, a dialog shows the description of the treasure. There is nothing special about this dialog.

Then, a second dialog shows the price of the treasure and asks if the player wants to buy it. The player can accept or refuse. This second dialog has the id `"_shop.question"`.

Since the price of the treasure is only known at runtime (dialogs.dat cannot know it), you should put a special sequence in the text of the dialog instead (like `"$v"`), and replace this special sequence by the actual price at runtime. At runtime, the price is the `info` argument of the game:on_dialog_started() event, so you can perform the substitution there.

Furthermore, when the dialog finishes, you have to tell the engine the decision of the player. This value to return can be `true` to buy the treasure, or anything else to refuse and do nothing. Use the `status` parameter of game:stop_dialog(status) to indicate it.

Remarks

Recall that when the game:on_dialog_started() event is not defined, the engine uses by default a built-in dialog box with minimal features. This built-in dialog box works correctly with shop items, providing that the text of the `"_shop.question"` dialog has the expected format. It should have three lines. The `"$v"` sequence (if any) is subsituted by the actual price like suggested above, and the user can select one of the last two lines: the first one is yes and the second one is no.

2.14.13.3 Methods inherited from map entity

Shop treasures are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.13.4 Methods of the type shop treasure

None.

2.14.13.5 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Shop treasures are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.13.6 Events of the type shop treasure

The following events are specific to shop treasures.

2.14.13.6.1 shop_treasure:on_buying()

Called when the player is about to buy this treasure.

He already accepted to buy it and validated the dialog. At this point, you may cancel the deal for some reason by returning `false` (for example if you realize that the player has no bottle and therefore cannot buy the potion he wanted).

- Return value (boolean): `true` if the player is allowed to buy the treasure.

2.14.13.6.2 shop_treasure:on_bought()

Called when the player has just bought this treasure.

- Return value (boolean): `true` if the player is allowed to buy the treasure.

Remarks

This event is called right after the more general events item:on_obtaining() and map:on_obtaining_treasure(). Those two events are called no matter how the treasure is being obtained: from a chest, from a pickable treasure, from a shop treasure or explicitly with hero:start_treasure().

2.14.14 Enemy

An enemy is a bad guy that hurts the hero when touching him.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_enemy().

2.14.14.1 Overview

Enemies can exist in various breeds. Each breed corresponds to a model of enemy with its behavior, its sprites and its movements.

The script file `enemies/XXXX.lua` defines the enemy breed `XXXX`. This script is executed every time an enemy of that model is created. The corresponding Lua enemy object is passed as parameter of that script. Use the Lua notation `"..."` to get this parameter and store it into a regular variable.

Here is a basic example of script for the enemy breed `"tentacle"`, a simple model of enemy that just follows the hero.

```
-- First, we put the parameter into a variable called "enemy".
-- (In Lua, the notation "..." refers to the parameter(s) of the script.)
local enemy = ...

-- Called when the enemy was just created on the map.
function enemy:on_created()

  -- Define the properties of this enemy.
  self:set_life(1)
  self:set_damage(2)
  self:create_sprite("enemies/tentacle")
  self:set_size(16, 16)
  self:set_origin(8, 13)
end

-- Called when the enemy should start or restart its movement
-- (for example if the enemy has just been created or was just hurt).
function enemy:on_restarted()

  -- Create a movement that walks toward the hero.
  local m = sol.movement.create("target")
  m:set_speed(32)
  m:start(self)
end
```

Such a script is all what you need to define a model of enemy. The engine handles for you the detection of collisions with the hero or his weapons, hurts the hero or the enemy when appropriate, removes the enemy when it gets killed, etc. But you can customize everything using the API described on this page, like what kind of attacks can hurt the enemy. You can also make complex enemies composed of several sprites, and set different behavior to each sprite. This is very useful to program a boss.

Timers are handy to script some repeated behavior on enemies, like performing a particular attack every 5 seconds. Create your timers from the enemy:on_restarted() event. Timers of an enemy are automatically destroyed when the enemy is hurt or immobilized, so that they don't get triggered during these special states. They are also destroyed when you call enemy:restart(). When enemy:on_restarted() is called, you are guaranteed that no timers exist on your enemy. Thus, it is safe to create them there.

Basic enemies often have the same behavior. To avoid duplication of code, you can factorize some code into a generic file and call it from each enemy breed script with `require()`, sol.main.do_file() or sol.main.load_file().

2.14.14.2 Methods inherited from map entity

Enemies are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.14.3 Methods of the type enemy

The following methods are specific to enemies.

2.14.14.3.1 enemy:get_breed()

Returns the breed (the model) of this enemy.

- Return value (string): The enemy's breed.

2.14.14.3.2 enemy:get_life()

Returns the current life of this enemy.

- Return value (number): Number of health points of the enemy.

2.14.14.3.3 enemy:set_life(life)

Sets the current life of this enemy.

The initial value is `1`.

- `life` (number): Number of health points to set. A value of `0` or lower kills the enemy.

2.14.14.3.4 enemy:add_life(life)

Adds some life to the enemy.

- `life` (number): Number of health points to add.

Remarks

Equivalent to `enemy:set_life(enemy:get_life() + life)`

2.14.14.3.5 enemy:remove_life(life)

Removes some life from the enemy.

- `life` (number): Number of health points to remove.

Remarks

Equivalent to `enemy:set_life(enemy:get_life() - life)`

2.14.14.3.6 enemy:get_damage()

Returns the number of life points that the enemy removes from the hero when touching him. This number will be divided by the level of resistance ability of the player (his tunic).

- Return value (number): Damage inflicted to the hero.

2.14.14.3.7 enemy:set_damage(damage)

Sets the number of life points that the enemy removes from the hero when touching him. This number will be divided by the tunic level of the player, unless you override this default calculation in hero:on_taking_damage().

The default value is `1`.

- `damage` (number): Damage inflicted to the hero.

2.14.14.3.8 enemy:is_pushed_back_when_hurt()

Returns whether the enemy is pushed away when it is hurt.

- Return value (boolean): `true` if the enemy is pushed away when hurt.

2.14.14.3.9 enemy:set_pushed_back_when_hurt([pushed_back_when_hurt])

Sets whether the enemy should be pushed away when it is hurt.

The default value is `true`.

- `pushed_back_when_hurt` (boolean, optional): `true` to make the enemy pushed away when hurt. No value means `true`.

2.14.14.3.10 enemy:get_push_hero_on_sword()

Returns whether the hero is pushed away when he hits this enemy with his sword.

- Return value (boolean): `true` if the hero is pushed away when hitting this enemy with his sword.

2.14.14.3.11 enemy:set_push_hero_on_sword([push_hero_on_sword])

Sets whether the hero should be pushed away when he hits this enemy with his sword.

The default value is `false`.

- `push_hero_on_sword` (boolean, optional): `true` to push the hero away when hitting this enemy with his sword. No value means `true`.

2.14.14.3.12 enemy:get_can_hurt_hero_running()

Returns whether this enemy can hurt the hero even when the hero is running.

- Return value (boolean): `true` if the hero can be hurt by this enemy even when running.

2.14.14.3.13 enemy:set_can_hurt_hero_running([can_hurt_hero_running])

Sets whether this enemy can hurt the hero even when the hero is running.

The default value is `false`.

- `can_hurt_hero_running` (boolean, optional): `true` so that the hero can be hurt by this enemy even when running. No value means `true`.

2.14.14.3.14 enemy:get_hurt_style()

Returns the style of sounds and animations to play when this enemy is hurt.

- Return value (string): `"normal"`, `"monster"` or `"boss"`.

2.14.14.3.15 enemy:set_hurt_style(hurt_style)

Sets the style of sounds and animations to play when this enemy is hurt. The default values is `"normal"`.

- `hurt_style` (string): `"normal"`, `"monster"` or `"boss"`.

2.14.14.3.16 enemy:get_can_attack()

Returns whether this enemy can currently attack the hero.

- Return value (boolean): `true` if the enemy can currently attack the hero.

2.14.14.3.17 enemy:set_can_attack([can_attack])

Sets whether this enemy can currently attack the hero.

When the enemy restarts after being hurt, `can_attack` is always set to `true`.

- `can_attack` (boolean, optional): `true` to allow the enemy to attack the hero. No value means `true`.

2.14.14.3.18 enemy:get_minimum_shield_needed()

Returns the level of protection (if any) that stops attacks from this enemy.

If the player has a protection ability greater than or equal to this value, he will stop attacks from this enemy if he is facing the direction of the enemy. The special value of `0` means that attacks cannot be stopped with the protection ability. Returns the required level of protection to stop attacks from this enemy.

- Return value (number): The level of protection that stops attacks from this enemy. A value of `0` means that the hero cannot stop the attacks.

2.14.14.3.19 enemy:set_minimum_shield_needed(minimum_shield_needed)

Sets a level of protection that stops attacks from this enemy.

If the player has a protection ability greater than or equal to this value, he will stop attacks from this enemy if he is facing the direction of the enemy. The special value of `0` means that attacks cannot be stopped with the protection ability. The default value is `0`.

- `minimum_shield_needed` (number): The level of protection that stops attacks from this enemy. A value of `0` means that the hero cannot stop the attacks.

2.14.14.3.20 enemy:get_attack_consequence(attack)

Returns how this enemy reacts when he receives an attack.

Recall that enemies may have several sprites. This attack consequence applies to all sprites of the enemy, unless you override some of them with enemy:set_attack_consequence_sprite().

- `attack` (string): Name of an attack against the enemy: `"sword"`, `"thrown_item"`, `"explosion"`, `"arrow"`, `"hookshot"`, `"boomerang"` or `"fire"`.
- `consequence` (number or string): Indicates what happens when this enemy receives the attack. It may be:
  - A positive integer: The enemy is hurt and loses this number of life points. In the particular case of a sword attack, this number will by default be increased by the level of the sword (see enemy:on_hurt_by_sword()).
  - `"ignored"`: Nothing happens. The weapon (if any) traverses the enemy.
  - `"protected"`: The enemy stops the attack. An attack failure sound is played.
  - `"immobilized"`: The enemy is immobilized for a few seconds.
  - `"custom"`: Event enemy:on_custom_attack_received() is called.

2.14.14.3.21 enemy:set_attack_consequence(attack, consequence)

Sets how this enemy reacts when he receives an attack.

Recall that enemies may have several sprites. This attack consequence applies to all sprites of the enemy, unless you override some of them with enemy:set_attack_consequence_sprite().

- `attack` (string): Name of an attack against the enemy: `"sword"`, `"thrown_item"`, `"explosion"`, `"arrow"`, `"hookshot"`, `"boomerang"` or `"fire"`.
- `consequence` (number or string): Indicates what happens when this enemy receives the attack. The possible values are the same as in enemy:get_attack_consequence().

2.14.14.3.22 enemy:get_attack_consequence_sprite(sprite, attack)

Returns how this enemy reacts when one of his sprites receives an attack.

This method returns the same result as enemy:get_attack_consequence(), unless you override the reaction of the enemy for a particular sprite with enemy:set_attack_consequence_sprite().

- `sprite` (sprite): A sprite of this enemy.
- `attack` (string): Name of an attack against the enemy: "sword", "thrown_item", "explosion", "arrow", "hookshot", "boomerang" or "fire".
- `consequence` (number or string): Indicates what happens when this sprite receives the attack. The possible values are the same as in enemy:get_attack_consequence().

2.14.14.3.23 enemy:set_attack_consequence_sprite(sprite, attack, consequence)

Sets how this enemy reacts when one of his sprites receives an attack.

This method overrides for a particular sprite the attack consequences defined by enemy:set_attack_consequence().

- `sprite` (sprite): A sprite of this enemy.
- `attack` (string): Name of an attack against the enemy: "sword", "thrown_item", "explosion", "arrow", "hookshot", "boomerang" or "fire".
- `consequence` (number or string): Indicates what happens when this sprite receives the attack. The possible values are the same as in enemy:get_attack_consequence().

2.14.14.3.24 enemy:set_default_attack_consequences()

Restores the default attack consequences for this enemy and its sprites.

2.14.14.3.25 enemy:set_default_attack_consequences_sprite(sprite)

Restores the default attack consequences for a particular sprite of this enemy.

- `sprite` (sprite): A sprite of this enemy.

2.14.14.3.26 enemy:set_invincible()

Makes this enemy ignore all attacks.

Equivalent to calling `lua_api_enemy_set_attack_consequence(attack, "ignored")` for each attack.

2.14.14.3.27 enemy:set_invincible_sprite(sprite)

Makes a sprite of this enemy ignore all attacks.

Equivalent to calling `lua_api_enemy_set_attack_consequence_sprite(sprite, attack, "ignored")` for each attack.

- `sprite` (sprite): A sprite of this enemy.

2.14.14.3.28 enemy:has_layer_independent_collisions()

Returns whether this enemy can detect collisions with entities even if they are not on the same layer.

By default, enemies can only have collisions with entities on the same layer.

- Return value (boolean): `true` if this enemy can detect collisions even with entities on other layers.

2.14.14.3.29 enemy:set_layer_independent_collisions([independent])

Sets whether this enemy can detect collisions with entities even if they are not on the same layer.

By default, enemies can only have collisions with entities on the same layer. If you set this property to `true`, this enemy will be able to hurt the hero even from a different layer.

- `independent` (boolean, optional): `true` to make this enemy detect collisions even with entities on other layers. No value means `true`.

2.14.14.3.30 enemy:get_treasure()

Returns the pickable treasure that will drop this enemy when killed.

- Return value 1 (string): Name of an equipment item. `nil` means no item dropped (in this case, other return values are `nil` too).
- Return value 2 (number): Variant of this equipment item (`1` means the first variant).
- Return value 3 (string): Name of the boolean value that stores in the savegame whether the treasure dropped is found. `nil` means that the treasure is not saved.

2.14.14.3.31 enemy:set_treasure([item_name, [variant, [savegame_variable]]])

Sets the pickable treasure that will drop this enemy when killed.

- `item_name` (string, optional): Name of an equipment item. `nil` or no value means no item.
- `variant` (number, optional): Variant of this equipment item (`1` means the first variant). The default value is `1`.
- `savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the treasure dropped is found. `nil` or no value means that the treasure is not saved.

2.14.14.3.32 enemy:is_traversable()

Returns whether this enemy can be traversed by other entities.

- Return value (boolean): `true` if this enemy is traversable.

2.14.14.3.33 enemy:set_traversable([traversable])

Sets whether this enemy can be traversed by other entities.

By default, the enemy is traversable. For example, if you want to prevent the hero to pass without killing the enemy, you can use this function to make the enemy become an obstacle. But make sure that the enemy's sprite is greater than its bounding box, otherwise, the enemy cannot touch the hero (and hurt him) anymore.

- `traversable` (boolean, optional): `true` to make this enemy traversable. No value means `true`.

2.14.14.3.34 enemy:get_obstacle_behavior()

Returns how the enemy behaves with obstacles.

- Return value (string): `"normal"`, `"flying"` or `"swimming"`.

2.14.14.3.35 enemy:set_obstacle_behavior(obstacle_behavior)

Sets how this enemy should behave with obstacles. The default value is `"normal"`. `"swimming"` allow the enemy to traverse water. `"flying"` allows the enemy to traverse holes, water and lava.

- `obstacle_behavior` (string): `"normal"`, `"flying"` or `"swimming"`.

2.14.14.3.36 enemy:set_size(width, height)

Sets the size of the bounding box of this enemy.

The default value is `16x16` pixels. This is the effective size used to detect obstacles when moving, but the sprite of the enemy may be larger, especially for a boss.

- `width` (number): Width of the enemy in pixels.
- `height` (number): Height of the enemy in pixels.

Remarks

Collisions with the hero are pixel-precise and use the sprite of the enemy, not his bounding box. Therefore, this function has no influence on collisions with the hero, but only on the detection of obstacles of the map when the enemy moves.

2.14.14.3.37 enemy:set_origin(origin_x, origin_y)

Sets the origin point of this enemy, relative to the upper left corner of its bounding box.

This origin point property allows entities of different sizes to have comparable reference points that can be used by the engine. Indeed, when two enemies overlap, the engine needs to determine which one has to be displayed first (it is always the one with the lowest Y coordinate). Sometimes, the engine also needs to compute an angle between two entities, for example to push away an enemy that was just hit. Using the upper left corner of their bounding box would not give the correct angle (unless both entities had the same size).

The origin point is also the point of synchronization of an entity with its sprites (because again, an entity that has a given size may have sprites with different sizes).

The default values is `8,13` and is usually okay for enemies of size `16x16`. See entity:get_origin() for more explanations about the origin point.

- `origin_x` (number): X coordinate of the origin point in pixels, relative to the upper left corner of the enemy's bounding box.
- `origin_y` (number): Y coordinate of the origin point in pixels, relative to the upper left corner of the enemy's bounding box.

2.14.14.3.38 enemy:restart()

Restarts this enemy.

This plays animation "walking" on its sprites, destroys any timer of the enemy and calls the event enemy:on_restarted().

This function has no effect if the enemy is dying.

2.14.14.3.39 enemy:hurt(life_points)

Hurts this enemy if he is currently vulnerable.

If the enemy is vulnerable, the hurting animation and sound start and the given number of life points are removed.

Nothing happens if the enemy is currently invulnerable (for example because he is already being hurt).

- life_points (number): Number of life points to remove from the enemy.

Remarks

If you just want to silently remove some life, call enemy:remove_life() instead.

2.14.14.3.40 enemy:immobilize()

Immobilizes this enemy for a while if possible.

After a few seconds, the enemy shakes and then restarts.

2.14.14.3.41 enemy:create_sprite(animation_set_id, [sprite_name])

Creates a sprite for this enemy.

- animation_set_id (string): Animation set to use for the sprite.
- sprite_name (string, optional): An optional name to identify the created sprite. Only useful for entities with multiple sprites (see entity:get_sprite()).
- Return value (sprite): The sprite created.

Remarks

If you don't create a sprite, your enemy will be invisible.

2.14.14.3.42 enemy:remove_sprite([sprite])

Removes and destroys a sprite of this enemy.

The sprite must have been created before by enemy:create_sprite().

- sprite (sprite): The sprite to remove. The default value is the first sprite that was created.

2.14.14.3.43 enemy:create_enemy(properties)

Creates another enemy on the map, specifying its coordinates as relative to the current enemy.

This function is similar to map:create_enemy() but the coordinates are relative to the current enemy, and the layer is the one of the current enemy by default.

- `properties` (table): A table that describes all properties of the enemy to create. Its key-value pairs must be:
  - `name` (string, optional): Name identifying the entity. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique. No value means no name.
  - `layer` (number, optional): The layer, between map:get_min_layer() and map:get_max_layer(). No value means the same layer as the current enemy.
  - `x` (number, optional): X coordinate on the map, relative to the current enemy. The default value is `0`.
  - `y` (number, optional): Y coordinate on the map, relative to the current enemy. The default value is `0`.
  - `direction` (number, optional): Initial direction of the enemy, between `0` (East) and `3` (South). The default value is `3`.
  - `breed` (string): Model of enemy to create.
  - `savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this enemy is dead. No value means that the enemy is not saved. If the enemy is saved and was already killed, then no enemy is created. Instead, its pickable treasure is created if it is a saved one.
  - `treasure_name` (string, optional): Kind of pickable treasure to drop when the enemy is killed (the name of an equipment item). If this value is not set, or corresponds to a non obtainable item, then the enemy won't drop anything.
  - `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
  - `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the pickable treasure of this enemy was obtained. No value means that the state of the treasure is not saved. If the treasure is saved and the player already has it, then the enemy won't drop anything.
- Return value (enemy or pickable treasure): The enemy created, except when it is a saved enemy that is already dead. In this case, if the enemy dropped a saved treasure that is not obtained yet, this pickable treasure is created and returned. Otherwise, `nil` is returned.

2.14.14.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Enemies are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.14.5 Events of the type enemy

The following events are specific to enemies.

2.14.14.5.1 enemy:on_update()

Called at each cycle while this enemy is alive.

Remarks

As this function is called at each cycle, it is recommended to use other solutions when possible, like timers and other events.

2.14.14.5.2 enemy:on_suspended(suspended)

Called when the map has just been suspended or resumed.

The map is suspended by the engine in a few cases, like when the game is paused or when the camera is being moved by a script. When this happens, all map entities stop moving and most sprites stop their animation.

- `suspended` (boolean): `true` if the map was just suspended, `false` if it was resumed.

2.14.14.5.3 enemy:on_created()

called when this enemy has just been created on the map.

2.14.14.5.4 enemy:on_enabled()

called when this enemy has just been enabled.

2.14.14.5.5 enemy:on_disabled()

called when this enemy has just been disabled.

2.14.14.5.6 enemy:on_restarted()

Called when this enemy should start or restart its movement and timers because something happened. For example, the enemy has just been created, or it was just hurt or immobilized, or you called enemy:restart(). If your enemy should move, this is the right place to create its movement.

Timers associated to the enemy were automatically destroyed. Thus, you should also recreate them from this event.

2.14.14.5.7 enemy:on_pre_draw()

Called just before the enemy is drawn on the map. You may display additional things below the enemy.

2.14.14.5.8 enemy:on_post_draw()

Called just after the enemy is drawn on the map. You may display additional things above the enemy.

2.14.14.5.9 enemy:on_collision_enemy(other_enemy, other_sprite, my_sprite)

Called when a sprite of this enemy overlaps another enemy's sprite.

- `other_enemy` (enemy): Another enemy.
- `other_sprite` (sprite): A sprite of that other enemy.
- `my_sprite` (sprite): A sprite of the current enemy.

2.14.14.5.10 enemy:on_custom_attack_received(attack, sprite)

Called when this enemy receives an attack with a custom effect.

This function is called if you have set consequence of the attack to `"custom"`. You have to define what happens, for example hurting the enemy, making a special reaction, etc.

- `attack` (string): The attack that was received: `"sword"`, `"thrown_item"`, `"explosion"`, `"arrow"`, `"hookshot"`, `"boomerang"` or `"fire"`.
- `sprite` (sprite): The sprite of this enemy that receives the attack, or `nil` if the attack does not come from a pixel-precise collision.

2.14.14.5.11 enemy:on_hurt_by_sword(hero, enemy_sprite)

Called when this enemy is successfully hurt by the sword of the hero.

You should define this event to customize the damage inflicted by the sword.

This event can only be called if the reaction to the `"sword"` attack is hurting the enemy.

At this point, the enemy is in the state of being hurt. His hurting animation and sound have just started. This is a good time to remove some life points with enemy:remove_life().

By default, if you don't define this event, the enemy loses a number of life points computed as his reaction to sword attacks multiplied by the sword ability level of the hero, and doubled during a spin attack.

- `hero` (hero): The hero who used the sword.
- `enemy_sprite` (sprite): The sprite of this enemy that was hit. You may use this information if your enemy has several sprites with different behaviors.

2.14.14.5.12 enemy:on_hurt(attack)

Called when this enemy is successfully hurt by any attack.

This event can only be called if the reaction to the attack is hurting the enemy.

At this point, the enemy is in the state of being hurt. His hurting animation and sound have just started and he has just lost some life.

- `attack` (string): The attack that was received: `"sword"`, `"thrown_item"`, `"explosion"`, `"arrow"`, `"hookshot"`, `"boomerang"` or `"fire"`.

Remarks

In the case of a `"sword"` attack, this event is called right after enemy:on_hurt_by_sword()

2.14.14.5.13 enemy:on_dying()

Called when the enemy's life comes to `0`.

When the life comes to `0`, the movement of the enemy is stopped, its timers are stopped too, the dying animation starts and a sound is played. The details of the dying animation and the sound played depend on the hurt style property. If the hurt style is `"enemy"` or `"monster"`, any sprite you created on the enemy is automatically removed and replaced by the sprite `"enemies/enemy_killed"`. If the hurt style is `"boss"`, your sprites continue to exist and to play animation "hurt", while explosions appear on the enemy.

In both cases, the enemy will be removed from the map when the dying animation ends.

Remarks

This event is called right after enemy:on_hurt().

2.14.14.5.14 enemy:on_dead()

Called when the enemy's dying animation is finished.

At this point, the enemy no longer exists on the map. In other words, enemy:exists() returns `false`, trying to get the enemy from its name returns `nil`, and functions like map:get_entities(prefix) won't find this enemy.

This means that you can safely use map:has_entities(prefix) from `enemy:on_dead()` to detect when all enemies with a common prefix are dead.

2.14.14.5.15 enemy:on_immobilized()

Called when the enemy is immobilized.

2.14.14.5.16 enemy:on_attacking_hero(hero, enemy_sprite)

Called when the hero is successfully touched by this enemy.

This event is not called if the hero was protected by his shield, or if he currently cannot be hurt for some reason, like when he is already being hurt, when he is temporarily invincible, or when he is in a special state like brandishing a treasure.

Your script can define this event to customize what bad things happen to the hero. If you define this event, the engine does absolutely nothing and lets you handle this.

If you don't define this event, the hero is hurt with the predefined behavior as follows. The hero goes to the state `"hurt"` where is pushed away from the enemy. He loses some life depending on the enemy's damage property, and on the hero's tunic and on hero:on_taking_damage() if defined. Then, he recovers and his sprites blink for a while. During this short period, he is temporarily invincible.

- `hero` (hero): The hero being attacked.
- `enemy_sprite` (sprite): The sprite of the enemy that caused the collision with the hero. You may use this information if your enemy has several sprites with different behaviors.

Remarks

If you call hero:start_hurt(), you will obtain something equivalent to the default behavior. But if you don't, keep in mind that if the hero can still be hurt after your call, this event will continue to be called while there is a collision with the enemy. To avoid this, see for example hero:set_invincible() to make the hero temporarily invincible.

2.14.15 Non-playing character

A non-playing character (NPC) is somebody or something that the hero can interact with by pressing the action command or by using an equipment item just in front of it.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_↩ npc().

2.14.15.1 Overview

An NPC is not necessarily a person: it can also be a more general, solid interactive entity. In this case, it is called a generalized NPC.

Interactions with an NPC may start a dialog or be forwarded to Lua. More precisely, what happens can be one of the following.

- A dialog starts.
- A Lua event is called on the NPC itself, for instance npc:on_interaction(). or npc:on_interaction_item().
- A Lua event is called on an equipment item for instance item:on_npc_interaction(). or item:on_npc_↩ interaction_item().

An NPC can move if you apply a movement to it by calling movement:start(npc).

The size of an NPC is always 16x16 pixels (like the hero) and the NPC is an obstacle for most map entities, including the hero.

Usual NPCs (i.e. non-generalized ones) are suited to usual interactions with people. They must have a sprite with at least four directions, and with animations named `"stopped"` and (if you move them) `"walking"`. These predefined animations are automatically started by the engine when you make the NPC move. When the hero talks to them, their sprite automatically stop its animation and looks into his direction.

Example of use of a usual NPC: a non-playing character walking randomly in a town.

Generalized NPCs are highly customizable. They are solid entities that the hero can interact with, and there is essentially nothing more predefined by the engine. They may even have no sprite. You can choose to allow interactions with the hero from any of the four directions or from a precise direction only.

Example of use of a generalized NPC: a stone with something to read on it.

2.14.15.2 Methods inherited from map entity

Non-playing characters are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.15.3 Methods of the type non-playing character

The following methods are specific to non-playing characters.

2.14.15.3.1 npc:is_traversable()

Returns whether this NPC can be traversed by other entities.

By default, NPCs are not traversable. However, be aware that some entities can override this setting. Indeed, other NPCs, enemies and projectiles (thrown objects, arrows, boomerang) can traverse usual NPCs but cannot traverse generalized NPCs. And custom entities can have finer customization.

- Return value (boolean): `true` if this NPC is traversable.

2.14.15.3.2 npc:set_traversable([traversable])

Sets whether this NPC can be traversed by other entities.

By default, NPCs are not traversable. However, be aware that some entities can override this setting. Indeed, other NPCs, enemies and projectiles (thrown objects, arrows, boomerang) can traverse usual NPCs but cannot traverse generalized NPCs. And custom entities can have finer customization.

If you want to allow the hero to be able to traverse this NPC, you can use this function.

- `traversable` (boolean, optional): `true` to make this NPC traversable. No value means `true`.

2.14.15.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Non-playing characters are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.15.5 Events of the type non-playing character

The following events are specific to non-playing characters.

2.14.15.5.1 npc:on_interaction()

Called when the hero interacts (the player pressed the action command) in front of this NPC, if the NPC has the property to notify its own Lua script.

2.14.15.5.2 npc:on_interaction_item(item_used)

Called when the hero uses any equipment item (the player pressed an item command) with this NPC, if the NPC has the property to notify its own Lua script.

- `item_used` (item): The item currently used by the player.
- Return value (boolean): `true` if an interaction happened. If you return `false` or nothing, then item_used:on_using() will be called (just like if there was no NPC in front of the hero).

2.14.15.5.3 npc:on_collision_fire()

Called when fire touches this NPC, if the NPC has the property to notify its own Lua script.

2.14.16 Block

Blocks are solid map entities that may be pushed or pulled by the hero.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_block().

2.14.16.1 Overview

Blocks that can move may be pushable, pullable or both pushable and pullable. A block can be moved either once or have an unlimited number of moves. It can be moved to any direction (the four main directions) or to a specific direction only.

The size of a block is always 16x16 pixels (the size of the hero), but as usual, its sprite may be larger.

Blocks are normally always moved by steps of 16 pixels. Thus, they can stay aligned on the 8x8 grid of the map. However, they may get stopped in the middle of their movement if they collide with entities like enemies or non-playing characters. In this case, they lose their alignment on the grid, and this might be a problem because you often want the to move a block precisely through narrow places and place it at an exact position to solve a puzzle. To deal with this potential issue, the engine automatically realigns the block to the 8x8 grid the next time it is moved.

2.14.16.2 Methods inherited from map entity

Blocks are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.16.3 Methods of the type block

The following methods are specific to blocks.

2.14.16.3.1 block:reset()

Restores the block at its initial position and resets its counter of moves.

It means that the hero will be able to move again a block that could be moved only once and that already moved.

2.14.16.3.2 block:is_pushable()

Returns whether this block can be pushed.

This property is independent of whether or not the block was already moved its maximum number of times.

- Return value (boolean): `true` if this block can be pushed.

2.14.16.3.3 block:set_pushable([pushable])

Sets whether this block can be pushed.

This property is independent of whether or not the block was already moved its maximum number of times.

- `pushable` (boolean): `true` to make this block pushable. No value means `true`.

2.14.16.3.4 block:is_pullable()

Returns whether this block can be pulled.

This property is independent of whether or not the block was already moved its maximum number of times.

- Return value (boolean): `true` if this block can be pulled.

2.14.16.3.5 block:set_pullable([pullable])

Sets whether this block can be pulled.

This property is independent of whether or not the block was already moved its maximum number of times.

- `pullable` (boolean): `true` to make this block pullable. No value means `true`.

2.14.16.3.6 block:get_maximum_moves()

Returns a value indicating the maximum number of times the block can be moved.

This function returns the maximum moves value that was set at creation time or by block:set_maximum_moves(), no matter if the block was moved then.

- Return value (integer): How many times the block can be moved: `0`: none, `1`: once, `nil`: infinite.

2.14.16.3.7 block:set_maximum_moves(maximum_moves)

Sets the maximum number of times the block can be moved.

This resets the remaining allowed moves.

- `maximum_moves` (integer): How many times the block can be moved: `0`: none, `1`: once, `nil`: infinite.

2.14.16.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Blocks are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.16.5 Events of the type block

The following events are specific to blocks.

2.14.16.5.1 block:on_moving()

Called when the hero starts moving the block of a step.

2.14.16.5.2 block:on_moved()

Called when the hero has just moved this block of a step.

2.14.17 Jumper

A jumper is an invisible detector that makes the hero jump into one of the 8 main directions when touching it.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_jumper().

2.14.17.1 Overview

The hero makes a jump when touching the jumper. Properties of the jumper include the distance and the direction of the jump.

During the jump, the hero cannot be controlled by the player, and he can traverse obstacles. You need to make sure the destination of the jump is a valid place for when the control is restored to the player.

Think of a jumper like an horizontal, vertical or diagonal line (depending on the direction of the jump: one of the 8 main directions). This line has actually a thickness of 8 pixels so that it can be handled more easily in the editor, but this thickness does not really matter: the jump starts as soon as the hero touches the jumper.

2.14.17.2 Methods inherited from map entity

Jumpers are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.17.3 Methods of the type jumper

None.

2.14.17.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Jumpers are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.17.5 Events of the type jumper

None.

2.14.18 Switch

A switch is a button that can be activated to trigger a mechanism.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_switch().

2.14.18.1 Overview

A switch may be activated by the hero, by a block or by a projectile, depending on its subtype. The following subtypes of switches are available:

- Walkable, traversable pressure plate.
- Button to be activated by shooting an arrow on it with the bow.
- Solid switch to be activated with the sword or other weapons.

When a switch is activated, the event switch:on_activated() is called. Define that event to implement what happens: opening a door, showing a chest, etc.

Some switches get inactivated when the hero (or the entity that activated them) leaves them. In this case, when the switch is inactivated, the event switch:on_inactivated() is called.

The size of a switch is always 16x16 pixels. Its sprite can be defined at creation time, as well as the sound played when it gets activated.

2.14.18.2 Methods inherited from map entity

Switches are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.18.3 Methods of the type switch

The following methods are specific to switches.

2.14.18.3.1 switch:is_walkable()

Returns whether this is a walkable switch.

- Return value (boolean): `true` if this switch is a walkable one.

2.14.18.3.2 switch:is_activated()

Returns whether this switch is activated.

- Return value (boolean): `true` if this switch is currently activated.

2.14.18.3.3 switch:set_activated([activated])

Sets whether this switch is activated or not.

The change is quiet and immediate: no sound is played and no event is triggered.

- `activated` (boolean, optional): `true` to make the switch activated, `false` to make is inactivated. No value means `true`.

2.14.18.3.4 switch:is_locked()

Returns whether this switch is current locked.

When a switch is locked, its state cannot change anymore: it can no longer be activated or inactivated by other entities. However, it can still changed programmatically by calling switch:set_activated().

- Return value (boolean): `true` if this switch is currently activated.

2.14.18.3.5 switch:set_locked([locked])

Locks this switch in its current state or unlocks it.

When a switch is locked, its state cannot change anymore: it can no longer be activated or inactivated by other entities. However, it can still changed programmatically by calling switch:set_activated().

- `locked` (boolean, optional): `true` to lock the switch, `false` to unlock it. No value means `true`.

Remarks

The method switch:set_activated() works even on a locked switch.

2.14.18.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Switches are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.18.5 Events of the type switch

The following events are specific to switches.

2.14.18.5.1 switch:on_activated()

Called when this switch has just been turned on.

This is the right place to define the action that you want your switch to perform.

2.14.18.5.2 switch:on_inactivated()

Called when a switch has just been turned off.

2.14.18.5.3 switch:on_left()

Called when an entity placed on a switch (like the hero or a block) has just left the switch, regardless of whether the switch was activated or not.

2.14.19 Sensor

A sensor is an invisible detector that triggers something when the hero overlaps it.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_↩ sensor().

2.14.19.1 Overview

Unlike switches, sensors cannot be avoided by a jump. If you want the hero to be detected for sure when he enters somewhere, use a sensor rather than an invisible switch.

When a sensor is activated, the event sensor:on_activated() is called. Define that event to implement what happens: closing a door, starting a dialog, etc.

Sensors can have any size, but like all entities, their width and height must be multiples of 8 pixels. The minimum size if 16x16 pixels (the size of the hero).

A sensor is activated only if the bounding box of the hero is entirely on the sensor.

2.14.19.2 Methods inherited from map entity

Sensors are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.19.3 Methods of the type sensor

None.

2.14.19.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Sensors are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.19.5 Events of the type sensor

The following events are specific to sensors.

2.14.19.5.1 sensor:on_activated()

Called when the hero overlaps this sensor.

The bounding box of the hero (of size 16x16 pixels) must fit entirely the sensor. This means that if the sensor has a size of 16x16, the hero and the sensor must overlap perfectly.

This event is the right place to define the action that you want your sensor to perform.

2.14.19.5.2 sensor:on_activated_repeat()

Called continuously while the hero overlaps this sensor.

2.14.19.5.3 sensor:on_left()

Called when the hero stops overlapping this sensor.

2.14.19.5.4 sensor:on_collision_explosion()

Called when an explosion touches this sensor.

2.14.20 Separator

Separators allow to visually separate different regions of a map like if there was several maps.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_separator().

2.14.20.1 Overview

A separator must be a horizontal bar with a height of 16 pixels, or a vertical bar with a width of 16 pixels. The separation is made in the middle of this bar: in other words, 8 pixels belong to each side.

When the camera touches the separation, it stops like if there was a limit of a map. If the hero touches the separation, he automatically scrolls to the other side.

It is your responsibility to make sure that your separators don't conflict. For example, if you make a separator too close to a limit of the map, or if you make two vertical separators too close to each other, it is impossible for the engine to respect both constraints. In such cases, the result will look weird. The good news is that you know the maximum size of the visible quest screen: it is a property of your quest, specified in (quest.dat). If the space between vertical separators is greater than the maximum quest width, you will be safe. The rule is the same for horizontal separators and the maximum quest height.

Enemies and other entities cannot cross separators. But you should take care of enemies whose behavior is to attack the hero when they are close to him or to shoot projectiles at him: if the enemy is close to the hero but on the other side of a separator, you probably don't want to perform the attack. Use entity:is_in_same_region() to determine whether the enemy is in the same region than the hero.

To get all entities in a given region, see map:get_entities_in_region().

2.14.20.2 Methods inherited from map entity

Separators are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.20.3 Methods of the type separator

None.

2.14.20.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Separators are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.20.5 Events of the type separator

The following events are specific to separators.

2.14.20.5.1 separator:on_activating(direction4)

Called when the camera starts traversing on this separator.

- `direction4` (number): Direction of the hero when traversing the separator, `beween` 0 (East) to `3` (South).

2.14.20.5.2 separator:on_activated(direction4)

Called when the camera has finished traversing this separator.

The hero is now on the other side.

- `direction4` (number): Direction of the hero when traversing the separator, `beween` 0 (East) to `3` (South).

2.14.21 Wall

A wall is an invisible obstacle that stops some specific types of map entities.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_↩ wall().

2.14.21.1 Overview

If you want to prevent a kind of entity to leave a delimited area while allowing others to pass, you can use walls.

For example, at the entrance of a village, you can make a wall that blocks enemies enemies and lets the hero pass.

Walls can have any size, but like all entities, their width and height must be multiples of 8 pixels.

2.14.21.2 Methods inherited from map entity

Walls are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.21.3 Methods of the type wall

None.

2.14.21.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Walls are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.21.5 Events of the type wall

None.

2.14.22 Crystal

A crystal is a switch that lowers or raises alternatively some special colored blocks in the ground called crystal blocks.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_↩ crystal().

2.14.22.1 Overview

A crystal is essentially a solid switch that inverts theconfiguration" of \ref lua_api_crystal_block "crystal blocks" when activated.

2.14.22.1.1 Crystal sprites

Two sprites for a crystal are automatically created by the engine. You can access them like for any other entity, specifying their name in entity:get_sprite([name]).

- `"main"`: Main sprite representing the crystal. Its animation set is `"entities/crystal"`. This is the default one in entity:get_sprite([name]).
- `"star"`: Star twinkling over the crystal. Its animation set is `"entities/star"`.

2.14.22.2 Methods inherited from map entity

Crystals are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.22.3 Methods of the type crystal

None.

2.14.22.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Crystals are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.22.5 Events of the type crystal

None.

2.14.23 Crystal block

A crystal block is a colored low wall that may be raised or lowered in the ground.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_crystal_block().

2.14.23.1 Overview

An entity of type crystal block is a pattern of size 16x16 pixels that may be repeated horizontally and vertically like a tile.

When they are lowered in the ground, crystal blocks are traversable. When they are raised, they become obstacles. If the hero overlaps them while they get raised, then he can walk on them.

Crystal blocks exists in two colors. One of them is initially lowered and the other is initially raised. The state of crystal blocks is swapped when the hero activates a crystal or when you call map:set_change_crystal_state().

This state persists accross maps of the same world. It is reset when the world changes and when the savegame is reloaded.

2.14.23.2 Methods inherited from map entity

Crystals are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.23.3 Methods of the type crystal

None.

2.14.23.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Crystals are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.23.5 Events of the type crystal

None.

2.14.24 Stream

When walking on a stream, the hero automatically moves into one of the eight main directions.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_stream().

2.14.24.1 Overview

A stream is a map entity of size 16x16 pixels (the size of the hero). When the hero overlaps a significant part of it, he moves into a specific direction (one of the 8 main directions).

During this process, the player may or may not continue to control the hero, use his sword and his equipment items, depending on the stream properties.

The hero is not the only map entity that can follow a stream: Bombs also have this ability.

2.14.24.2 Streams and holes

If you make a stream that moves toward a hole or other bad ground, it is your responsibility to make sure to call hero:save_solid_ground() before (typically, when entering the room), otherwise the hero will reappear on the stream, one pixel before the hole and he will fall again repeatedly.

2.14.24.3 Methods inherited from map entity

Streams are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.24.4 Methods of the type stream

The following methods are specific to streams.

2.14.24.4.1 stream:get_direction()

Returns the direction of this stream.

This direction will be applied to entities that follow the stream.

- Return value (number): The direction between `0` (East) and `7` (South-East).

2.14.24.4.2 stream:set_direction(direction)

Sets the direction of this stream.

This direction will be applied to entities that follow the stream.

- `direction` (number): The direction to set, between `0` (East) and `7` (South-East).

2.14.24.4.3 stream:get_speed()

Returns the speed applied by this stream.

- Return value (number): The speed of the stream in pixels per second.

2.14.24.4.4 stream:set_speed(speed)

Sets the speed applied by this stream.

- `speed` (number): The speed to set in pixels per second. The default value is `40`.

2.14.24.4.5 stream:get_allow_movement()

Returns whether the player can still move the hero while being on this stream.

- Return value (boolean): `true` if the player can still move, `false` if this is a blocking stream.

2.14.24.4.6 stream:set_allow_movement(allow_movement)

Sets whether the player can still move the hero while being on this stream.

- `allow_movement` (boolean): `true` to allow the player to move, `false` to make a blocking stream. No value means `true`.

2.14.24.4.7 stream:get_allow_attack()

Returns whether the player can still use the sword while being on this stream.

- Return value (boolean): `true` if the player use his sword.

2.14.24.4.8 stream:set_allow_attack(allow_attack)

Sets whether the player can still use the sword while being on this stream.

- `allow_attack` (boolean): `true` to allow the player to use the sword. No value means `true`.

2.14.24.4.9 stream:get_allow_item()

Returns whether the player can still use equipment items while being on this stream.

- Return value (boolean): `true` if the player can still use equipment itens.

2.14.24.4.10 stream:set_allow_item(allow_item)

Sets whether the player can still use equipment items while being on this stream.

- `allow_item` (boolean): `true` to allow the player to use equipment items. No value means `true`.

2.14.24.5 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Streams are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.24.6 Events of the type stream

None.

2.14.25 Door

A door is an obstacle that can be opened by Lua, and optionally by the hero under some conditions.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_door().

2.14.25.1 Overview

A door may have four states: open, closed, opening or closing. States opening and closing are transitional states used to play a door opening or closing animation on the door's sprite. The sprite of a door must define animations `"open"` and `"closed"`. Animations `"opening"` and `"closing"` are optional: if they don't exist, the door won't use states opening and closing. If they exist, be aware that after you call a function like map:open_doors(), your door will first be in state opening (for the duration of its sprite `"opening"` animation), and only after that, will get in state open. In other words, door:is_open() does not return true as soon as you open the door, unless the sprite has no animation `"opening"`.

Doors often work by pairs. Indeed, when you have two adjacent rooms on the same map, you normally use two door entities. They can be linked by setting a common prefix to their name. Then, you can use map:open_doors() and map:close_doors() to handle both of them at the same time more easily. Additionaly, dynamic tiles whose names also have this prefix and end by `_open` or `_closed` are automatically enabled or disabled, respectively. This helps a lot to make tiles follow harmoniously the state of doors. See map:open_doors() for more information about these mechanisms.

A door may be opened by one of the following methods:

- `"none"`: Cannot be opened by the player. You can only open it explicitly by calling map:open_doors().
- `"interaction"`: Can be opened by pressing the action command in front of it.
- `"interaction_if_savegame_variable"`: Can be opened by pressing the action command in front of it, provided that a specific savegame variable is set.
- `"interaction_if_item"`: Can be opened by pressing the action command in front of it, provided that the player has a specific equipment item.
- `"explosion"`: Can be opened by an explosion.

For doors whose opening method is `"interaction_if_savegame_variable"` or `"interaction_if_item"`, you can specify which savegame variable or equipment item is required. You can also choose whether opening the door should consume the savegame variable of equipment item that was required.

2.14.25.2 Methods inherited from map entity

Doors are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.25.3 Methods of the type door

The following methods are specific to doors.

2.14.25.3.1 door:is_open()

Returns whether this door is open.

- Return value (boolean): `true` if this door is open, `false` if it is opening, closed or closing.

2.14.25.3.2 door:is_opening()

Returns whether this door is being opened.

- Return value (boolean): `true` if this door is being opened, `false` if it is open, closed or closing.

2.14.25.3.3 door:is_closed()

Returns whether this door is closed.

- Return value (boolean): `true` if this door is closed, `false` if it is closing, open or opening.

2.14.25.3.4 door:is_closing()

Returns whether this door is being closed.

- Return value (boolean): `true` if this door is being closed, `false` if it is closed, open or opening.

2.14.25.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Doors are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.25.5 Events of the type door

The following events are specific to doors.

2.14.25.5.1 door:on_opened()

Called when this door starts being opened.

2.14.25.5.2 door:on_closed()

Called when this door starts being closed.

2.14.26 Stairs

Stairs make fancy animations, movements and sounds when the hero takes them.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_↩ stairs().

2.14.26.1 Overview

Stairs are here to provide class transitions between two maps or two layers on the same map, linked by tiles that look like stairs.

They don't provide any fundamental feature: you can already use teletransporters and destinations to make the hero go from a map to another one. And you can use sensors to make him go from a layer to another on the same map. But they add fanciness to these transitions: the sound of climbing or going down stairs, a special movement and appropriate sprite animations.

For stairs that transport the hero to a different map, you still have to use a teletransporter and a destination. Indeed, the stairs are additional, they don't replace teletransporters and destinations. You just have put all three entities (the stairs, a teletransporter and a destination) at the same coordinates and it works.

Stairs that move from a layer to another layer inside the same map work differently. They change this layer automatically for the hero. They must be placed on the lowest one of those two layers.

Stairs are invisible: you are supposed to place appropriate tiles where you make stairs. Their size is always 16x16 pixels.

2.14.26.2 Methods inherited from map entity

Stairs are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.26.3 Methods of the type stairs

None.

2.14.26.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Stairs are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.26.5 Events of the type stairs

None.

2.14.27 Bomb

A bomb is an entity that explodes after a few seconds.

You can create this type of map entity only dynamically with map:create_bomb(). It cannot be declared in the map data file.

2.14.27.1 Overview

A bomb can be lifted by the hero before it explodes. When it explodes, it is removed from the map, a new entity of type explosion is created at the same position and the sound `"explosion"` is played.

If a bomb is placed on a stream, it follows a movement corresponding to this stream.

2.14.27.2 Methods inherited from map entity

Bombs are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.27.3 Methods of the type bomb

None.

2.14.27.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Bombs are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.27.5 Events of the type bomb

None.

2.14.28 Explosion

This is an explosion whose sprite hurts the hero and enemies.

You can create this type of map entity only dynamically with map:create_explosion(). It cannot be declared in the map data file.

2.14.28.1 Overview

Explosions are automatically created by bombs that explode, and by dying enemies whose hurting style is `"boss"`.

Explosions hurt the hero and enemies that are sensible to them. But they may also blast doors and can be detected by sensors.

2.14.28.2 Methods inherited from map entity

Explosions are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.28.3 Methods of the type explosion

None.

2.14.28.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Explosions are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.28.5 Events of the type explosion

None.

2.14.29 Fire

This is some fire whose sprite hurts enemies.

You can create this type of map entity only dynamically with map:create_fire(). It cannot be declared in the map data file.

2.14.29.1 Overview

Fire is typically created with an equipment item like a lamp.

Fire hurts enemies that are sensible to it. It does not hurt the hero.

Fire can also be detected by non-playing characters: this is useful to implement a torch as a generalized %NPC that interacts with fire.

2.14.29.2 Methods inherited from map entity

Fire entities are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.29.3 Methods of the type fire

None.

2.14.29.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Fire entities are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.29.5 Events of the type fire

None.

2.14.30 Arrow

When the hero uses a hero:start_bow() bow, an arrow is created.

This type of entity can only be created by the engine.

2.14.30.1 Overview

An arrow entity is created when you call hero:start_bow(), after the bow animation. The arrow flies for a few seconds and disappears later.

It can hurt enemies and activate switches that are sensible to arrows.

Note that there are no predefined bow and arrows equipment items. It is your responsability to make them if you want. If you decide to make them, you will typically call hero:start_bow() from the item:on_using() callback of your bow equipment item.

2.14.30.2 Methods inherited from map entity

Arrows are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.30.3 Methods of the type arrow

None.

2.14.30.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Arrows are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.30.5 Events of the type arrow

None.

2.14.31 Hookshot

A hookshot map entity is used to implement the hero's hookshot state.

2.14.31.1 Overview

A hookshot entity is created when you call hero:start_hookshot().

It can immobilize or hurt enemies and transport the hero to distant places.

Note that there is no predefined hookshot equipment item. It is your responsability to make one if you want. If you decide to make one, you will typically call hero:start_hookshot() from the item:on_using() callback of your hookshot equipment item.

2.14.31.2 Methods inherited from map entity

A hookshot is a particular map entity. Therefore, it inherits all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.31.3 Methods of the type hookshot

None.

2.14.31.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

A hookshot is a particular map entity. Therefore, it inherits all events from the type map entity.

See Events of all entity types to know these events.

2.14.31.5 Events of the type hookshot

None.

2.14.32 Boomerang

A boomerang map entity is used to implement the hero's boomerang state.

2.14.32.1 Overview

A boomerang entity is created when you call hero:start_boomerang().

It can immobilize or hurt enemies and activate mechanisms.

Note that there is no predefined boomerang equipment item. It is your responsability to make one if you want. If you decide to make one, you will typically call hero:start_boomerang() from the item:on_using() callback of your boomerang equipment item.

2.14.32.2 Methods inherited from map entity

A boomerang is a particular map entity. Therefore, it inherits all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.32.3 Methods of the type boomerang

None.

2.14.32.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

A boomerang is a particular map entity. Therefore, it inherits all events from the type map entity.

See Events of all entity types to know these events.

2.14.32.5 Events of the type boomerang

None.

2.14.33 Camera

The camera is a rectangular shape that determines the visible part of the map. There is always exactly one camera on the current map. The camera is automatically created by the engine when loading a map. You cannot create or remove it. To access the camera of the map from one of your scripts, you can use map:get_camera().

2.14.33.1 Overview

The camera can work in two states. It can be either centered on an entity and track it automatically, or it can be controlled manually by your scripts or by the engine.

When a map starts, its camera is initially set to track the hero.

Use camera:start_manual() to switch to manual state, and camera:start_tracking() to switch to tracking state. Starting a movement on the camera also switches it to manual state.

2.14.33.1.1 Movements and constraints

Separators and map limits are obstacles for the camera.

It means that if the camera is close to a separator or to a limit of the map, its movement stops as you would expect, independently of its state.

Remember however that obstacles can be explicitly ignored with movement:set_ignore_obstacles(true). For a camera, it means that you can ignore separators and even map limits if you want. Parts of the camera that are outside the map are displayed with the background color of the tileset.

2.14.33.1.2 Camera size

By default, the size of the camera is the quest size, meaning that the camera occupies the whole screen.

It is possible to set a smaller size to the camera, in order to show the map only on a subpart of the screen. You can then use the rest of the screen to display some information like a HUD. Use camera:set_size() and camera:set_position_on_screen() to indicate the exact subpart of the screen where you want the camera to be displayed.

2.14.33.1.3 Suspending the game

Moving the camera does not automatically suspends the game, except when it is scrolling on a separator in tracking state. You can call game:set_suspended() if you want the game to be suspended during your camera sequence. Note that unlike most entities, the camera can still move when the game is suspended.

2.14.33.2 Methods inherited from map entity

A camera is a particular map entity. Therefore, it inherits all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.33.3 Methods of the type camera

The following methods are specific to cameras.

2.14.33.3.1 camera:set_size(width, height)

Sets the size of the camera.

This is the size of the rectangle the camera occupies on the screen. By default, this is the quest size, meaning that the camera has the size of the whole screen.

- `width` (number): Width of the camera in pixels.
- `height` (number): Height of the camera in pixels.

2.14.33.3.2 camera:get_position_on_screen()

Returns where the camera is displayed on the quest screen.

The default position is `0, 0`, meaning that the upper left corner of the camera is displayed on the upper left corner of the screen.

- Return value 1 (number): X coordinate of the camera on the screen, in quest screen coordinates.
- Return value 2 (number): Y coordinate of the camera on the screen, in quest screen coordinates.

2.14.33.3.3 camera:set_position_on_screen(x, y)

Sets where the camera is displayed on the quest screen.

You can use this function in conjunction with camera:set_size() to display the camera only on a subpart of the screen and for example keep the rest of the space for the HUD.

The default position is `0,0`, meaning that the upper left corner of the camera is displayed on the upper left corner of the screen.

- `x` (number): X coordinate of the camera on the screen, in quest screen coordinates.
- `y` (number): Y coordinate of the camera on the screen, in quest screen coordinates.

2.14.33.3.4 camera:get_position_to_track(entity), camera:get_position_to_track(x, y)

Returns the coordinates this camera should have in order to track the given entity or point, respecting constraints of map limits and separators.

The returned coordinates make their best to have the entity or point centered in the camera, but make sure that the camera does not cross separators or map limits. This function can be used to compute legal coordinates for the camera, and for example pass them to camera:set_position() or start a movement.

To get coordinates that center the camera on a map entity:

- `entity` (entity) The entity to center the camera on.

To get coordinates that center the camera on a point:

- `x` (number): X coordinate of the point to center the camera on.
- `y` (number): Y coordinate of the point to center the camera on.

2.14.33.3.5 camera:get_state()

Returns the name of the current state of the camera.

- Return value (string): The current camera state: `"tracking"` or `"manual"`.

2.14.33.3.6 camera:start_tracking(entity)

Switches the camera to tracking state.

The camera will be focused on an entity to track, and follow it when it moves.

When the tracked entity crosses a separator, the engine automatically starts a scrolling movement on the camera. The game is automatically suspended during the scrolling. After the scrolling, the camera continues to normally track the entity on the other side of the separator.

- `entity` (entity): The entity to track.

2.14.33.3.7 camera:get_tracked_entity()

Returns the entity currently tracked by this camera, if any.

- Return value (entity): The tracked entity if the camera is in tracking state, or `nil` if the camera is not in tracking state.

2.14.33.3.8 camera:start_manual()

Switches the camera to manual state.

Remarks

The camera automatically switches to manual state if you start a movement on it.

2.14.33.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

A camera is a particular map entity. Therefore, it inherits all events from the type map entity.

See Events of all entity types to know these events.

2.14.33.5 Events of the type camera

The following events are specific to cameras.

2.14.33.5.1 camera:on_state_changed()

Called when the state of the camera changes.

- `state` (string): Name of the new state. See camera:get_state() for the list of possible states.

Remarks

When the map starts, the initial state of the camera always `"tracking"`, because the camera initially tracks the hero.

2.14.34 Custom entity

A custom entity is a map entity entirely defined by your Lua scripts.

This type of map entity can be declared in the map data file. It can also be created dynamically with map:create_custom_entity().

2.14.34.1 Overview

Custom entities have no special properties or behavior. You can define them entirely in your scripts.

Optionally, a custom entity may be managed by a model. The model is the name of a Lua script that will be applied to all custom entities refering to it. This works exactly like the breed of enemies, except that it is optional. The model is useful if you have a lot of identical (or very similar) custom entities in your game, like for example torches.

If you make an entity that is unique in your game, like for example, a big rock that blocks the entrance of a dungeon and that requires some special action from the player, you don't need a model. You can just program the behavior of your custom entity in the script of its map. Similarly, to define a customized weapon of the hero, like a hammer, you don't need a model. Just create a custom entity from the item script of the hammer and define its behavior there.

2.14.34.2 Methods inherited from map entity

Custom entities are particular map entities. Therefore, they inherit all methods from the type map entity.

See Methods of all entity types to know these methods.

2.14.34.3 Methods of the type custom entity

The following methods are specific to custom entities.

2.14.34.3.1 custom_entity:get_model()

Returns the model of this custom entity.

The model is the name of a Lua script in the `"entities"` directory that manages this custom entity. This works exactly like the breed of enemies, except that it is optional.

- Return value (string): The model of custom entity, or `nil` if the custom entity has no model script.

2.14.34.3.2 custom_entity:set_size(width, height)

Sets the size of the bounding box of this custom entity.

The default value is `16x16` pixels. This is the effective size used to detect obstacles when moving, but the sprite(s) of the custom entity may be larger.

- `width` (number): Width of the custom entity in pixels.
- `height` (number): Height of the custom entity in pixels.

Remarks

If you create a pixel-precise collision test, the test will use the sprite(s) of this custom entity, not its bounding box. Therefore, this function has no influence on pixel-precise collisions, only on the detection of obstacles of the map when the custom entity moves.

2.14.34.3.3 custom_entity:set_origin(origin_x, origin_y)

Sets the origin point of this custom entity, relative to the upper left corner of its bounding box.

This origin point property allows entities of different sizes to have comparable reference points. For example, if you need to compute an angle between two entities, for example to move an entity away from another one, the calculation uses the origin point of both entities. Using the upper left corner of their bounding box would not give the correct angle (unless both entities had the same size).

The origin point is also the point of synchronization of an entity with its sprites (because again, an entity that has a given size may have sprites with different sizes).

The default values is `8,13` and is usually okay for custom entities of size `16x16`. See entity:get_origin() for more explanations about the origin point.

- `origin_x` (number): X coordinate of the origin point in pixels, relative to the upper left corner of the custom entity's bounding box.
- `origin_y` (number): Y coordinate of the origin point in pixels, relative to the upper left corner of the custom entity's bounding box.

2.14.34.3.4 custom_entity:get_direction()

Returns the direction of this custom entity.

This direction is set at creation time or when you can call custom_entity:set_direction().

- Return value 1 (number): The direction.

2.14.34.3.5 custom_entity:set_direction(direction)

Sets the direction of this custom entity.

Sprites of your custom entity that have such a direction automatically take it.

- Return value 1 (number): The direction.

2.14.34.3.6 custom_entity:create_sprite(animation_set_id, [sprite_name])

Creates a sprite for this custom entity.

- `animation_set_id` (string): Animation set to use for the sprite.
- `sprite_name` (string, optional): An optional name to identify the created sprite. Only useful for entities with multiple sprites (see entity:get_sprite()).
- Return value (sprite): The sprite created.

2.14.34.3.7 custom_entity:remove_sprite([sprite])

Removes and destroys a sprite of this custom_entity.

It may have been created at the creation of the custom entity, or when you called custom_entity:create_sprite().

- `sprite` (sprite, optional): The sprite to remove. The default value is the first sprite that was created.

2.14.34.3.8 custom_entity:is_drawn_in_y_order()

Returns whether this custom entity is drawn in Y order.

The displaying order of entities having this property depends on their Y position. Entities without this property are drawn in the normal order (i.e. in the order of their creation), and before all entities with this property.

Entities with this property are displayed from the one the most to the north to the one the most to the south.

- Return value (boolean): `true` if this custom entity is displayed in Y order, `false` if it is displayed in creation order.

2.14.34.3.9 custom_entity:set_drawn_in_y_order([y_order])

Sets whether this custom entity should be drawn in Y order.

The default setting is `false`, meaning that by default, a custom entity is displayed in Z order, which is the creation order unless you call entity:bring_to_front() or entity:bring_to_back().

See custom_entity:is_drawn_in_y_order() for more details.

- `y_order` (boolean, optional): `true` if this custom entity is displayed in Y order, `false` if it is displayed in creation order. No value means `true`.

2.14.34.3.10 custom_entity:set_traversable_by([entity_type], traversable)

Sets whether this custom entity can be traversed by other entities.

By default, a custom entity can be traversed.

- `entity_type` (string, optional): A type of entity. See entity:get_type() for the possible values. If not specified, the setting will be applied to all entity types that do not override it.
- `traversable` (boolean, function or `nil`): Whether this entity type can traverse your custom entity. This can be:
  - A boolean: `true` to make your custom entity traversable by this entity type, `false` to make it obstacle.
  - A function: Custom test. This allows you to decide dynamically. The function takes your custom entity and then the other entity as parameters, and should return `true` if you allow the other entity to traverse your custom entity. This function will be called every time a moving entity of the specified type is about to overlap your custom entity.
  - `nil`: Clears any previous setting for this entity type and therefore restores the default value.

2.14.34.3.11 custom_entity:set_can_traverse([entity_type], traversable)

Sets whether this custom entity can traverse other entities.

This is important only if your custom entity can move.

By default, this depends on the other entities: for example, sensors can be traversed by default while doors cannot unless they are open.

- `entity_type` (string, optional): A type of entity. See entity:get_type() for the possible values. If not specified, the setting will be applied to all entity types that do not override it.
- `traversable` (boolean, function or `nil`): Whether your custom entity can traverse the other entity type. This can be:
  - A boolean: `true` to allow your custom entity to traverse entities of the specified type, `false` otherwise.
  - A function: Custom test. This allows you to decide dynamically. The function takes your custom entity and then the other entity as parameters, and should return `true` if you allow your custom entity to traverse the other entity. When your custom entity has a movement, this function will be called every time it is about to overlap an entity of the specified type.
  - `nil`: Clears any previous setting for this entity type and therefore restores the default value.

2.14.34.3.12 custom_entity:can_traverse_ground(ground)

Returns whether this custom entity can traverse a kind of ground.

This is important only if your custom entity can move.

The ground is the terrain property of the map. It is defined by tiles and by other entities that may change it dynamically.

- `ground` (string): A kind of ground. See map:get_ground() for the possible values.
- Return value (boolean): `true` if your custom entity can traverse this kind of ground.

2.14.34.3.13 custom_entity:set_can_traverse_ground(ground, traversable)

Sets whether this custom entity can traverse a kind of ground.

This is important only if your custom entity can move.

The ground is the terrain property of the map. It is defined by tiles and by other entities that may change it dynamically.

By default, this depends on the the ground: for example, the `"grass"` ground can be traversed by default while the `"low wall"` ground cannot.

- `ground` (string): A kind of ground. See map:get_ground() for the possible values.
- `traversable` (boolean): Whether your custom entity can traverse this kind of ground.

2.14.34.3.14 custom_entity:add_collision_test(collision_mode, callback)

Registers a function to be called when your custom entity detects a collision when another entity.

- `collision_mode` (string or function): Specifies what kind of collision you want to test. This may be one of:
  - `"overlapping"`: Collision if the bounding box of both entities overlap. This is often used when the other entity can traverse your custom entity.
  - `"containing"`: Collision if the bounding box of the other entity is fully inside the bounding box of your custom entity.
  - `"origin"`: Collision if the origin point or the other entity is inside the bounding box of your custom entity.
  - `"center"`: Collision if the center point of the other entity is inside the bounding box of your custom entity.
  - `"facing"`: Collision if the facing position of the other entity's bounding box is touching your custom entity's bounding box. Bounding boxes don't necessarily overlap, but they are in contact: there is no space between them. When you consider the bounding box of an entity, which is a rectangle with four sides, the facing point is the middle point of the side the entity is oriented to. This `"facing"` collision test is useful when the other entity cannot traverse your custom entity. For instance, if the other entity has direction "east", there is a collision if the middle of the east side of its bounding box touches (but does not necessarily overlap) your custom entity's bounding box. This is very often what you need, typically to let the hero interact with your entity when he is looking at it.
  - `"touching"`: Like `"facing"`, but accepts all four sides of the other entity's bounding box, no matter its direction.
  - `"sprite"`: Collision if a sprite of the other entity overlaps a sprite of your custom entity. The collision test is pixel precise.
  - A function: Custom collision test. The function takes your custom entity and then the other entity as parameters and should return `true` if there is a collision between them. This function will be called every time the engine needs to check collisions between your custom entity and any other entity.
- `callback` (function): A function that will be called when the collision test detects a collision with another entity. This allows you to decide dynamically. This function takes your custom entity and then the other entity as parameters. If the collision test was `"sprite"`, both involved sprites are also passed as third and fourth parameters: the third parameter is the sprite of your custom entity, and the fourth parameter is the sprite of the other entity. This may be useful when your entities have several sprites, otherwise you can just ignore these additional sprite parameters.

Remarks

See also entity:overlaps() to directly test a collision rather than registering a callback.

2.14.34.3.15 custom_entity:clear_collision_tests()

Disables any collision test previously registered with custom_entity:add_collision_test().

2.14.34.3.16 custom_entity:has_layer_independent_collisions()

Returns whether this custom entity can detect collisions with entities even if they are not on the same layer.

By default, custom entities can only have collisions with entities on the same layer.

- Return value (boolean): `true` if this entity can detect collisions even with entities on other layers.

2.14.34.3.17 custom_entity:set_layer_independent_collisions([independent])

Sets whether this custom entity can detect collisions with entities even if they are not on the same layer.

By default, custom entities can only have collisions with entities on the same layer. If you set this property to `true`, the collision tests will be performed even with entities that are on a different layer.

- `independent` (boolean, optional): `true` to make this entity detect collisions even with entities on other layers. No value means `true`.

2.14.34.3.18 custom_entity:get_modified_ground()

Returns the kind of ground (terrain) defined by this custom entity on the map.

- Return value (string): The ground defined by this custom entity, or `nil` if this custom entity does not modify the ground. See map:get_ground() for the list of possible grounds.

2.14.34.3.19 custom_entity:set_modified_ground(modified_ground)

Sets the kind of ground (terrain) defined by this custom entity on the map.

The ground of the map is normally defined by tiles, but other entities may modify it dynamically.

This property allows you to make a custom entity that modifies the ground of the map, for example a hole with a special sprite or ice with particular collision callbacks. The modified ground will be applied on the map in the rectangle of this custom entity's bounding box. Your custom entity can move: the ground will still be correctly applied.

- `modified_ground` (string): The ground defined by this custom entity, or `nil` (or `"empty"`) to make this custom entity stop modifying the ground. See map:get_ground() for the list of possible grounds.

Remarks

If you only need to modify the ground of the map dynamically, for example to make a moving platform over holes, a dynamic tile with a movement may be enough.

2.14.34.4 Events inherited from map entity

Events are callback methods automatically called by the engine if you define them.

Custom entities are particular map entities. Therefore, they inherit all events from the type map entity.

See Events of all entity types to know these events.

2.14.34.5 Events of the type custom entity

The following events are specific to custom entities.

2.14.34.5.1 custom_entity:on_update()

Called at each cycle while this custom entity lives on the map.

Remarks

As this function is called at each cycle, it is recommended to use other solutions when possible, like timers and other events.

2.14.34.5.2 custom_entity:on_suspended(suspended)

Called when the map has just been suspended or resumed.

The map is suspended by the engine in a few cases, like when the game is paused or when the camera is being moved by a script. When this happens, all map entities stop moving and most sprites stop their animation.

- `suspended` (boolean): `true` if the map was just suspended, `false` if it was resumed.

2.14.34.5.3 custom_entity:on_created()

Called when this custom entity has just been created on the map.

2.14.34.5.4 custom_entity:on_enabled()

Called when this custom entity has just been enabled.

2.14.34.5.5 custom_entity:on_disabled()

Called when this custom entity has just been disabled.

2.14.34.5.6 custom_entity:on_pre_draw()

Called just before this custom entity is drawn on the map. You may display additional things below the sprites.

2.14.34.5.7 custom_entity:on_post_draw()

Called just after this custom entity is drawn on the map. You may display additional things above the sprites.

2.14.34.5.8 custom_entity:on_ground_below_changed(ground_below)

Called when the kind of ground on the map below this custom entity has changed. It may change because this custom entity is moving, or when because another entity changes it.

- `ground_below` (string): The kind of ground at the origin point of this custom entity. `nil` means empty, that is, there is no ground at this point on the current layer.

2.14.34.5.9 custom_entity:on_interaction()

Called when the hero interacts with this custom entity, that is, when the player presses the action command while facing this custom entity.

Remarks

This event is also available with NPCs.

2.14.34.5.10 custom_entity:on_interaction_item(item_used)

Called when the hero uses any equipment item (the player pressed an item command) while facing this custom entity.

- `item_used` (item): The item currently used by the player.
- Return value (boolean): `true` if an interaction happened. If you return `false` or nothing, then item_used:on_using() will be called (just like if there was no custom entity in front of the hero).

Remarks

This event is also available with NPCs.

Chapter 3

Solarus 1.5 - Quest data files specification

We explain here how a quest is built.

A quest is a data package that may be run by the Solarus C++ engine (either by the `solarus-run` executable or by the Solarus GUI).

When you run the `solarus-run` executable file, it needs the path of the quest to launch. This can be done at runtime by specifying the path as a command-line argument: `solarus-run path/to/your/quest`. If the quest path is not specified, the current directory is considered by default, unless you compiled Solarus with another default quest path.

When you run the Solarus GUI (the `solarus` executable file), you can select graphically the quest to run and you can change some settings.

In both cases, the quest may have one of the following two forms:

- A zip archive called `data.solarus` or `data.solarus.zip` and containing all data of your quest. This archive form is useful when your quest is finished and you distribute it to people.
- A directory called `data` and containing all data of your quest. This form is handy when you are developing your quest. It is also the only possible form to edit your quest with Solarus Quest Editor. In fact, when you modify a map in the editor, you don't even need to restart the game.

The data files represent all resources used by both the engine and the quest, such as sounds, musics, images, sprites, dialogs, maps and Lua scripts. We specify here the details for all those files.

Here is the full structure of the `data` directory / `data.solarus` archive / `data.solarus.zip` archive of a quest and the syntax of each file.

- quest.dat: global properties of your quest.
- project_db.dat: list of all resources (maps, sprites, enemies...) with their id and human-readable name.
- main.lua: main Lua script of your quest.
- *.lua: other Lua scripts, possibly organized in subdirectories.
- logos/: icons and logos of your quest, used by the Solarus GUI to represent your game.
  - `logos/logo.png`: a 200x140 logo of your quest.
  - `logos/icon_*.png`: icon of your quest with various possible sizes.

- sounds/: contains all sound effects.
  - `sounds/*`.ogg: your sound files.
- musics/: contains all musics.
  - `musics/*`.{ogg, it, spc}: your music files.
- fonts/: contains the fonts used to draw text in your quest.
  - `text/*`.{ttf, ttc, fon, png, ...}: your font files.
- `languages/`: contains the language-specific files.
  - languages/xx/text/strings.dat: strings of language `"xx"`.
  - languages/xx/text/dialogs.txt: dialogs of language `"xx"`.
  - languages/xx/images/: contains all images that have text in them in language `"xx"`.
    - `languages/xx/images/*`.png: your language-specific images.
- `sprites/`: contains all animated sprites, possibly organized in subdirectories.
  - sprites/xx.dat: definition of sprite `"xx"`.
  - `sprites/*`.png: images used by your sprites (except the tileset-dependent ones).
- `maps/`: contains all maps of your quest.
  - maps/xx.dat: definition of map `"xx"` and all its entities.
  - maps/xx.lua: Lua script of map `"xx"`.
- `tilesets/`: contains all tilesets available to maps.
  - tilesets/xx.dat: definition of tileset `"xx"` and all its tile patterns.
  - tilesets/xx.tiles.png": image with all tile patterns of tileset \c "xx".
  - tilesets/xx.entities.png: image with all tileset-dependent sprite animations for tileset `"xx"`.
- `items/`: contains all equipment item Lua scripts.
  - items/xx.lua: Lua script that defines the properties and the behavior of equipment item `"xx"`.
- `enemies/`: contains all enemy Lua scripts.
  - enemies/xx.lua: Lua script that defines the properties and the behavior of the enemy model `"xx"`.
- `entities/`: contains all custom entity Lua scripts.
  - entities/xx.lua: Lua script that defines the properties and the behavior of the custom entity model `"xx"`.

3.1 Quest properties file

Some general information regarding your quest are stored in the quest properties file `quest.dat`. This include information that the Solarus GUI can use to show a description of each quest before the user plays, as well as technical settings indicating how your quest should run.

`quest.dat` is mandatory: if this file does not exist, the engine considers that there is no quest in the directory.

3.1.1 Syntax of the quest properties file

Solarus Quest Editor fully supports the edition of the quest properties file. You should not have to edit `quest.dat` by hand unless you know what you are doing.

`quest.dat` is a text file encoded in UTF-8.

The sequence of characters `--` (two dashes) marks the beginning of a comment. After them, the rest of the line is ignored by the engine. Empty lines are also ignored.

The definition of your quest properties starts with `quest{` and ends with `}`. Each property is declared with the syntax `key = value` and separated with commas. It is allowed to have an extra comma after the last property. String values are enclosed within double quotes.

The recognized properties are listed below. Only `solarus_version` and `title` are mandatory, but `write_dir` is also required if you want to be able to use savegames.

- `solarus_version` (string): Format of the engine your data files are compatible with. This string should be a major and minor version, for example "1.5". You can also indicate a full version string, including the patch number (like "1.5.0") but the patch number will be ignored. Patch versions are ignored because they don't break compatibility.
- `title` (string): Name of your quest.
- `short_description` (string, optional): A one-line description of the quest.
- `long_description` (string, optional): A longer description of the quest.
- `author` (string, optional): Person or team who created the quest.
- `quest_version` (string, optional): Current version of your quest.
- `release_date` (string, optional): Date of your last quest release (YYYY-MM-DD).
- `website` (string, optional): URL of your website.
- `write_dir` (string, optional): Directory where Solarus will write savegames and setting files for your quest. It will be a subdirectory of `'$HOME/.solarus/'`, automatically created by the engine. Its name should identify your quest, to avoid confusion with other Solarus quests that might also be installed on the user's machine. You must define it before you can use savegames or setting files.
- `normal_quest_size` (string, optional): Usual size of the game screen in pixels, as two integer values separated by `"x"`. Example: `"320x240"`. The default value is `"320x240"`.
- `min_quest_size` (string, optional): Minimum size of the game screen in pixels. Example: `"320x200"`. No value means the same value as `normal_quest_size`.
- `max_quest_size` (table, optional): Maximum size of the game screen in pixels. Example: `"400x240"`. No value means the same value as `normal_quest_size`.

3.1.2 About the quest size

`normal_quest_size`, `min_quest_size` and `max_quest_size` define the range of possible sizes of the game screen. This is the logical size of the game area. It represents how much content the user can see on the map when playing your quest. Allowing a range of sizes instead of only one size improves the portability: each system and each user can choose a size that occupies the full ratio of the screen instead of having black bars. However, be aware that if you allow a range, some users will be able to see more game content than others!

At compilation time, systems can specify their preferred quest size. It will be used if it is in the range of sizes allowed by your quest. And the quest size can also be overriden at runtime with a command-line option.

At runtime, you are guaranteed that the actual quest size will be in the range allowed in this quest properties file.

Remarks

The quest size is the logical size of your quest and it never changes at runtime. Don't confuse it with the size of the window. The actual window shown to the user may be bigger or smaller than that. It can resized and it can also be in fullscreen mode. When the window size is different from the quest size, the quest image is scaled to fit the window. Black bars are added if necessary to keep the correct ratio.

3.1.3 Example

Example of `quest.dat` file:

```
quest{
  solarus_version = "1.4",
  write_dir = "my_quest",
  title_bar = "My incredible quest",
  normal_quest_size = "320x240",  -- Works on most configurations.
  min_quest_size = "320x200",     -- Might make fullscreen compatible with more systems.
  max_quest_size = "400x240",     -- Suited for Android and other mobile devices.
}
```

Remarks

The syntax of the quest properties file is actually valid Lua. The engine internally uses Lua to parse it.

3.2 Resource list

The `project_db.dat` file declares all resources (maps, musics, sprites, items, enemies, fonts, etc.) of your quest and their user-friendly names. Each resource usually corresponds to a data file.

The engine needs this file to preload or initialize some resources when the program starts.

The quest editor needs this file to provide graphical components that let the user choose a resource in a list of user-friendly names. For instance, when the user creates a teletransporter on the map, a graphical list lets him choose the destination map of the teletransporter. This list contains the user-friendly name of each map.

We give here the syntax of `project_db.dat`.

`project_db.dat` is a text file encoded in UTF-8. The sequence of characters `--` (two dashes) marks the beginning of a comment. After them, the rest of the line is ignored. Empty lines are also ignored.

The definition of each resource starts with the name of a resource type (`map`, `tileset`, `sound`, `music`, `sprite`, `item`, `enemy` or `language`), followed by an opening brace, and ends with a closing brace. Inside the braces, the properties of the resource are specified. Properties are declared with the syntax `key = value` and separated with commas. It is allowed to have an extra comma after the last property. String values are enclosed within double quotes. Each resource must have the following two properties:

- `id` (string): Id of the element. It should not contain spaces. The id determines the data file(s) of this element.
  - For a map: the id is the name of the map data file without its extension, relative to the `maps` directory. Note that the first map declared in this file will be the default map.
  - For a tileset: the id is the name of the tileset data file without its extension, relative to the `tilesets` directory.
  - For a sound: the id is the sound file name without its extension, relative to the `sounds` directory.
  - For a music: the id is the music file name without its extension, relative to the `musics` directory.

  - For a sprite: the id is the name of the sprite data file, relative to the `sprites` directory. The sprite data file may be in a hierarchy of subdirectories. Subdirectories must be separated by a slash "/" character in the sprite id.
  - For an equipment item: the id is the name of the item's Lua script without its extension, relative to the `items` directory.
  - For an enemy model: the id is the is name of the enemy's Lua script without its extension, relative to the `enemies` directory.
  - For a language: the id is the name of a subdirectory of the `languages` directory.
  - For a font: the id is the font file name without its extension, relative to the `fonts` directory.

- `description` (string): A human-readable name that describes this element. It may contain spaces. It is useful to show to userse something more user-friendly than the id.

Example of a quest resource list file:

```
map{      id = "outside",      description = "Outside World" }
map{      id = "hero_house",   description = "House of the hero" }
map{      id = "shop",         description = "Shop" }
map{      id = "dungeon_1_1f", description = "Dungeon 1 - First floor" }
map{      id = "dungeon_1_2f", description = "Dungeon 1 - Second floor" }

tileset{  id = "overworld",    description = "Overworld" }
tileset{  id = "house",        description = "House" }
tileset{  id = "dungeon",      description = "Dungeon" }

sound{    id = "door_closed",  description = "Door closing" }
sound{    id = "door_open",    description = "Door opening" }
sound{    id = "enemy_hurt",   description = "Enemy hurt" }
sound{    id = "jump",         description = "Jumping" }
sound{    id = "treasure",     description = "Treasure" }

item{     id = "sword",        description = "Sword" }
item{     id = "bow",          description = "Bow" }
item{     id = "arrow",        description = "Arrows (x1 / x5 / x10)" }

enemy{    id = "soldier",      description = "Soldier" }
enemy{    id = "dragon",       description = "Dragon" }

language{ id = "en",           description = "English" }

font{     id = "8_bit",        description = "8 bit font" }
```

In the quest editor, you can add, remove and change the id and the description of resources from the quest tree. Therefore, you probably don't need to edit this file by hand.

Remarks

The syntax of this resource list file is actually valid Lua. The engine internally uses Lua to parse it.

3.3 Main Lua script

`main.lua` is the entry point of your quest. Everything starts from here at runtime. Of course, you can make other script files, possibly organized in subdirectories, and call them from `main.lua`. For instance, you will probably make a title screen and then start a game.

Here is an example of main script that does almost nothing:

```
function sol.main.on_started()
  -- This function is called when Solarus starts.
  print("Welcome to my quest.")
end

function sol.main.on_finished()
  -- This function is called when Solarus stops or is reset.
  print("See you!")
end
```

See the Lua API for more information about Lua scripting in Solarus.

3.4 Quest logos and icons

The "logos/" directory of your quest can contain a logo and some icons for your quest. The Solarus GUI uses them in the quest list to represent each quest before the user plays.

The logo and icons are optional, but it is recommended to make some for your quest to be better identified in the quest list of the Solarus GUI.

3.4.1 Quest logo

The logo of your quest should be a PNG image of size 200x140 called "logos/logo.png". The logo is optional.

3.4.2 Quest icons

An icon is also useful for the Solarus GUI to distinguish your quest. Similarly to the logo, the icon is optional. Multiple icon sizes are allowed, and each size should be in a separate PNG file. The following icon file names are allowed, with the corresponding sizes from 16x16 pixels to 1024x1024 pixels:

- "logos/icon_16.png",
- "logos/icon_24.png",
- "logos/icon_32.png",
- "logos/icon_48.png",
- "logos/icon_64.png",
- "logos/icon_128.png",
- "logos/icon_256.png",
- "logos/icon_512.png",
- "logos/icon_1024.png".

It is possible (and recommended) to provide an icon with multiple sizes. The Solarus GUI will then automatically pick the most suitable one(s) to fit its needs.

3.5 Sounds

The `sounds` directory contains all small sound effects used by both the engine and the quest. Sounds files must have the extension .ogg (Ogg Vorbis).

Don't forget to add all sound files to the quest tree in the editor (or by editing project_db.dat by hand).

Here is a list of sound effects played by the engine in various situations. All these sound effects should exist in the `sounds` directory.

```
arrow_hit.ogg
boomerang.ogg
boss_hurt.ogg
boss_killed.ogg
bow.ogg
bush.ogg
cane.ogg
chest_open.ogg
cursor.ogg
danger.ogg
door_closed.ogg
door_open.ogg
door_unlocked.ogg
enemy_hurt.ogg
enemy_killed.ogg
explosion.ogg
heart.ogg
hero_dying.ogg
hero_falls.ogg
hero_hurt.ogg
hero_lands.ogg
hero_pushes.ogg
hookshot.ogg
jump.ogg
lift.ogg
message_end.ogg
message_letter.ogg
monster_hurt.ogg
ok.ogg
picked_item.ogg
running.ogg
shield.ogg
splash.ogg
stairs_up_start.ogg
stairs_up_end.ogg
stairs_down_start.ogg
stairs_down_end.ogg
stone.ogg
swim.ogg
switch.ogg
sword1.ogg ...
sword_spin_attack_load.ogg
sword_spin_attack_release.ogg
sword_tapping.ogg
sword_tapping_weak_wall.ogg
throw.ogg
timer.ogg
timer_hurry.ogg
treasure.ogg
victory.ogg
walk_on_grass.ogg
walk_on_water.ogg
warp.ogg
wrong.ogg
```

3.6 Musics

The `musics` directory contains all musics files used by the quest. For now, the following formats are recognized:

- OGG: Ogg Vorbis audio file,
- IT: Impulse Tracker Module,
- SPC: a Super NES original music file (not recommended).

The SPC format is supported but not recommended because it is slow to decode. Therefore, it is recommended to convert SPC musics to IT musics.

Future releases may support more music formats.

3.6.1 Loop settings

When starting a map, its music is automatically played and will loop to the beginning if it finishes. When you play a music from a script (using sol.audio.play_music()), you can specify the action to do when the music finishes. By default, this action is also to loop the music to the beginning.

However, some musics already have their own loop internal loop. It means that they don't finish (they loop by themselves) so the rules above have no effect on them. Instead, such musics are able to loop to a specific point rather than to the beginning.

- This is the case of SPC musics because of their special emulation format. Some of them simply play forever, other play an empty sound forever after some point, but SPC musics have no end.
- This can also be the case of IT musics if they contain internal loop information to play forever.
- For OGG musics, Solarus detects the metadata tags `LOOPSTART`, `LOOPEND` and `LOOPLENGTH`. These tags are all optional. They should be set to a number of PCM samples. There will be an internal loop if `LOOPSTART` is specified. You can then use either `LOOPEND` or `LOOPLENGTH` to indicate from where you want the music to loop (by default: from the end of the file). If `LOOPEND` is specified, the music will loop from position `LOOPEND` to position `LOOPSTART`. Otherwise, if `LOOPLENGTH` is specified, the music will loop from position `LOOPSTART + LOOPLENGTH` to position `LOOPSTART`.

3.7 Fonts

The `fonts` directory contains the font files used by the engine and your quest. Solarus supports both outline fonts and bitmap fonts.

Like maps, musics, languages, etc., fonts are a resource declared in the quest resource list.

3.7.1 Outline fonts

Outline fonts (or vector fonts) are the fonts you commonly use in a word processor. They are composed of vector lines and curves, and can be scaled to any size without pixellation.

In Solarus, the formats of outline fonts supported are the ones of the SDL_ttf library, including .ttf, .tcc and `fon`. At runtime, you can set their size and their color (see the text surface API).

3.7.2 Bitmap fonts

Sometimes, you don't want scalable fonts, but you want to define your font pixel by pixel. This is especially true for old-school looking games when you want small letters.

A bitmap font is a font whose letters directly come from a PNG image file. With a bitmap font, you cannot change the size or the color. But you can for example add a border to your letters, which is impossible with outline fonts.

In Solarus, a bitmap font is a single PNG image that contains characters organized in the following specific way.

All characters in the image have the same size (for example 8x12 pixels). The characters in the image must be organized in 16 rows and 128 columns. It is like a big table of characters. The size of a single character is implicit: the engine determines it by dividing the height of your image by 16 and the width by 128.

Characters are organized in the UTF-8 order. The first row of the image corresponds to the 128 characters that are encoded in a single byte in UTF-8. This is exactly the ASCII table. If you only need ASCII characters, you can leave the rest of your image entirely transparent. The other 15 rows allow you to put any character that can be encoded in two bytes in UTF-8. In UTF-8, characters that use two bytes have the form `110xxxxx 10xxxxxx`. If you remove the fixed `110` in the first byte and the fixed `10` in the second byte, you obtain a 11-bit number, in other words, a number between 128 and 2047 (the Unicode code point of the character). This code point gives you the index of the rectangle of your character in the image.

All data files that include text are encoded in UTF-8 in Solarus. Therefore, unless you have characters that need more than two bytes in UTF-8, you can draw your text with a PNG font.

3.8 Translated strings

Strings displayed during the game, for example in your menus, need to be localized in the current language. To this end, a `strings.dat` file is placed inside each `languages/xx/text` directory, where `xx` is a language code defined in project_db.dat (e.g. `"en"` or `"fr"`).

When translating the game, you have to translate this `strings.dat` file as well as the dialogs file (`dialogs.dat`). Have a look at the How to translate a quest page if you are interested in translating the game (and you should also contact us).

Solarus Quest Editor fully supports the edition of string files. You can either use it to edit a string file graphically, or you can edit `strings.dat` by hand in a text editor if you prefer.

`string.dat` is a text file encoded in UTF-8. We specify here its full syntax, with instructions for quest makers and translators.

The sequence of characters `--` (two dashes) marks the beginning of a comment. After them, the rest of the line is ignored by the engine. Empty lines are also ignored. Quest makers should put many comments to explain where each dialog is used and give instructions to translators if needed.

The definition of a string starts with `text{` and ends with `}`. Inside the braces, the properties of the string are specified. Properties are declared with the syntax `property_name = property_content` and separated with commas. It is allowed to have an extra comma after the last property. String contents should be enclosed within double quotes, except when specified otherwise. Each string must have exactly the following two properties:

- `key` (string): A name identifying this string for all languages. Don't change the key when you are translating a quest, since the key identifies the string.
- `value` (string): The content of this string in the current language.

For each language, the `strings.dat` file must define strings with the same keys.

I recommend to add comments (lines beginning by `--`) to explain where each string is used and how many characters it should not exceed. Future translators will thank you.

An an example, here is a part of a possible `languages/fr/text/strings.dat` file:

```
-- Savegame selection menu.
-- Strings beginning with "selection_menu.phase" should not exceed 45 characters.
text{ key = "selection_menu.phase.select_file", value = "Veuillez choisir un fichier" }
text{ key = "selection_menu.phase.choose_mode", value = "Choisissez un mode de jeu" }
text{ key = "selection_menu.phase.choose_name", value = "Quel est votre nom ?" }
text{ key = "selection_menu.phase.confirm_erase", value = "Etes-vous sûr ?" }
text{ key = "selection_menu.phase.erase_file", value = "Quel fichier voulez-vous effacer ?" }
text{ key = "selection_menu.phase.options", value = "Appuyez sur Espace pour modifier" }
text{ key = "selection_menu.phase.options.changing", value = "<  > : choisir, Espace : valider" }
text{ key = "selection_menu.erase", value = "Effacer" }
text{ key = "selection_menu.cancel", value = "Annuler" }
text{ key = "selection_menu.big_yes", value = "OUI" }
text{ key = "selection_menu.big_no", value = "NON" }
text{ key = "selection_menu.empty", value = "Vide" }
text{ key = "selection_menu.options", value = "Options" }
text{ key = "selection_menu.back", value = "Retour" }
text{ key = "selection_menu.options.language", value = "Langue" }
text{ key = "selection_menu.options.video_mode", value = "Mode" }
text{ key = "selection_menu.options.music_volume", value = "Volume de la musique" }
text{ key = "selection_menu.options.sound_volume", value = "Volume des sons" }
```

Remarks

This syntax of the strings file is actually valid Lua. The engine internally uses Lua to parse it.

3.9 Translated dialogs

Dialogs are all messages shown to the player in the dialog box during the game. Some dialogs are triggered by the engine and others by your quest. They are all defined in the file `languages/xx/text/dialogs.dat`, where `xx` is a language code defined in project_db.dat (e.g. `"en"` or `"fr"`).

When translating the game, you have to translate this `dialogs.dat` file as well as the strings file (`strings.dat`). Have a look at the How to translate a quest page if you are interested in translating a Solarus game.

Solarus Quest Editor fully supports the edition of dialog files. You can either use it to edit a dialog file graphically, or you can edit `dialogs.dat` by hand in a text editor if you prefer.

`dialogs.dat` is a text file encoded in UTF-8. We specify here its full syntax, with instructions for quest makers and translators.

The sequence of characters `--` (two dashes) marks the beginning of a comment. After them, the rest of the line is ignored by the engine. Empty lines are also ignored. Quest makers should put many comments to explain where each dialog is used and give instructions to translators if needed.

The definition of a dialog starts with `dialog{` and ends with `}`. Inside the braces, the properties of the dialog are specified. Properties are declared with the syntax `key = value` and separated with commas. It is allowed to have an extra comma after the last property. String values should be enclosed within double quotes, except when specified otherwise. Each dialog must have at least the following two properties:

- `id` (string): A name identifying the dialog. The id allows the engine and your Lua scripts to refer to the dialog whenever they want to display it. If you are translating dialogs, you must never translate the id.

- `text` (multi-line string, optional): Contents of this dialog. If you are translating dialogs, `text` is the property to translate. The text has usually several lines, but can be empty. The text should be enclosed between [[and]], which is the notation for multiline strings. The closing marker]] should be on a new line.

Only `id` and `text` are recognized by the engine (because the default minimal built-in dialog box use them), but you can add other properties depending on your dialog box implementation. Properties of a dialog are always of type string. Numbers and booleans are also accepted, but they will be converted to strings. For booleans, `true` is replaced by the string "1" and `false` is replaced by the string "0".

The dialog box system is entirely customizable, so you it is up to you to put the properties that fit your needs. For example, you can make properties that define:

- the speed of the dialog (assuming that your dialog box displays the letters gradually),
- whether the dialog stops automatically or manually,
- whether it can be skipped by the player,
- whether it contains special features like a question or an icon,
- etc.

When a dialog starts at runtime, your dialog box script receives these properties through the game:on_dialog_started() event and is responsible to handle them.

Here is a small example of dialog file. Again, remember that only `id` and `text` are mandatory: other properties are specific to a particular implementation of dialog box.

```
dialog{
  id = "wise_man",
  skip = "current",
  question = true,
  next = "wise_man.thanks",
  next2 = "wise_man.insisting",
  text = [[
I can feel courage in
you.
Do you think you can
protect the Princess?
Definitely!
No way
]],
}

dialog{
  id = "wise_man.insisting",
  question = true,
  next = "wise_man.thanks",
  next2 = "wise_man.insisting",
  text = [[
I beg you.
You are the only one
one able to protect
our kingdom...
Sure
No way
]],
}

dialog{
  id = "wise_man.thanks",
  text = [[
I knew I could count on
your immense bravery.
]],
}
```

Remarks

This syntax of the dialog file is actually valid Lua. The engine internally uses Lua to parse it.

3.10 Sprite data file

A sprite is an animated image that can be displayed on a surface or attached to an entity of the map.

3.10.1 Overview

A sprite displays images from an animation set with a current animation, a current direction and a current frame. Several sprites displayed at the same time can share the same animation set and have different states (i.e. different animations, directions and frames).

An animation set has at least two data files:

- One or more PNG images that contain the images of the animation set.
- A sprite definition file like `sprites/xx.dat`, where `xx` is the id of the sprite animation set. This file contains the definition of each animation, direction and frame of the animation set.

The sprite animation sets are represented as in Multimedia Fusion:

- A sprite animation set is composed of one or several animations. For instance, the animation set of a basic enemy may have four animations: `"stopped"`, `"walking"`, `"hurt"` and `"shaking"`.
- Each animation is composed of one or several directions, numbered from 0. Non-playing characters and usual enemies have four directions.
- Each direction is a sequence of frames, where a frame is a static image. Frames are numbered from 0. Once the animation is finished, the sequence may loop, that is, come back to a specific frame (often, but not necessarily, the first one).

The definition file `sprites/xx.dat` contains those information.

3.10.2 Origin point

A crucial concept that comes with sprites is the notion of origin point.

When a sprite is drawn at some coordinates, the origin point determines what exact point of the sprite's rectangle is at those coordinates. This point is not necessarily the upper-left corner of the rectangle.

The origin point is essential when your sprite has several animations of different frame sizes, and also when your sprite represents a map entity whose collision box has a different size (usually, the sprite of a character is larger than its collision box).

Let's take an exemple: the animation `"walking"` of a character's sprite has frames of size `24x32` in the PNG image, while its animation `"plunging"` has frames of size `64x24` (for instance, because some water splashing is displayed next to the body).

In such a case, how can you indicate to the engine the relative position of frames of both animations? When switching from `"walking"` to `"plunging"`, you most likely don't want the upper-left corner of the `64x24` rectangle to be simply displayed where the upper-left corner of the `24x32` rectangle was displayed.

The solution is to specify a point of `"walking"` frames that should match a point of `"plunging"` frames. We call this point the origin point. You can specify it on each animation, relative to the upper-left corner of a frame. In the example above, the origin point might be `12,29` for animation `"walking"` and `32,21` for animation `"plunging"`.

Furthermore, when the sprite is used to represent a character or an enemy on the map, the collision box of that character or enemy is a rectangle whose size is usually `16x16` (used to avoid obstacles when moving). The sprite is often larger than this collision box, so in this situation, we also need the notion of origin point. Therefore, map entities also have an origin point property that allows to correctly anchor sprites of any size on their collision box.

We normally choose as origin point the central point of contact between the entity and the ground. For example, the hero and the non-playing characters always have a collision box of `16x16`, and their origin point is always `8,13` (the center of their shadow). Sprites have to take care of this setting to obtain correct results.

Remarks

The origin point is the same thing as the "hotspot" property in Multimedia Fusion, except that in Solarus, it is the same for all frames of a direction.

3.10.3 Syntax of a sprite sheet file

Solarus Quest Editor fully supports the edition of sprites. You should not have to edit sprite data files by hand unless you know what you are doing.

The sequence of characters `--` (two dashes) marks the beginning of a comment. After them, the rest of the line is ignored by the engine. Empty lines are also ignored.

The definition of each animation starts with `animation{` and ends with `}`. Between the braces are defined the properties of the animation. Properties are declared with the syntax `key = value` and are separated with commas. It is allowed to have an extra comma after the last property. String values are enclosed within double quotes. The properties must be:

- `name` (string): A name identifying this animation.
- `src_image` (string): Name of a PNG image that contains all frames of all directions of this animation. It cannot contain spaces. Its path is relative to the directory `sprites`. Alternatively, `src_image` may also be the keyword `"tileset"`: in this case, it means that the sprite is tileset-dependent. Indeed, you may want to make sprites that change with the skin of the current map (like animated doors). The PNG file is then `tilesets/yy.entities.png` where `yy` is the id of the current tileset.
- `frame_delay` (number, optional): Delay in milliseconds between two frames of the animation. Zero or no value means an infinite delay. In this case, the sprite will show a fixed image and will never finish its animation.
- `frame_to_loop_on` (number, optional): Index of the frame where you want the animation to come back when the last frame finishes (the first frame is `0`). No value means no loop (the sprite stops being displayed after the last frame). Note that when `delay` is `0`, this can never happen since the first frame can never finish.
- `directions` (table): List of all directions (sequences of frames) of this animation. The list of directions is enclosed between `{` and `}`, where directions are separated by commas. Each direction defines a sequence of individual frames. In the PNG image, all frames of a direction are rectangles of the same size, placed in adjacent locations from left to right and possibly from top to bottom (in other words, organized in several columns and several rows). Therefore, instead of defining the frames one by one, we just specify the location of the first frame, the number of frames and the number of columns by row. This is enough to determine the exact position of each frame. The syntax of a direction is as follows. The direction is enclosed between `{` and `}` and contains some properties declared with the syntax `key = value` and separated by commas. A direction has the following properties:
  - `x` (number): X coordinate of the top-left corner of area containing the frames in the PNG image.
  - `y` (number): Y coordinate of the top-left corner of area containing the frames in the PNG image.
  - `frame_width` (number): Width of each frame in the PNG image.
  - `frame_height` (number): Height of each frame in the PNG image. Thus, the first frame is the rectangle of coordinates (`x`, `y`) and size (`frame_width`, `frame_height`).
  - `origin_x` (number, optional): X coordinate of the origin point of the sprite, relative to the upper-left rectangle of a frame. When a sprite is drawn at some coordinates, the origin point determines what exact point of the sprite's rectangle is at those coordinates. It is not necessarily the top-left corner of the rectangle. No values means `0`.
  - `origin_y` (number, optional): Y coordinate of the origin point of the sprite, relative to the upper-left rectangle of a frame. No values means `0`.
  - `num_frames` (number, optional): Number of frames of this direction. Most of the time, this number is the same for all directions. The default number of frames is `1`.

- `num_columns` (number, optional): Number of columns of the grid containing the frames of this direction in the PNG image. The number of rows is then deduced from `num_frames` and `num_columns`. Note that the last row of the grid may be incomplete. No value means `num_frames`, meaning that all frames are arranged in the only one row in the PNG image.

Example of a sprite animation set definition file:

```
animation{
  -- This animation is actually a fixed image (only one frame, no delay property).
  name = "stopped",
  src_image = "hero/stopped.tunic.png",
  directions = {
    { x =  0, y = 0, frame_width = 24, frame_height = 24, origin_x = 12, origin_y = 21 },
    { x = 24, y = 0, frame_width = 24, frame_height = 24, origin_x = 12, origin_y = 21 },
    { x = 72, y = 0, frame_width = 24, frame_height = 24, origin_x = 12, origin_y = 21 },
    { x = 96, y = 0, frame_width = 24, frame_height = 24, origin_x = 12, origin_y = 21 },
  }
}

animation{
  -- This animation loops on the first frame.
  name = "walking",
  src_image = "hero/walking.tunic.png",
  frame_delay = 100,
  frame_to_loop_on = 0,
  directions = {
    { x = 0, y =   0, frame_width = 24, frame_height = 32, origin_x = 12, origin_y = 29, num_frames = 8 },
    { x = 0, y =  32, frame_width = 24, frame_height = 32, origin_x = 12, origin_y = 29, num_frames = 8 },
    { x = 0, y =  96, frame_width = 24, frame_height = 32, origin_x = 12, origin_y = 29, num_frames = 8 },
    { x = 0, y = 128, frame_width = 24, frame_height = 32, origin_x = 12, origin_y = 29, num_frames = 8 },
  }
}

animation{
  -- This animation stops after its last frame (no frame_to_loop_on property).
  name = "sword",
  src_image = "hero/sword.tunic.png",
  frame_delay = 30,
  directions = {
    { x = 0, y =  0, frame_width = 32, frame_height = 32, origin_x = 16, origin_y = 29, num_frames = 12 },
    { x = 0, y = 32, frame_width = 32, frame_height = 32, origin_x = 16, origin_y = 29, num_frames = 12 },
    { x = 0, y = 64, frame_width = 32, frame_height = 32, origin_x = 16, origin_y = 29, num_frames = 12 },
    { x = 0, y = 96, frame_width = 32, frame_height = 32, origin_x = 16, origin_y = 29, num_frames = 12 },
  }
}
```

Don't forget to add all sprite animation sets to the quest tree in the editor (or by editing project_db.dat by hand). Otherwise, they won't show up when you configure entities with sprites (like non-playing characters) in the editor.

Remarks

The syntax of sprite data files is actually valid Lua. The engine and the editor internally use Lua to parse it.

3.11 Map definition file

The `maps` directory contains the full description and the script of each map.

Maps are areas where the game takes place. They may be rooms, houses, entire dungeon floors, parts of the outside world or any place.

A map with id `xx` is defined with two files:

- `xx.dat`: Definition of all entities of this map (tiles, enemies, non-playing characters, chests, teletransporters, etc.) and the properties of the map. This page will describe the syntax of this file.
- `xx.lua`: Lua script of the map. It defines the dynamic events that happen on your map at runtime, such as opening a door, making enemies appear, moving a non-playing character, etc. See the map API page for more details about map scripts.

A map is composed of the following information:

- A size in pixels.
- A tileset (the graphical skin used to draw its tiles).
- A background music.
- The world it belongs to (optional). Allows to groups maps together.
- Its location relative to its world.
- The floor it belongs to (if any).
- All entities of the map (tiles, enemies, chests, pots, switches, etc.), separated in different layers.

All of this is stored in the map data file `xx.dat`. We now specify its syntax.

3.11.1 Syntax of the map data file

Solarus Quest Editor fully supports the edition of maps. You should not have to edit map data files by hand unless you know what you are doing.

The sequence of characters `--` (two dashes) marks the beginning of a comment. After them, the rest of the line is ignored by the engine. Empty lines are also ignored.

3.11.1.1 Map properties

The first element in the map data file describes the properties of the map. The definition of these properties starts with `properties{` and ends with `}`. Properties are declared with the syntax `key = value` and separated with commas. It is allowed to have an extra comma after the last property. String values should be enclosed within double quotes. The following properties must be defined:

- `width` (number): Width of the map in pixels. Must be a multiple of 8.
- `height` (number): Height of the map in pixels. Must be a multiple of 8.
- `min_layer` (number): Index of the lowest layer (0 or less).
- `max_layer` (number): Index of the highest layer (0 or less).
- `world` (string, optional): A name that identifies a group of maps this map belongs to. Worlds allow to group maps together. The world can be any arbitrary name. Maps that have the same non-empty world name are considered to be part of the same environment. The starting location of the savegame is automatically set by the engine when the world changes (not when the map changes). See map:get_world() for more details.
- `x` (number, optional): X coordinate of the top-left corner of the map in its world. Useful to store the location of this map if it belongs to a group of adjacent maps. The engine uses this information to implement scrolling between two adjacent maps. The default value is `0`. See map:get_location() for more details.

- `y` (number, optional): Y coordinate of the top-left corner of the map in it world. The default is `0`.
- `floor` (number, optional): The floor of this map if it is part of a floor system. This property is optional. The engine does not do anything particular with the floor, but your quest can use it in scripts, for example to show the current floor on the HUD when it changes or to make a minimap menu. `0` is the first floor, `1` is the second floor, `-1` is the first basement floor, etc.
- `tileset` (string): Id of the tileset to use as a skin for this map.
- `music` (string, optional): Id of the music to play when entering this map. It can be a music file name relative to the `"musics"` directory (without extension), or the special value `"none"` to play no music on this map, or the special value `"same"` to keep the music unchanged. No value means no music.

Example of map properties definition:

```
properties{
  width = 2048,
  height = 2048,
  world = "outside",
  x = 0,
  y = 0,
  tileset = "overworld",
  music = "village",
}
```

3.11.1.2 Declaration of map entities

The rest of the map data file declares all entities (tiles, enemies, chests, etc.) initially present on the map when this map is loaded during the game.

Remarks

> Recall that during the game, after this loading phase, map entities can also be created or destroyed dynamically using the Lua map scripting API.

There exists many types of entities and most of them can be declared in the map data file. Here is their list (if you want all types of map entities, including the ones that cannot be declared in the map data file, see the map entity API):

- Tile: A small brick that composes a piece of the map, with a pattern picked from the tileset.
- Dynamic tile: A special tile that can be enabled or disabled dynamically (usual tiles are optimized away at runtime).
- Teletransporter: When walking on it, the hero is transported somewhere, possibly on the same map or another map.
- Destination: A possible destination place for teletransporters.
- Pickable treasure: A treasure placed on the ground and that the hero can pick up.
- Destructible object: An entity that can be cut or lifted by hero, and that may hide a pickable treasure.
- Chest: A chest that contains a treasure.
- Shop treasure: A treasure that the hero can buy for a price.
- Enemy: A bad guy (possibly a boss) who may also drop a pickable treasure when killed.
- Non-playing character (NPC): Somebody or something the hero can interact with.

- Block: An entity that the hero can push or pull.
- Jumper: When walking on it, the hero jumps into a direction.
- Switch: A button or another mechanism that the hero can activate.
- Sensor: An invisible detector that detects the presence of the hero.
- Wall: An invisible object that stops some kinds of entities.
- Crystal: A switch that lowers or raises crystal blocks.
- Crystal block: A low wall that can be lowered (travserable) or raised (obstacle) using a crystal.
- Stream: When walking on it, the hero automatically moves into a direction.
- Door: A door to open with an equipment item or another condition.
- Stairs: Stairs between two maps or to a platform of a single map.
- Separator: A visual separation between parts of the map.
- Custom entity: An entity fully customizable, with no built-in behavior.

The definition of a map entity starts with `entity_type{` and ends with `}`, where `entity_type` is a type of entity (see below). Inside the braces, the properties of the entity are specified. Properties are declared with the syntax `key = value` and separated with commas. It is allowed to have an extra comma after the last property. String values should be enclosed within double quotes.

In each layer, entities are ordered like they are declared in the map data file: the first entity in the map data file is the most to the back, the last one is the most to the front. But tiles are always below all other types of entities because they are optimized at runtime. Therefore, tiles are always declared first in the map data file for each layer.

3.11.1.2.1 Common properties

The following properties exist for all types of entities:

- `name` (string, optional): Name identifying the entity on the map. The name is optional: you will only need it if you need to refer to this entity individually, for example to make a switch open a door. If you define a name to an entity, it should be unique in the map. If the name is already used by another entity, a suffix (of the form `"_2"`, `"_3"`, etc.) will be automatically appended to keep entity names unique. All entities can have a name except tiles: this is because tiles don't exist individually at runtime for performance reasons.
- `layer` (number): Layer where the entity is on the map.
- `x` (number): X coordinate of the entity relative to the upper-left corner of the map. The origin point of the entity will be placed at these coordinates (for tiles, it is the upper-left corner).
- `y` (number): Y coordinate of the entity relative to the upper-left corner of the map.

`name`, `x`, `y` and `layer` are common to all types of entities (except `name` for tiles). We now detail the specific properties of each type of entity.

3.11.1.2.2 Tile

Tiles are the small fixed bricks that compose the map.

The engine makes a special performance treatment for them. For this reason, this map data file is the only place where they can be declared: they can never be accessed or created at runtime. If you want to access a tile at runtime (typically, to make it appear or disappear), use a dynamic tile instead.

A tile must be declared in the map data file with `tile{ ... }`.

Additional properties:

- `pattern` (string): Id of the tile pattern to use.
- `width` (number): Width of the tile in pixels. It must be a multiple of the width of the pattern. The pattern will be repeated horizontally to fit this width.
- `height` (number): Height of the tile in pixels. It must be a multiple of the height of the pattern. The pattern will be repeated vertically to fit this height.

Example of tile:

```
tile{
  layer = 0,
  x = 1656,
  y = 104,
  width = 96,
  height = 16,
  pattern = "dark_wall",
}
```

3.11.1.2.3 Dynamic tile

Dynamic tiles are like tiles, except that they are not optimized at runtime. Therefore, they can be enabled, disabled, created and deleted using the Lua API.

A dynamic tile can be declared in the map data file with `dynamic_tile{ ... }`.

Additional properties:

- `pattern` (string): Id of the tile pattern to use.
- `width` (number): Width of the tile in pixels. It must be a multiple of the width of the pattern. The pattern will be repeated horizontally to fit this width.
- `height` (number): Height of the tile in pixels. It must be a multiple of the height of the pattern. The pattern will be repeated vertically to fit this height.
- `enabled_at_start` (boolean): `true` to make the dynamic tile initially enabled, `false` to make it initially disabled.

Example of dynamic tile:

```
dynamic_tile{
  layer = 0,
  x = 552,
  y = 488,
  width = 48,
  height = 8,
  name = "bridge",
  pattern = "bridge_path",
  enabled_at_start = false,
}
```

3.11.1.2.4 Teletransporter

A teletransporter is a detector that sends the hero to another place when he walks on it.

A teletransporter can be declared in the map data file with `teletransporter{ ... }`.

Additional properties:

- `width` (number): Width of the teletransporter in pixels.
- `height` (number): Height of the teletransporter in pixels.
- `sprite` (string, optional): Id of the animation set of a sprite to create for the teletransporter. No value means no sprite (the teletransporter will then be invisible).
- `sound` (string, optional): Sound to play when the hero uses the teletransporter. No value means no sound.
- `transition` (string, optional): Style of transition to play when the hero uses the teletransporter. Must be one of:
  - `"immediate"`: No transition.
  - `"fade"`: Fade-out and fade-in effect.
  - `"scrolling"`: Scrolling between maps. The default value is `"fade"`.
- `destination_map` (string): Id of the map to transport to (can be the id of the current map).
- `destination` (string, optional): Location on the destination map. Can be the name of a destination entity, the special value `"_same"` to keep the hero's coordinates, or the special value `"_side"` to place on hero on the corresponding side of an adjacent map (normally used with the scrolling transition style). No value means the default destination entity of the map.

Example of teletransporter:

```
teletransporter{
  layer = 1,
  x = 112,
  y = 600,
  width = 16,
  height = 16,
  transition = "fade",
  destination_map = "dungeon_1_1f",
  destination = "from_outside",
}
```

3.11.1.2.5 Destination

A destination is a possible arrival place for teletransporters.

A destination can be declared in the map data file with `destination{ ... }`.

Additional properties:

- `direction` (number): Direction that the hero should take when arriving on the destination, between `0` (East) and `3` (South), or `-1` to keep his direction unchanged.
- `sprite` (string, optional): Name of the animation set of a sprite to create for the destination. No value means no sprite (the destination will then be invisible).

- `starting_location_mode` (string, optional): Whether to update the starting location of the player when arriving to this destination. If yes, when the player restarts his game, he will restart at this destination. Must be one of:
  - `"when_world_changes"` (default): Updates the starting location if the current world has just changed when arriving to this destination.
  - `"yes"`: Updates the starting location.
  - `"no"`: Does not update the starting location.
- `default` (boolean, optional): Sets this destination as the default one when teletransporting the hero to this map without destination specified. No value means `false`. Only one destination can be the default one on a map. If no default destination is set, then the lowest one in Z order is set as the default one.

Example of destination:

```
destination{
  name = "from_outside",
  layer = 0,
  x = 160,
  y = 37,
  direction = 1,
  default = true,
}
```

3.11.1.2.6 Pickable treasure

A pickable treasure is a treasure on the ground that the hero can pick up.

A pickable treasure can be declared in the map data file with `pickable{ ... }`.

Additional properties:

- `treasure_name` (string, optional): Kind of treasure to create (the id of an equipment item). If this value is not set, or corresponds to a non-obtainable item, then the pickable treasure won't be created.
- `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
- `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this pickable treasure was found. No value means that the treasure is not saved. If the treasure is saved and the player already has it, then the pickable treasure won't be created.

Example of pickable treasure:

```
pickable{
  layer = 0,
  x = 216,
  y = 845,
  treasure_name = "rupee",
  treasure_variant = 3,
  treasure_savegame_variable = "castle_rupee_1_found",
}
```

3.11.1.2.7 Destructible object

A destructible object is an entity that can be cut or lifted by the hero and that may hide a pickable treasure.

A destructible object can be declared in the map data file with `destructible{ ... }`.

Additional properties:

- `treasure_name` (string, optional): Kind of pickable treasure to hide in the destructible object (the id of an equipment item). If this value is not set, then no treasure is placed in the destructible object. If the treasure is not obtainable when the object is destroyed, no pickable treasure is be created.
- `treasure_variant` (number, optional): Variant of the treasure if any (because some equipment items may have several variants). The default value is `1` (the first variant).
- `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the pickable treasure hidden in the destructible object was found. No value means that the treasure (if any) is not saved. If the treasure is saved and the player already has it, then no treasure is put in the destructible object.
- `sprite` (string): Name of the animation set of a sprite to create for the destructible object.
- `destruction_sound` (string, optional): Sound to play when the destructible object is cut or broken after being thrown. No value means no sound.
- `weight` (number, optional): Level of `"lift"` ability required to lift the object. `0` allows the player to lift the object unconditionally. The special value `-1` means that the object can never be lifted. The default value is `0`.
- `can_be_cut` (boolean, optional): Whether the hero can cut the object with the sword. No value means `false`.
- `can_explode` (boolean, optional): Whether the object should explode when it is cut, hit by a weapon and after a delay when the hero lifts it. The default value is `false`.
- `can_regenerate` (boolean, optional): Whether the object should automatically regenerate after a delay when it is destroyed. The default value is `false`.
- `damage_on_enemies` (number, optional): Number of life points to remove from an enemy that gets hit by this object after the hero throws it. If the value is `0`, enemies will ignore the object. The default value is `1`.
- `ground` (string, optional): Ground defined by this entity. The ground is usually `"wall"`, but you may set `"traversable"` to make the object traversable, or for example `"grass"` to make it traversable too but with an additional grass sprite below the hero. See map:get_ground() for the list of grounds. The default value is `"wall"`.

Examples of destructible objects:

```
destructible{
  layer = 0,
  x = 232,
  y = 165,
  treasure_name = "heart",
  sprite = "entities/white_stone",
  destruction_sound = "stone",
  weight = 1,
  damage_on_enemies = 2,
}

destructible{
  layer = 0,
  x = 264,
  y = 165,
  treasure_name = "heart",
  sprite = "entities/grass",
  can_be_cut = true,
  ground = "grass",
}
```

3.11.1.2.8 Chest

A chest is a box that contains a treasure.

A chest can be declared in the map data file with `chest{ ... }`.

Additional properties:

- `treasure_name` (string, optional): Kind of treasure to place in the chest (the name of an equipment item). If this value is not set, then the chest will be empty. If the treasure is not obtainable when the hero opens the chest, it becomes empty.
- `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
- `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this chest is open. No value means that the state of the treasure is not saved. If the treasure is saved and the player already has it, then no treasure is placed in the chest (the chest will appear open).
- `sprite` (string): Id of the animation set of the sprite to create for the chest. The sprite must have animations `"open"` and `"closed"`.
- `opening_method` (string, optional): Specifies the permissions for the hero to open the chest. Must be one of:
  - `"interaction"` (default): Can be opened by pressing the action command in front of the chest.
  - `"interaction_if_savegame_variable"`: Can be opened by pressing the action command in front of the chest, provided that a specific savegame variable is set.
  - `"interaction_if_item"`: Can be opened by pressing the action command in front of the chest, provided that the player has a specific equipment item.
- `opening_condition` (string, optional): The condition required to open the chest. Only for opening methods `"interaction_if_savegame_variable"` and `"interaction_if_item"`.
  - For opening method `"interaction_if_savegame_variable"`, it must be the name of a savegame variable. The hero will be allowed to open the chest if this saved value is either `true`, an integer greater than zero or a non-empty string.
  - For opening method `"interaction_if_item"`, it must be the name of an equipment item. The hero will be allowed to open the chest if he has that item and, for items with an amount, if the amount is greater than zero.
  - For the default opening method (`"interaction"`), this setting has no effect.
- `opening_condition_consumed` (boolean, optional): Whether opening the chest should consume the savegame variable or the equipment item that was required. The default setting is `false`. If you set it to `true`, the following rules are applied when the hero successfully opens the chest:
  - For opening method `"interaction_if_savegame_variable"`, the savegame variable that was required is reset to `false`, `0` or `""` (depending on its type).
  - For opening method is `"interaction_if_item"`, the equipment item that was required is removed. This means setting its possessed variant to `0`, unless it has an associated amount: in this case, the amount is decremented.
- `cannot_open_dialog` (string, optional): Id of the dialog to show if the hero fails to open the chest. If you don't set this value, no dialog is shown.

Example of chest:

```
chest{
  layer = 1,
  x = 168,
  y = 645,
  treasure_name = "sword",
  treasure_variant = 2,
  treasure_savegame_variable = "dungeon_6_sword_2_found",
  sprite = "entities/chest",
}
```

3.11.1.2.9 Shop treasure

A shop treasure is a treasure that can be purchased by the hero for money.

A shop treasure can be declared in the map data file with `shop_item{ ... }`.

Additional properties:

- `price` (number): Money amount required to buy the treasure.
- `font` (string, optional): Id of the font to use to display to price. The default value is the first one in alphabetical order.
- `dialog` (string): Id of the dialog to show when the player asks for information about the treasure.
- `treasure_name` (string): Kind of treasure to sell (the name of an equipment item). If this value or corresponds to a non-obtainable item, then the shop treasure is not created.
- `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).
- `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the player has purchased this treasure. No value means that the state of the treasure is not saved. If the treasure is saved and the player already has it, then the shop treasure is not created.

Example of shop treasure:

```
shop_item{
  layer = 1,
  x = 200,
  y = 104,
  treasure_name = "health_potion",
  price = 160,
  dialog = "witch_shop.health_potion",
}
```

3.11.1.2.10 Enemy

An enemy is a bad guy that hurts the hero when touching him.

An enemy can be declared in the map data file with `enemy{ ... }`.

Additional properties:

- `direction` (number): Initial direction of the enemy, between `0` (East) and `3` (South).
- `breed` (string): Model of enemy to create.
- `savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this enemy is dead. No value means that the enemy is not saved. If the enemy is saved and was already killed, then no enemy is created. Instead, its pickable treasure is created if it is a saved one.
- `treasure_name` (string, optional): Kind of pickable treasure to drop when the enemy is killed (the name of an equipment item). If this value is not set, then the enemy won't drop anything. If the treasure is not obtainable when the enemy is killed, then nothing is dropped either.
- `treasure_variant` (number, optional): Variant of the treasure (because some equipment items may have several variants). The default value is `1` (the first variant).

- `treasure_savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether the pickable treasure of this enemy was obtained. No value means that the state of the treasure is not saved. If the treasure is saved and the player already has it, then the enemy won't drop anything.

Example of enemy:

```
enemy{
  layer = 0,
  x = 912,
  y = 453,
  direction = 3,
  breed = "knight_soldier",
  treasure_name = "rupee",
}
```

3.11.1.2.11 Non-playing character

A non-playing character (NPC) is somebody or something that the hero can interact with by pressing the action command or by using an equipment item just in front of it.

A non-playing character can be declared in the map data file with `npc{ ... }`.

Additional properties:

- `direction` (number): Initial direction of the NPC's sprite, between `0` (East) and `3` (South).
- `subtype` (number): Kind of NPC to create: `1` for a usual NPC who the player can talk to, `0` for a generalized NPC (not necessarily a person). See the non-playing characters API for more details.
- `sprite` (string, optional): Id of the animation set of a sprite to create for the NPC. No value means no sprite (the NPC will then be invisible).
- `behavior` (string, optional): What to do when there is an interaction with the NPC.
  - `"dialog#XXXX"`: Start the dialog with id `"XXXX"` when the player talks to this NPC.
  - `"map"` (default): Notify the map Lua script.
  - `"item#XXXX"`: Notify an equipment item Lua script.

Example of non-playing character:

```
npc{
  layer = 0,
  x = 648,
  y = 389,
  direction = 3,
  subtype = "0",
  sprite = "entities/sign",
  behavior = "dialog#outside_world.old_castle_sign",
}
```

3.11.1.2.12 Block

Blocks are solid entities that may be pushed or pulled by the hero.

A block can be declared in the map data file with `block{ ... }`.

Additional properties:

- `direction` (number, optional): The only direction where the block can be moved, between `0` (East) and `3` (South). No value means no restriction and allows the block to be moved in any of the four main directions.
- `sprite` (string): Name of the animation set of a sprite to create for the block.
- `pushable` (boolean): `true` to allow the block to be pushed.
- `pullable` (boolean): `true` to allow the block to be pulled.
- `maximum_moves` (number): `Indicates` how many times the block can be moved (`0`: none, `1`: once, `2`: infinite).

Example of block:

```
block{
  layer = 1,
  x = 584,
  y = 69,
  direction = 3,
  sprite = "entities/block",
  pushable = true,
  pullable = false,
  maximum_moves = 1,
}
```

3.11.1.2.13 Jumper

A jumper is an invisible detector that makes the hero jump into one of the 8 main directions when touching it.

A jumper can be declared in the map data file with `jumper{ ... }`.

Additional properties:

- `width` (number): Width of the jumper in pixels.
- `height` (number): Height of the jumper in pixels.
- `direction` (number): Direction of the jump, between `0` (East) and `7` (South-East). If the direction is horizontal, the width must be `8` pixels. If the direction is vertical, the height must be `8` pixels. If the direction is diagonal, the size must be square.
- `jump_length` (number): Length of the baseline of the jump in pixels (see the jump movement page for details).

Example of jumper:

```
jumper{
  layer = 0,
  x = 192,
  y = 824,
  width = 8,
  height = 32,
  direction = 6,
  jump_length = 40,
}
```

3.11.1.2.14 Switch

A switch is a button that can be activated to trigger a mechanism.

A switch can be declared in the map data file with `switch{ ... }`.

Additional properties:

- `subtype` (string): Kind of switch to create:
  - `"walkable"`: A traversable pressure plate that gets activated when the hero walks on it.
  - `"solid"`: A non-traversable, solid switch that can be activated in various conditions: by the sword, by an explosion or by a projectile (a thrown object, an arrow, the boomerang or the hookshot).
  - `"arrow_target"` A switch that can be only activated by shooting an arrow on it.
- `sprite` (string): Name of the animation set of a sprite to create for the switch. The animation set must at least contain animations `"activated"` and `"inactivated"`. No value means no sprite.
- `sound` (string, optional): Sound to play when the switch is activated. No value means no sound.
- `needs_block` (boolean): If `true`, the switch can only be activated by a block (only for a walkable switch).
- `inactivate_when_leaving` (boolean): If `true`, the switch becomes inactivated when the hero or the block leaves it (only for a walkable switch).

Example of switch:

```
switch{
  name = "open_door_2_switch",
  layer = 1,
  x = 376,
  y = 152,
  subtype = "walkable",
  sprite = "entities/switch",
  sound = "switch_pressed",
  needs_block = false,
  inactivate_when_leaving = true,
}
```

3.11.1.2.15 Sensor

A sensor is an invisible detector that triggers something when the hero overlaps it.

A sensor can be declared in the map data file with `sensor{ ... }`.

Additional properties:

- `width` (number): Width of the sensor in pixels.
- `height` (number): Height of the sensor in pixels.

Example of sensor:

```
sensor{
  name = "start_boss_sensor",
  layer = 0,
  x = 392,
  y = 301,
  width = 16,
  height = 16,
}
```

3.11.1.2.16 Wall

A wall is an invisible obstacle that stops some specific types of map entities.

A wall can be declared in the map data file with `wall{ ... }`.

Additional properties:

- `width` (number): Width of the wall in pixels.
- `height` (number): Height of the wall in pixels.
- `stops_hero` (boolean, optional): `true` to make the wall stop the hero. No value means `false`.
- `stops_npcs` (boolean, optional): `true` to make the wall stop non-playing characters. No value means `false`.
- `stops_enemies` (boolean, optional): `true` to make the wall stop enemies. No value means `false`.
- `stops_blocks` (boolean, optional): `true` to make the wall stop blocks. No value means `false`.
- `stops_projectiles` (boolean, optional): `true` to make the wall stop projectiles: thrown objects, arrows, the hookshot and the boomerang. No value means `false`.

Example of wall:

```
wall{
  layer = 1,
  x = 416,
  y = 256,
  width = 24,
  height = 8,
  stops_hero = false,
  stops_enemies = true,
  stops_npcs = true,
  stops_blocks = true,
  stops_projectiles = true,
}
```

3.11.1.2.17 Crystal

A crystal is a switch that lowers or raises alternatively some special colored blocks in the ground called crystal blocks.

A crystal can be declared in the map data file with `crystal{ ... }`.

Additional properties: none.

Example of crystal:

```
crystal{
  layer = 0,
  x = 176,
  y = 965,
}
```

3.11.1.2.18 Crystal block

A crystal block is a colored low wall that may be raised or lowered in the ground.

A crystal block can be declared in the map data file with `crystal_block{ ... }`.

Additional properties:

- `width` (number): Width of the crystal block in pixels (must be a multiple of 16 pixels). The crystal block pattern will be repeated to fit the size.
- `height` (number): Height of the crystal block in pixels (must be a multiple of 16 pixels). The crystal block pattern will be repeated to fit the size.
- `subtype` (number): Kind of crystal block to create: `0` for a block initially lowered (orange), `1` for a block initially raised (blue).

Example of crystal block:

```
crystal_block{
  layer = 0,
  x = 120,
  y = 856,
  width = 112,
  height = 16,
  subtype = 1,
}
```

3.11.1.2.19 Stream

When walking on a stream, the hero automatically moves into one of the 8 main directions. The stream can allow or not the hero to move and use items. Streams can be used for example to make conveyor belts, water streams or air streams.

A stream can be declared in the map data file with `stream{ ... }`.

Additional properties:

- `direction` (number): Direction where the stream moves the hero, between `0` (East) and `7` (South-East).
- `sprite` (string, optional): Id of the animation set of a sprite to create for the stream. No value means no sprite (the stream will then be invisible).
- `speed` (number, optional): Speed of the movement applied to the hero by the stream, in pixels per second. The default value is `64`. A speed of `0` means that the hero will never be moved by the stream.
- `allow_movement` (boolean, optional): Whether the player can still move the hero when he is on the stream. The default value is `true`.
- `allow_attack` (boolean, optional): Whether the player can use the sword when he is on the stream. The default value is `true`.
- `allow_item` (boolean, optional): Whether the player can use equipment items when he is on the stream. The default value is `true`.

Example of stream:

```
stream{
  layer = 1,
  x = 200,
  y = 157,
  direction = 5,
  sprite = "entities/air_stream",
  allow_movement = false,
  allow_attack = true,
  allow_item = true,
}
```

3.11.1.2.20 Door

A door is an obstacle that can be opened by Lua scripts and optionally by the hero under some conditions.

A door can be declared in the map data file with `door{ ... }`.

Additional properties:

- `direction` (number): Direction of the door, between `0` (East of the room) and `3` (South of the room).
- `sprite` (string): Id of the animation set of the sprite to create for the door. The sprite must have an animation `"closed"`, that will be shown while the door is closed. When the door is open, no sprite is displayed. Optionally, the sprite can also have animations `"opening"` and `"closing"`, that will be shown (if they exist) while the door is being opened or closed, respectively. If they don't exist, the door will open or close instantly.
- `savegame_variable` (string, optional): Name of the boolean value that stores in the savegame whether this door is open. No value means that the door is not saved. If the door is saved as open, then it appears open.
- `opening_method` (string, optional): How the door is supposed to be opened by the player. Must be one of:
  - `"none"` (default): Cannot be opened by the player. You can only open it from a Lua script.
  - `"interaction"`: Can be opened by pressing the action command in front of the door.
  - `"interaction_if_savegame_variable"`: Can be opened by pressing the action command in front of the door, provided that a specific savegame variable is set.
  - `"interaction_if_item"`: Can be opened by pressing the action command in front of the door, provided that the player has a specific equipment item.
  - `"explosion"`: Can be opened by an explosion.
- `opening_condition` (string, optional): The condition required to open the door. Only for opening methods `"interaction_if_savegame_variable"` and `"interaction_if_item"`.
  - For opening method `"interaction_if_savegame_variable"`, it must be the name of a savegame variable. The hero will be allowed to open the door if this saved value is either `true`, an integer greater than zero or a non-empty string.
  - For opening method `"interaction_if_item"`, it must be the id of an equipment item. The hero will be allowed to open the door if he has that item and, for items with an amount, if the amount is greater than zero.
  - For other opening methods, this setting has no effect.
- `opening_condition_consumed` (boolean, optional): Whether opening the door should consume the savegame variable or the equipment item that was required. The default setting is `false`. If you set it to `true`, the following rules are applied when the hero successfully opens the door:
  - For opening method `"interaction_if_savegame_variable"`, the savegame variable that was required is reset to `false`, `0` or `""` (depending on its type).
  - For opening method is `"interaction_if_item"`, the equipment item that was required is removed. This means setting its possessed variant to `0`, unless it has an associated amount: in this case, the amount is decremented.
  - With other opening methods, this setting has no effect.
- `cannot_open_dialog` (string, optional): Id of the dialog to show if the hero fails to open the door. If you don't set this value, no dialog is shown.

Example of door:

```
door{
  layer = 1,
  x = 872,
  y = 184,
  direction = 2,
  sprite = "entities/door_small_key",
  savegame_variable = "dungeon_6_locked_door_1_open",
  opening_method = "interaction_if_savegame_variable",
  opening_condition = "dungeon_6_small_keys",
  opening_condition_consumed = true,
  cannot_open_dialog = "small_key_required",
}
```

3.11.1.2.21 Stairs

Stairs make fancy animations, movements and sounds when the hero takes them between two maps or to a platform of a single map.

Stairs can be declared in the map data file with `stairs{ ... }`.

Additional properties:

- `direction` (number): Direction where the stairs should be turned between `0` (East of the room) and `3` (South of the room). For stairs inside a single floor, this is the direction of going upstairs.
- `subtype` (number): Kind of stairs to create:
  - `0`: Spiral staircase going upstairs.
  - `1`: Spiral staircase going downstairs.
  - `2`: Straight staircase going upstairs.
  - `3`: Straight staircase going downstairs.
  - `4`: Small stairs inside a single floor (change the layer of the hero).

Example of stairs:

```
stairs{
  layer = 0,
  x = 384,
  y = 272,
  direction = 3,
  subtype = "1",
}
```

3.11.1.2.22 Separator

Separators allow to visually separate different regions of a map like if they were several maps. When the camera touches the separation, it stops like if there was a limit of a map. If the hero touches the separation, he scrolls to the other side.

A separator can be declared in the map data file with `separator{ ... }`.

Additional properties:

- `width` (number): Width of the separator in pixels.
- `height` (number): Height of the separator in pixels. One of `width` or `height` must be 16 pixels.

Example of separator:

```
separator{
  layer = 2,
  x = 320,
  y = 0,
  width = 16,
  height = 1280,
}
```

3.11.1.2.23 Custom entity

Custom entities have no special properties or behavior. You can define them entirely in your scripts. Optionally, a custom entity may be managed by model. The model is the name of a Lua script that will be applied to all custom entities refering to it. This works exactly like the breed of enemies, except that it is optional. The model is useful if you have a lot of identical (or very similar) custom entities in your game, like for example torches.

A custom entity can be declared in the map data file with `custom_entity{ ... }`.

Additional properties:

- `direction` (number): Direction of the custom entity, between `0` (East) and `3` (South). This direction will be applied to the entity's sprite if possible.
- `width` (number): Width of the entity in pixels.
- `height` (number): Height of the entity in pixels.
- `sprite` (string, optional): Id of the animation set of a sprite to create for the custom entity. No value means no sprite (the custom entity will then be invisible).
- `model` (string, optional): Model of custom entity to create. The model is the name of a Lua script in the `"entities"` directory of your quest. It will define the behavior of your entity. No value means no model: in this case, no particular script will be called but you can still define the behavior of your entity in the map script.

Examples of custom entities:

```
-- A cannon that exists only at this place of the game.
custom_entity{
  name = "cannon"
  layer = 0,
  x = 496,
  y = 232,
  width = 16,
  height = 16,
  sprite = "entities/cannon",
}

-- A minecart.
-- Managed by a model because several minecarts may exist in the game.
custom_entity{
  layer = 0,
  x = 152,
  y = 272,
  width = 16,
  height = 16,
  model = "minecart",
}
```

Remarks

The syntax of map data files is actually valid Lua. The engine and the editor internally use Lua to parse it.

3.12 Tileset definition file

Maps are composed of small bricks called tiles. Tiles are static entities: they cannot be moved or removed dynamically during the game. They are opposed to all dynamic entities such as enemies, doors, bushes, switches, etc. Each tile is a graphical element whose width and height are multiple of 8 pixels. When placed on a map, tiles can have different sizes and can overlap each other.

A tileset is a list of tiles that a map can use. You can call it the skin of the map. For example, there may exist a tileset for the outside world, another one for houses, another one for dungeons, etc.

The `tilesets` directory contains all tilesets and their images. For a tileset with id `xx`, there are three files:

- `xx.tiles.png`: The PNG image containing all tiles patterns of this tileset.
- `xx.dat`: The definition of each tile in `xx.tiles.png` and their properties. We detail the syntax of this file below.
- `xx.entities.png`: some dynamic entities (i.e. other that tiles), such as doors and blocks, that also have graphics that depend on the skin. A door inside a dungeon has different graphics and colors from a door inside a cave. This is the purpose of this file: sprites that depend on the tileset pick their frames in this image.

Tile patterns defined in `xx.tiles.png` can be fixed or animated with several frames. Examples of uses of multi-frame patterns are water, flowers and lava.

Tile patterns can also have special scrolling effects like parallax scrolling.

3.12.1 Syntax of the tileset data file

Solarus Quest Editor fully supports the edition of tilesets. You should not have to edit tileset data files by hand unless you know what you are doing.

We now specify the syntax of a tileset data file.

The sequence of characters `--` (two dashes) marks the beginning of a comment. After them, the rest of the line is ignored by the engine. Empty lines are also ignored.

3.12.1.1 Background color

The first element in the tileset file is the background color of the maps using this tileset. The definition of this background color starts with `background_color{` and ends with `}`. Inside the braces, the background color is specified in RGB format: three integers between 0 and 255, separated by commas. It is allowed to have an extra comma after the last integer value.

Example of background color definition:

```
background_color{ 104, 184, 104 }
```

3.12.1.2 Tile pattern definitions

The rest of the tileset data file defines all tile patterns of the tileset. The definition of a tile pattern starts with `tile_pattern{` and ends with `}`. Inside the braces, the properties of the tile pattern are specified. Properties are declared with the syntax `key = value` and separated with commas. It is allowed to have an extra comma after the last property. String values should be enclosed within double quotes. Each tile pattern must have the following properties:

- `id` (string): Id of the tile pattern. Maps refer to this id when they want to use the tile pattern. It must be unique in the tileset.
- `ground` (string): Kind of ground. Must be one of:
  - `"empty"`: There is no ground, this tile is purely decorative. Keep the ground of the tile below instead (if any).
  - `"traversable"`: Normal ground without obstacle.
  - `"wall"`: Cannot be traversed.
  - `"low_wall"`: Cannot only be traversed by projectiles like thrown items, arrows, the boomerang or flying enemies.
  - `"wall_top_right"`: The upper-right half of the tile is an obstacle.
  - `"wall_top_left"`: The upper-left halt of the tile is an obstacle.
  - `"wall_bottom_left"`: The lower-left half of the tile is an obstacle.
  - `"wall_bottom_right"`: The lower-right half of the tile is an obstacle.
  - `"wall_top_right_water"`: The upper-right half of the tile is an obstacle, the rest is deep water.
  - `"wall_top_left_water"`: The upper-left half of the tile is an obstacle, the rest is deep water.
  - `"wall_bottom_left_water"`: The lower-left half of the tile is an obstacle, the rest is deep water.
  - `"wall_bottom_right_water"`: The lower-right half of the tile is an obstacle, the rest is deep water.
  - `"deep_water"`: Deep water (the hero drowns or swims).
  - `"shallow_water"`: Shallow water (the hero can walk on it).
  - `"grass"`: Displays some grass under the hero.
  - `"hole"`: A hole (the hero falls into it).
  - `"ice"`: Ice (the hero slides).
  - `"ladder"`: A ladder (the hero walks slowly on it).
  - `"prickles"`: Untraversable spikes (it hurts the hero).
  - `"lava"`: Lava (the hero drowns and gets hurt).
- `default_layer` (number): The initial layer to be used when creating a tile with this pattern on a map with a map editor. This information is only useful for map editors (the engine does not use it).
- `x` (number or table): X coordinate(s) of the upper-left corner of the pattern's rectangle(s) in the tileset image file. The rectangle of a tile pattern must not overlap other tile patterns in the tileset. If it is a number, the tile pattern will be a single-frame tile pattern. If it is a table, it must have the syntax `{ x1, x2, x3 }` or `{ x1, x2, x3, x2 }` and the tile pattern will be a multi-frame tile pattern (an animated one). Multi-frame tile patterns play a sequence of three frames, either by repeating the x1-x2-x3 sequence or the x1-x2-x3-x2 sequence. The three rectangles of multi-frame tile patterns must be adjacent in the tileset image, and placed horizontally (from left to right) or vertically (from top to bottom).
- `y` (number or table): Y coordinate(s) of the upper-left corner of the pattern's rectangle(s) in the tileset image file. Same rules as `x`. Note that in the case of a multi-frame tile pattern, the table must have the same number of elements as `x`, and that due to the horizontal or vertical placement of the three frames in the tileset image, one of `x` or `y` will necessarily be a table with three identical values.

- `width` (number): Width of the pattern's rectangle(s) in pixels in the tileset image. In the case of a multi-frame tile pattern, this is the width of an individual rectangle. All three rectangles have the same size. Note that there is a redundancy: since the three rectangle must be adjacent, the size is actually implied by `x` and `y`.
- `height` (number): Height of the pattern's rectangle(s) in pixels in the tileset image. In the case of a multi-frame tile pattern, this is the height of an individual rectangle. All three rectangles have the same size. Note that there is a redundancy: since the three rectangle must be adjacent, the size is actually implied by `x` and `y`.
- `scrolling` (string, optional): Applies a scrolling effect on the tile pattern. No value means no special scrolling. If this value is defined, it must be one of:
  - `"parallax"`: Parallax scrolling. When the camera moves, the place where the pattern is displayed also moves, but twice slower, to give an illusion of depth. Placing such tiles on your map can be tricky because at runtime, they are displayed somewhere else (except when the camera is at the upper-left corner of the map). Note that collisions properties remain applied to the real position of the tiles and not to where they are displayed. Usually, parallax scrolling is used for decoration only. Parallax scrolling tile pattern can also be multi-frame. Example of use: a rich background panorama.
  - `"self"`: When the camera moves, the pattern scrolls on itself, but twice slower than the camera. This achieves the same effect as parallax scrolling, but the tile remains displayed at its position. However, this is limited to simple, repeatable patterns: you cannot make big scrolling backgrounds composed of different tiles (use parallax patterns for this) because nothing actually moves with the camera, tiles just scroll on themselves. Self-scrolling tile patterns cannot be multi-frame. Example of use: holes in dungeons.
- `repeat_mode` (string, optional): How the pattern is supposed to be repeatable. This property is only useful for map editors (the engine does not use it). Map editors can take advantage of it to resize patterns only in the way they are intended. It can be one of:
  - `"all"` (default): The pattern can be repeated both horizontally and vertically.
  - `"horizontal"`: The pattern can only be repeated horizontally.
  - `"vertical"`: The pattern can only be repeated vertically.
  - `"none"`: The pattern cannot be repeated.

Remarks

While the syntax of multi-frame patterns is redundant, it is flexible and will remain valid if one day we decide to remove the rules about the number of frames or their adjacent position in the tileset image.

Example of a valid tileset file:

```
background_color{ 104, 184, 104 }

tile_pattern{
  id = "path_dirt",
  ground = "traversable",
  default_layer = 0,
  x = 32,
  y = 0,
  width = 16,
  height = 16,
}

tile_pattern{
  id = "ocean",
  ground = "deep_water",
  default_layer = 0,
  x = { 0, 8, 16 },
  y = { 32, 32, 32 },
  width = 8,
  height = 8,
}
```

```
tile_pattern{
  id = "parallax_tree",
  ground = "traversable",
  default_layer = 0,
  x = 320,
  y = 448,
  width = 16,
  height = 24,
  scrolling = "parallax",
}
```

Remarks

This syntax of the tileset data file is actually valid Lua. The engine and the editor internally use Lua to parse it.

Chapter 4

How to translate a quest

We describe here how to translate a Solarus quest. If you wish to contribute translate one of our games, feel free to contact us.

Several files have to be translated so that a quest works in a new language:

- some images containing text,
- some strings displayed in the HUD and the menu,
- all dialogs of the game (most of the work to do is here).

All these language-specific data files are in the directory `languages/xx` of the quest data (where `xx` is the code of the language, for instance `en` or `fr`).

4.1 Images

Some images used by the game engine contain text. They all are in the directory `languages/xx/images` If you are making a translation of one of our games, we can do the pixel work for you, just give us the textual translations. We can give you explanations about all these images so that you can know when they are used.

4.2 Strings

The engine displays some text in the menus, before and during the game. All these strings are loaded from the file `languages/xx/text/strings.dat`. The file must be encoded in UTF-8 and respect the syntax specified in the Translated strings page.

4.3 Dialogs

Dialogs are all messages displayed to the player in the dialog box during the game. They represent the biggest part of the translation work. All dialogs are defined in the file `languages/xx/text/dialogs.dat`. The syntax of this file is specified in the Translated dialogs page. However, this is a very extensible syntax because the dialog box system is entirely customizable. Each game extends the syntax of `dialogs.dat` in its own way by adding custom properties.

If you want to translate one of our games, we give translation instructions and describe precisely these custom properties in the comments of the `dialogs.dat` file itself.