SpringBoard Documentation

Release 0.5

gajop

Aug 19, 2022

Contents

SpringBoard is an in-game editor for the SpringRTS engine, and it can be used to develop maps and scenarios.

CHAPTER 1

Installing

The simplest way to download SpringBoard is by using one of the provided installers:

- Windows build
- Linux build

For details refer to *Installing*.

CHAPTER 2

Game-specific modules

This is the core module, you may still want to get additional game-specific modules if you're making a scenario.

Some examples:

- https://github.com/Spring-SpringBoard/SpringBoard-BA
- https://github.com/Spring-SpringBoard/SpringBoard-ZK
- https://github.com/Spring-SpringBoard/SpringBoard-EVO
- https://github.com/Spring-SpringBoard/SpringBoard-S44

CHAPTER 3

Assets

A set of core assets are available. You can download them either via the launcher's `Asset Download` option (recommended), or manually via a direct link. In case of a manual download, you need to extract them to `springboard/assets/core/`.

CHAPTER 4

Help

Please post any questions, bugs and feature requests as Github issues.

For realtime troubleshooting feel free to join us on Discord in the #springboard channel.

CHAPTER 5

Contents

5.1 Installing

5.1.1 Using packaged builds

The simplest way to download SpringBoard is by using one of the provided installers. They will automatically download all of the needed resources (engine, editor archives, maps) and setup files, and launch SpringBoard. They will also check for updates on launch and keep the editor updated.

Packages:

- Windows build
- Linux build

Once you have downloaded one of the above files, simply run them and install as necessary. After installation, it will download the necessary files and launch SpringBoard itself.

Note: Linux users need to make the downloaded .AppImage file executable, by doing `chmod +x SpringBoard.AppImage`

5.1.2 Manual setup

It is also possible to manually setup SpringBoard. Please refer to the SpringRTS documentation for downloading and installing the engine and using pr-downloader.

The production version can be obtained from rapid, via: `pr-downloader sbc:test`

The development version can be obtained from this repository, by cloning it in your game folder: `git clone https://github.com/Spring-SpringBoard/SpringBoard-Core SB-C.sdd`

Note: Games can distribute their packages differently. Some games might include the editor as part of the ingame lobby. For more information, consult the game manual.

5.1.3 Hardware requirements

SpringBoard runs on most machines that support the SpringRTS engine, with the requirements described here. The only additional requirement is that the Graphics Card drivers must support basic OpenGL Shaders (GLSL). Most modern GPUs should be usable, but it is necessary to ensure the system has newest OpenGL drivers (see this for download instructions).

Additionally, for better performance having a good GPU will make terrain texture editing more efficient, while having a decent CPU and more RAM will make heightmap editing and scenario editing work more smoothly.

5.1.4 Software requirements

Linux has additional software requirements:

- SDL (Ubuntu package: *libsdl2-2.0-0*)
- OpenAL (Ubuntu package: *libopenal1*)

5.2 Starting out

This section introduces the layout of SpringBoard. First make sure you have properly *installed* SpringBoard. If you are using officially distributed packages, simply run the executable to start SpringBoard. In case you're using a game-specific version of SpringBoard, refer to its manual on how to run it.

Note: SpringBoard might show a black screen on the first run until it loads. This is expected behavior, during which the program is caching files, and the program isn't hanging.

Note: Avoid running it in Multiplayer mode (even if it's just a host bot), as SpringBoard might cause extensive network usage, and not work properly.

5.2.1 Layout

Once you start the SpringBoard you should see something similar to *Overview*.

The main editor elements are in the top right part of the UI, separated into tabs. Each of the large buttons with icons represents a specific Editor control.

At the bottom of the screen is the status display, which shows various information about the current state of the Editor, such as memory usage, mouse position, currently selected objects and the undo-redo stack.

At the top left is the project name, and just next to the tabs is the team changer control. In the top middle of the screen is the Start/Stop button which is used to start/stop testing the project.

Zooming in *Objects tab*, we look at the tab elements in more detail.

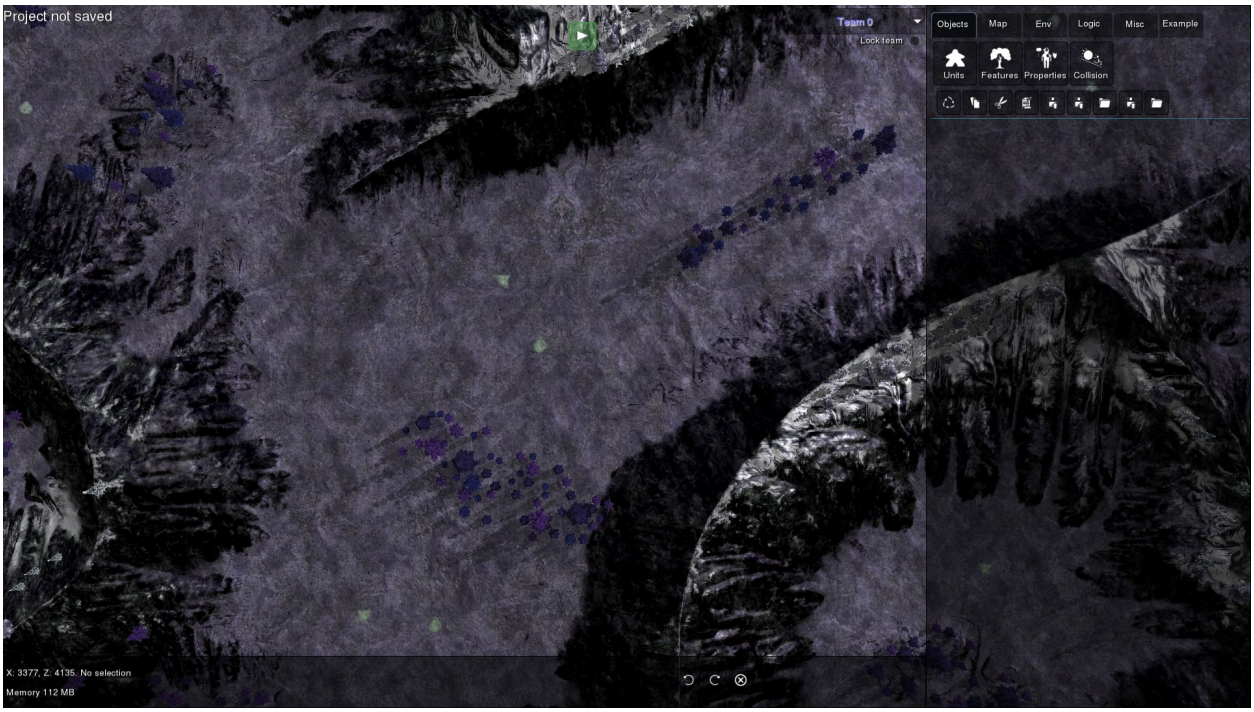


Fig. 1: Overview

Fig. 2: Objects tab

The editor is split into 5 main categories (with an extra one for the example tab). It is possible to navigate between these tabs by either clicking on them or by pressing Shift+Tab to go forward and Shift+Control+Tab to go backwards. For additinoal hotkeys refer to *Hot keys*.

The smaller buttons are shared between all tabs and represent common actions actions such as *Reload meta model, Copy, Cut, Paste, Save project, Save project as, Load project, Export to, Import from* are accessible from all tabs. *Reload meta model* is unique to SpringBoard, and will be described in the *Meta programming* section. Additional common buttons (*Undo, Redo, Clear undo-redo Stack*) are available in the bottom middle of the screen, in the status window.

The tab opened by default is the *Objects tab*. The first two buttons (*Units* and *Features*), are editors for adding and removing unit and feature objects in the game world, with the *Set* and *Brush* editing modes. The *Properties* editor allows editing properties of any selected object group, while *Collision* provides support for editing collision mechanics related properties.

Fig. 3: Map tab

The *Map tab* offers components for editing various Spring maps. Heightmap can be edited with the *Terrain*, via the *Add*, *Set*, and *Smooth* tools. Diffuse, Specular and DNTS maps can be edited using the *Texture* tools, which also support applying arbitrary filters like *blur* and *sharpen* as well as a *Void* editing tool which can be used to make certain map parts invisible.

The *Metal* and *Grass* elements provide support for editing the metal and grass maps respectively. Games that instead use metal spots (or any point-based resource system) should refer to *Extensions used in games*.

Settings allows configuring some map rendering properties, and it also includes an *experimental* map compilation tool (Linux only for now).

All map editing tools support custom *Assets* (both brushes and materials).

The *Env tab* (environment) can be used to set various rendering options. *Lighting*, *Sky* and *Water* can be used to set lighting, atmosphere and water options respectively.

The *Logic tab* provides components to program the scenario, and can be used to add areas, triggers and variables. This is explained in detail in the *Making Scenarios* section.

The *Misc tab* allows editing of players and alliances, as well as setting general project information.

Note: Any changes to the Players component that leads to new teams being added or old ones removed requires a restart of the game.

Lastly, the *Example tab* shows how a custom *extension* can be seamlessly integrated into SpringBoard.

Fig. 4: Env tab

Fig. 5: Logic tab

Fig. 6: Misc tab

Fig. 7: Example tab

5.3 SpringBoard directory structure

SpringBoard directory (`springboard`) is mentioned throughout this documentation, and is located in the Spring data directory.

The `springboard` directory contains user projects, assets and extensions:

1. Projects are maps and scenarios you are working on.
2. Assets are resources used during the project creation process.
3. Extensions are plugins that enhance the SpringBoard editor.

See also:

- *Assets*
- *Extensions*

5.4 Making Scenarios

Scenario programming consists of writing triggers - components that consist of events, conditions and actions. Each trigger can have multiple events - *any* of which can invoke it. Once the trigger has been invoked, *every* condition will be checked, and only if *all* of them are true, *all* actions will be executed.

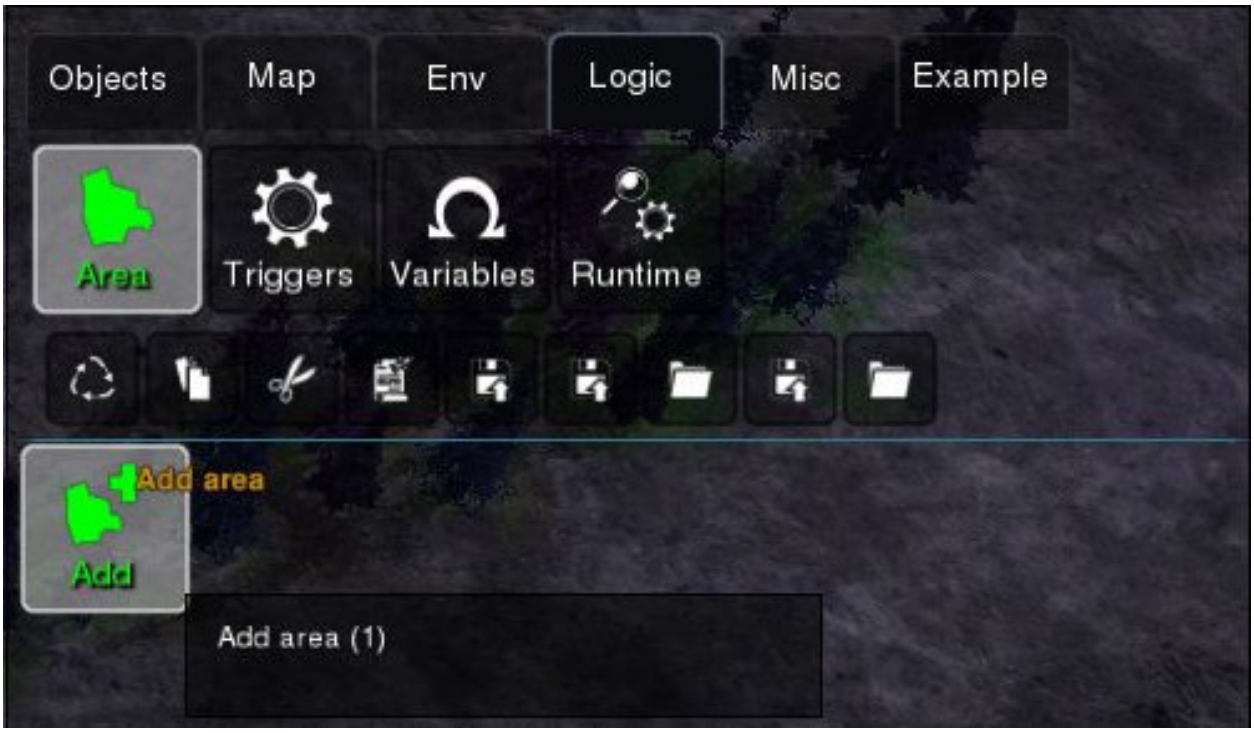
We are going to create an example trigger that demonstrates basic SpringBoard GUI programming elements, based on the Gravitas game.

To begin, we are first going to add a couple of Projector units to the map, using the *Objects/Units/Add* tool.

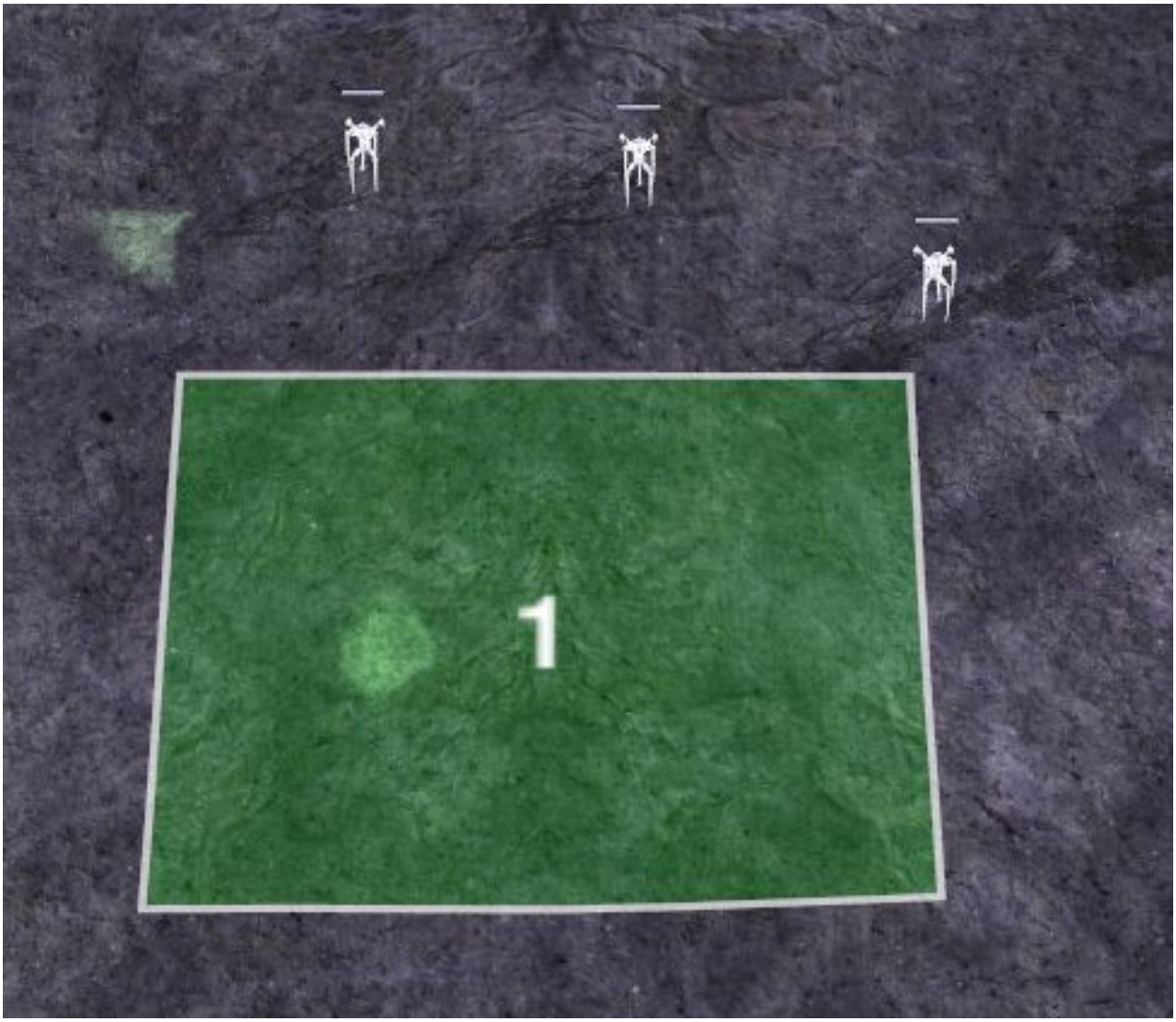

Objects
Map
Env
Logic
Misc
Example
Units
Features
Properties
Collision
Brush
Add
Type:
Units
Terrain:
Ground
Search:
B.O.B
Barrel
Canister
ComeToP...
Crate
Electrafi
Fire
Gate
Gravit
Plate
Projector
Wall1
Wall2
Wall3
Wall4
Projector
defName: projector
Wallier

Add three units like below.

Then we're going to add an Area on the map, using the *Logic/Area/Add* tool.



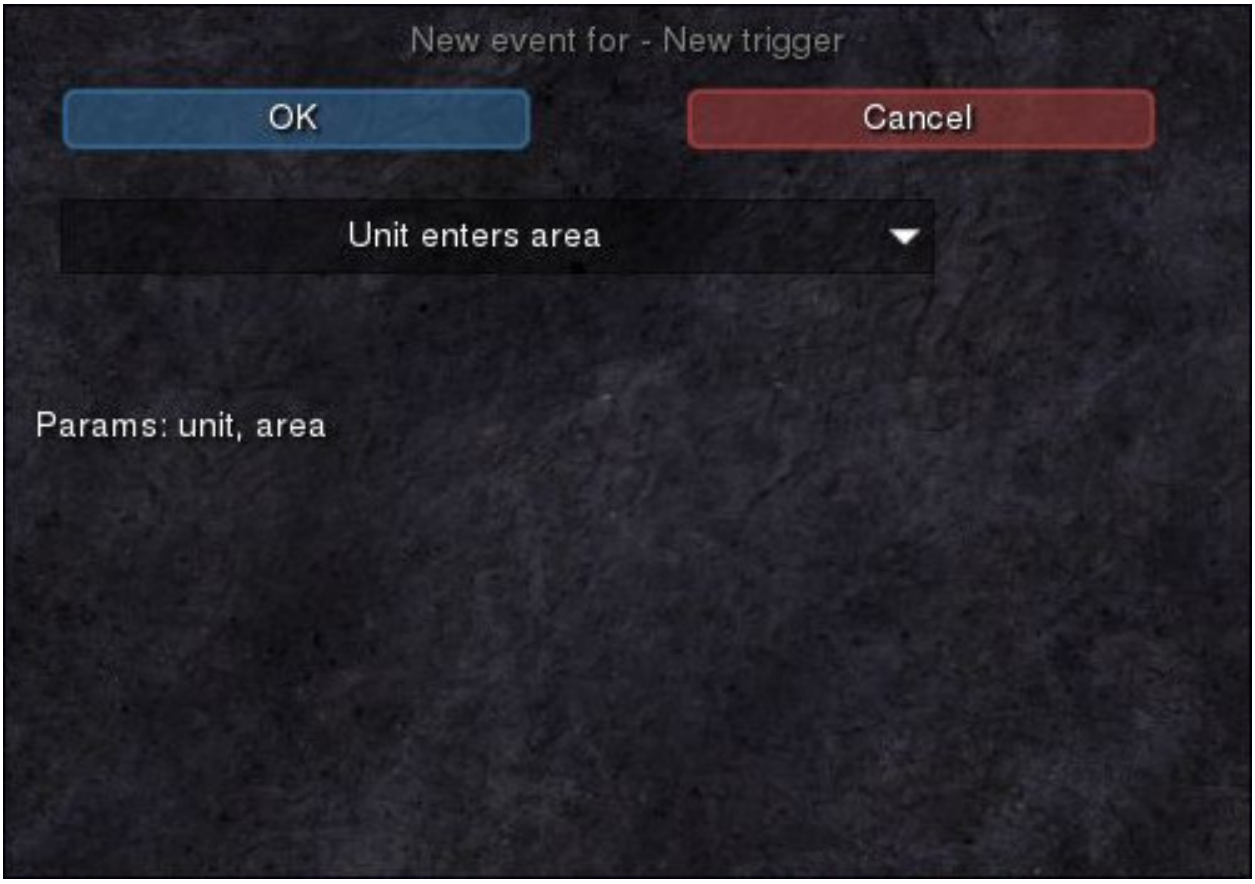
Add an area slightly below of units as demonstrated.



Now we're going to actually start GUI programming, by creating a new trigger trigger.

Pressing the *Add* button should open a new trigger window, which we can rename.

We are then going to add a new *event*: `Unit enters area`. This *event* provides two parameters: `unit` and `area`, which denote the objects that caused the event.



We are then going to confirm if the triggered area is our newly created area, by creating a `Compare area` *condition*.

Lastly, we are going to create a `Destroy unit` action which will destroy the unit that has entered the area.

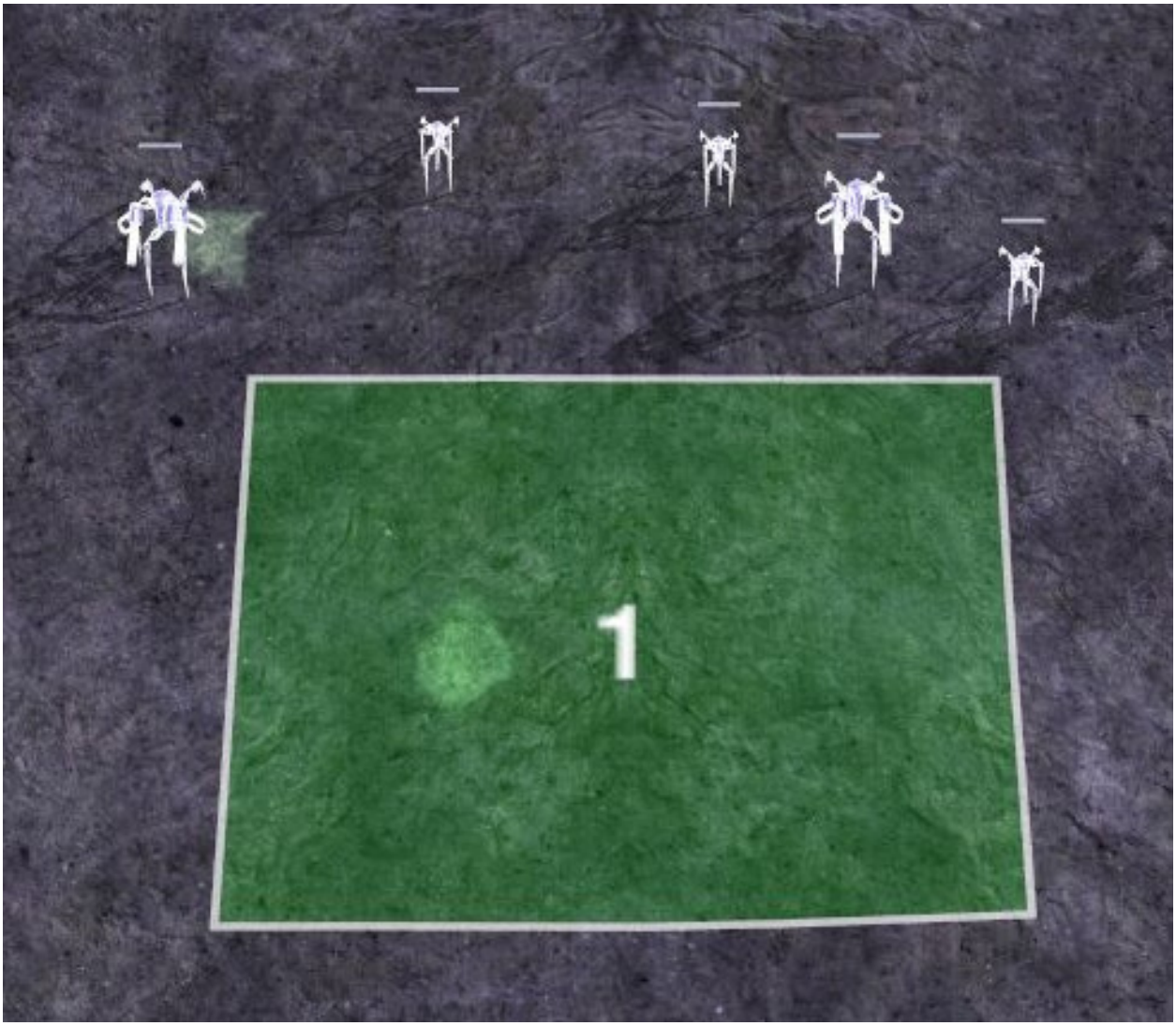
The resulting trigger should be similar to the picture below.

We can now test our trigger. Before starting the game, we should confirm that we have selected `Team 1` as our team,

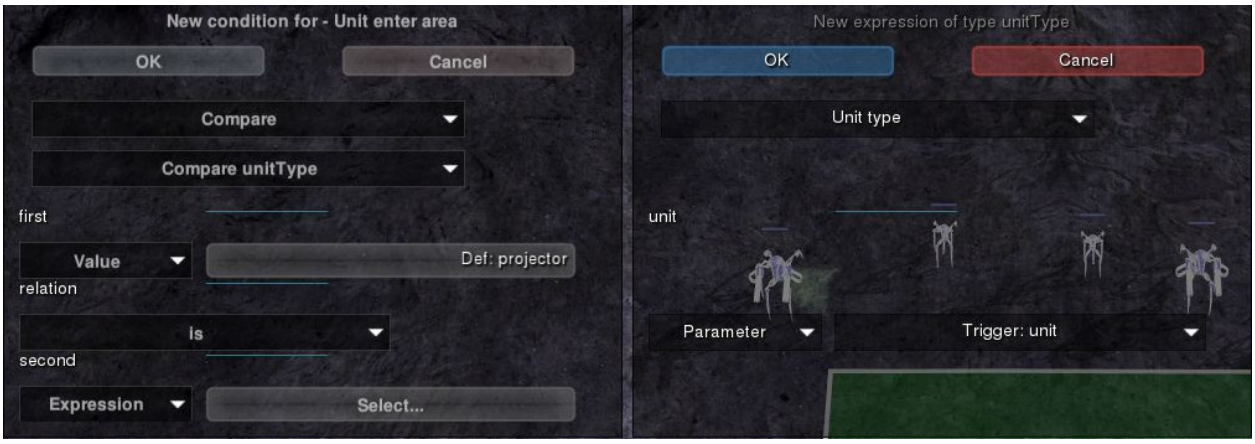
otherwise we won't be able to control the units.

To start testing, we can now press the *Start* button. Moving the units into the area will destroy them as expected.

After we have stopped testing, let's now add two *B.O.B* units using the same *Objects/Units/Add* tool as before.



Now, let's say we don't want *B.O.B* units to be destroyed when they enter the area. We can add another condition that will check the unit type of the entered unit. To do this, we need to use an expression, that will return the unit type of the entered unit.



The resulting trigger is displayed below. If we were to test the scenario again, we will see that *B.O.B* units aren't being

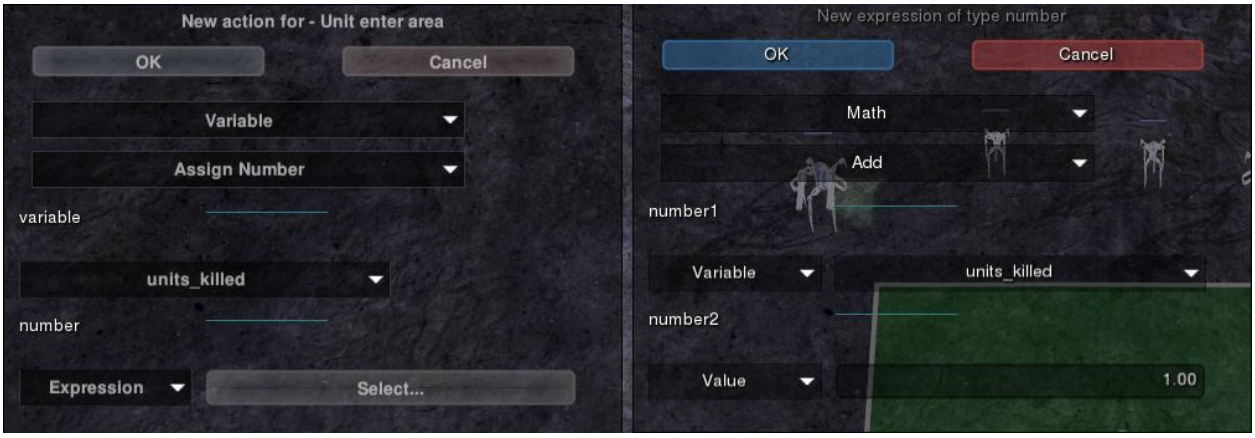
destroyed anymore, which is the desired effect.

Lastly, let's only destroy one unit. To do this, we can count the number of units that have been destroyed, and only destroy the unit if we have destroyed less than `1`. We are going to create a new *variable* `units_killed` which can be used for counting destroyed units.

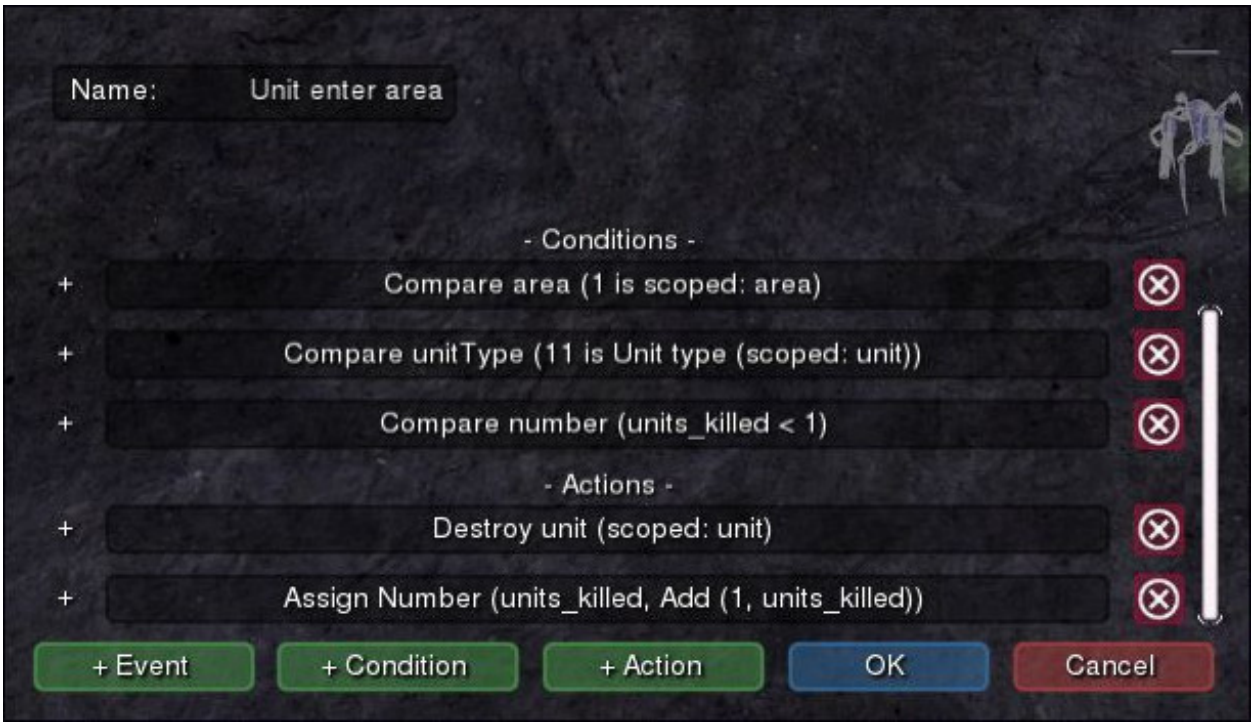
We're going to define it as a `number` variable type and set it to `0`.

Back in our trigger, we're going to create a new condition which compares the `units_killed` variable with `1`.

We also need to increment the variable as a new action. We do this by assigning a new value to the variable, that is the result of adding the old value to `1`.



The resulting trigger is displayed below. Testing it, only the first unit will be destroyed, as desired.



5.5 Meta programming

Trigger functionalities can be extended with Meta programming. It is possible to customize the following:

- *Events*
- *Functions*
- *Actions*
- *Data types*

Meta-model file format:

```
return {
    dataTypes = ..., -- table (or Lua function that returns a table) consisting of␣
↪action types
    events = ..., -- table (or Lua function that returns a table) consisting of␣
↪action types
    actions = ..., -- table (or Lua function that returns a table) consisting of␣
↪action types
    functions = ..., -- table (or Lua function that returns a table) consisting of␣
↪function types
}
```

5.5.1 Events

Events invoke triggers, and are caused by various Spring callins.

They have the following fields:

- **humanName** (mandatory). Human readable name for display in the UI.
- **name** (mandatory). Unique identifier that is used to produce readable models.
- **param** (optional). Additional, event data sources that are available to the entire trigger.
- **tags** (optional). List of human-readable tags used for grouping in the UI.

Example of event programming:

```
{
    humanName = "Unit enters area",
    name = "UNIT_ENTER_AREA",
    param = { "unit", "area" },
}
```

5.5.2 Actions

```
{
    humanName = "Hello world",
    name = "MY_HELLO_WORLD",
    execute = function()
        Spring.Echo("Hello world")
    end,
}
```

The *above* code block defines a simple action. The *name* and *humanName* properties of the action define the machine (unique) and display name respectively. The *execute* property defines the function to be executed when the trigger is successfully fired. When used in the editor, this action would print a *Hello World* on the screen.

It is common for actions to receive *input* that defines its behavior. One such example would be:

```
{
    humanName = "Print unit position",
    name = "PRINT_UNIT_POSITION",
    input = "unit",
    execute = function(input)
        local x, y, z = Spring.GetUnitPosition(input.unit)
        Spring.Echo("Unit position: ", x, y, z)
```

(continues on next page)

(continued from previous page)

```
    end,
}
```

As one might guess, this action would take the specified *unit* as *input* and print out its position. The GUI editor will parse the input type and the user (level designer) will be able to specify the unit when creating an instance of this action. This is equivalent to the following Lua code:

```
function PRINT_UNIT_POSITION(unitID)
    local x, y, z = Spring.GetUnitPosition(unitID)
    Spring.Echo("Unit position: ", x, y, z)
end
```

5.5.3 Functions

The real power of the meta programming comes with the introduction of function types. Function types produce an output (result of the function), which often depends on the input.

Note: There's a difference between a *Lua* function and a function type in the *meta model*. The *function type* represents a component in the meta model and is defined with a table.

Note: Function types should not have a side effect (they shouldn't cause any changes to the game state), but they don't have to be pure (they don't need to produce the same output for the same input).

Example of a function type:

```
{
    humanName = "Unit Health",
    name = "UNIT_HEALTH",
    input = "unit",
    output = "number"
    execute = function(input)
        return Spring.GetUnitHealth(input.unit)
    end,
}
```

This function type takes a *unit* as *input* and produce a *number* as *output*. A special class of these function types are those that return *bool* as *output*, and they represent *conditions* in the GUI programming.

5.5.4 Data types

Custom data types can be created as composites of builtin data types. This allows game developers to expose game-specific concepts. These data types are defined by specifying three fields: *humanName* (display name), *name* (machine name) and *input* (table of fields that it consists of). Example of a *Person* data type:

```
{
    humanName = "Person",
    name = "person",
    input = {
        {
```

(continues on next page)

(continued from previous page)

```
            name = "first_name",
            humanName = "First name",
            type = "string",
        },
        {
            name = "last_name",
            humanName = "Last name",
            type = "string",
        }
    }
}
```

This custom data type can then be used in meta-programming as usual. Below we present a sample action that would print out person's details.

```
{
    humanName = "Print person",
    name = "PRINT_PERSON",
    input = "person" ,
    execute = function(input)
        local person = input.person
        Spring.Echo("Hello! I am " .. person.first_name .. " " .. person.last_name)
    end
}
```

5.5.5 Higher-order functions (Advanced)

As one of the more advanced uses meta-programming also has support for higher-order functions, i.e. fuctions that take other functions as parameters. An example of a *filter* higher-order function implemented in Lua is given below. This function will filter out table elements that don't satisfy a given function. In this case, it will filter out elements that are lower or equal to five. As functions as first-class citizens in Lua, writing them is relatively simple.

```
function above5(x)
    return x > 5
end

function filter(elements, f)
    local retVal = {}
    for _, el in pairs(elements) do
            if f(el) then
                    table.insert(retVal, el)
            end
    end
    return retVal
end

elements = {1, 12, 3, -5, 7}
filter(elements, above5)
```

In SpringBoard's meta-programming however, higher-order functions need to have explicit types, as the meta-programming language is statically (and explicitly) typed. The same filter function type is given below, now in SpringBoard's meta-programming language. The *extraSources* parameter defines additional scoped inputs. The function signature is defined by the *output* parameter. Normally the *input* parameter could also be specified, but that wasn't done in this case, as the predicate function isn't *required* to use the *number* parameter.

```
{
    humanName = "Filter elements in number array",
    name = "number_array_FILTER",
    input = {
        "number_array",
        {
            name = "filter_function",
            type = "function",
            extraSources = {
                "number",
            },
            output = "bool",
        },
    },
    output = "number_array",
    tags = {"Array"},
    execute = function(input)
        local retVal = {}
        for _, element in pairs(input.number_array) do
            if input.filter_function({number = element}) then
                table.insert(retVal, element)
            end
        end
        return retVal
    end,
}
```

Additionally, it is possible to use actions as parameters to higher-order actions types, in the same way like it is done for functions. Below we present a *foreach* action type that will iterate through all elements of an array and execute the specified action for them.

```
{
    humanName = "For each number in number array",
    name = "number_array_FOR_EACH",
    input = {
        "number_array",
        {
            name = "for_each_action",
            type = "action",
            extraSources = {
                "number",
            },
        },
    },
    tags = {"Array"},
    execute = function(input)
        for _, element in pairs(input.number_array) do
            input.for_each_action({number = element})
        end
    end,
}
```

5.5.6 Example

An example of practical meta-programming usage can be seen in the case of Gravitas.

In particular we will focus on two parts of it: the *GATE_OPENED* event type and the *LINK_PLATE_GATE* action type.

The event type is straightfoward, and signals a gate being opened. The unit parameter represents the gate being opened.

```
{
    humanName = "Gate opened",
    name = "GATE_OPENED",
    param = "unit",
}
```

The *LINK_PLATE_GATE* action type takes two unit parameters, one representing a plate, and other representing a gate. It then uses game API to link the two together, causing the gate to open if the pressure plate is activated.

```
{
    humanName = "Link Plate To Gate",
    name = "LINK_PLATE_GATE",
    input = {
        {
            name = "plate",
            type = "unit",
        },
        {
            name = "gate",
            type = "unit",
        },
    },
    execute = function(input)
        GG.Plate.SimpleLink(input.plate, input.gate)
    end
}
```

Example scenario implemented using this custom meta-programming is available at Gravitas Example. To use it, extract it and open it as a SpringBoard project.

5.6 Map Features

5.6.1 Importing

SpringBoard doesn't come with any map features besides the engine-default geovent, so if you want to add them you first need to import them into your project.

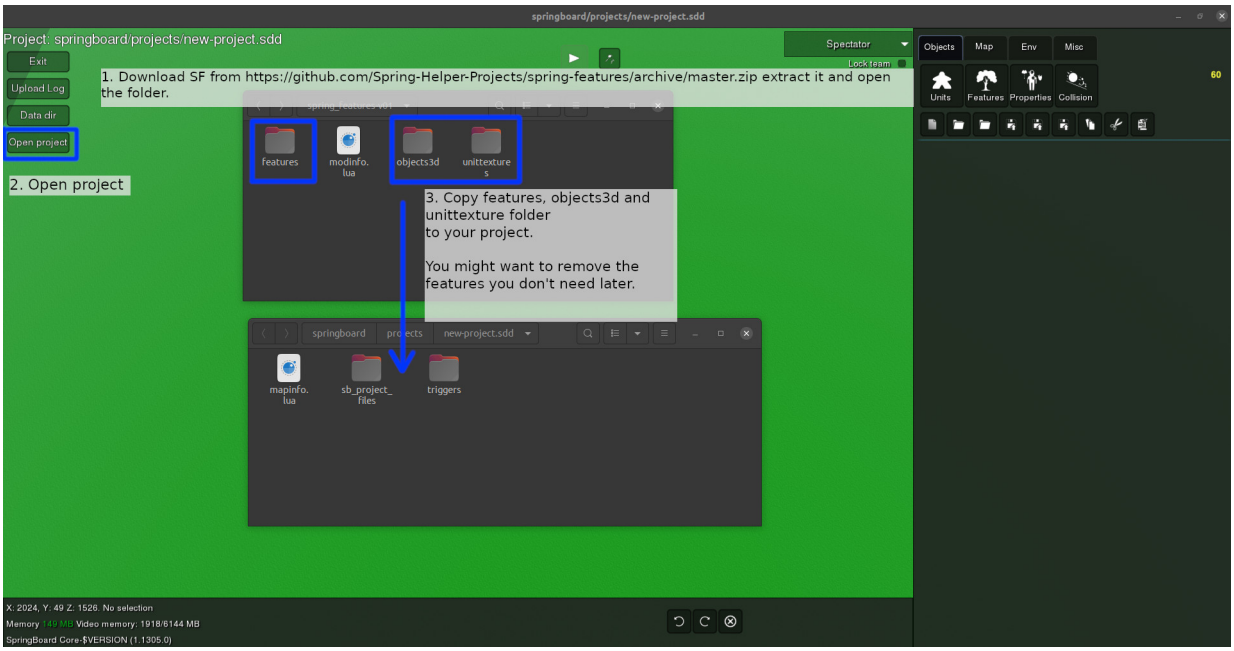
To import new features you need to perform the following steps:

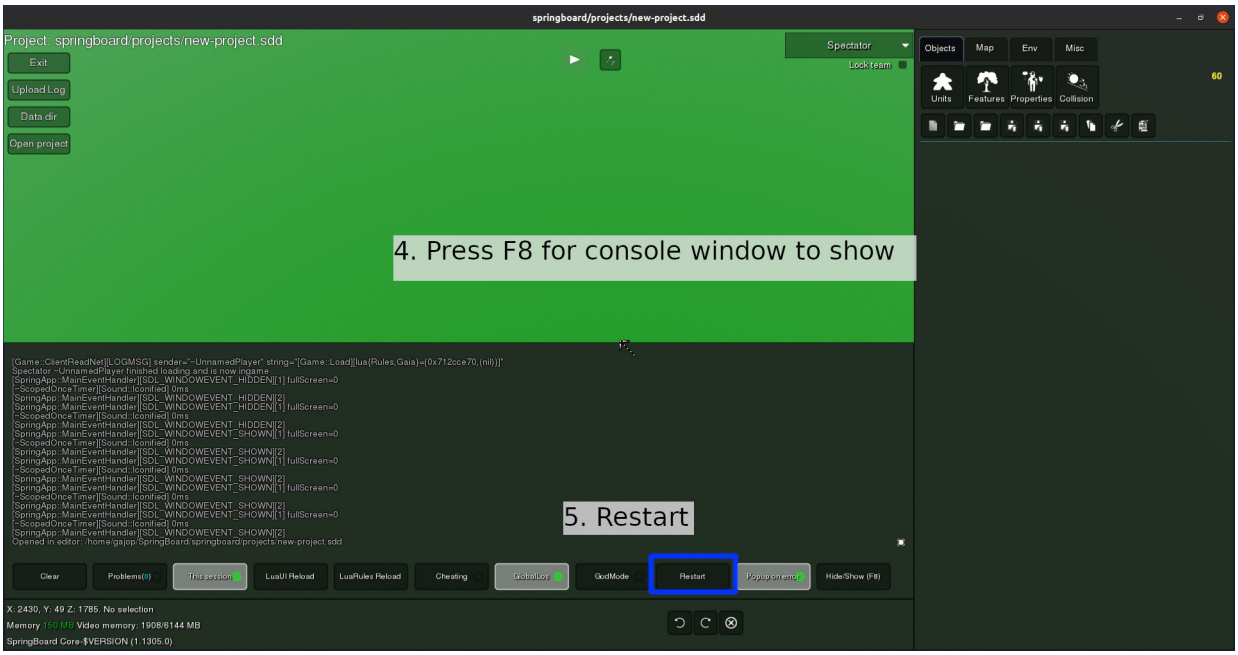
1. Find existing feature sets (e.g. SpringFeatures) or make your own following the official Spring docs.
2. Copy features into your project and restart.

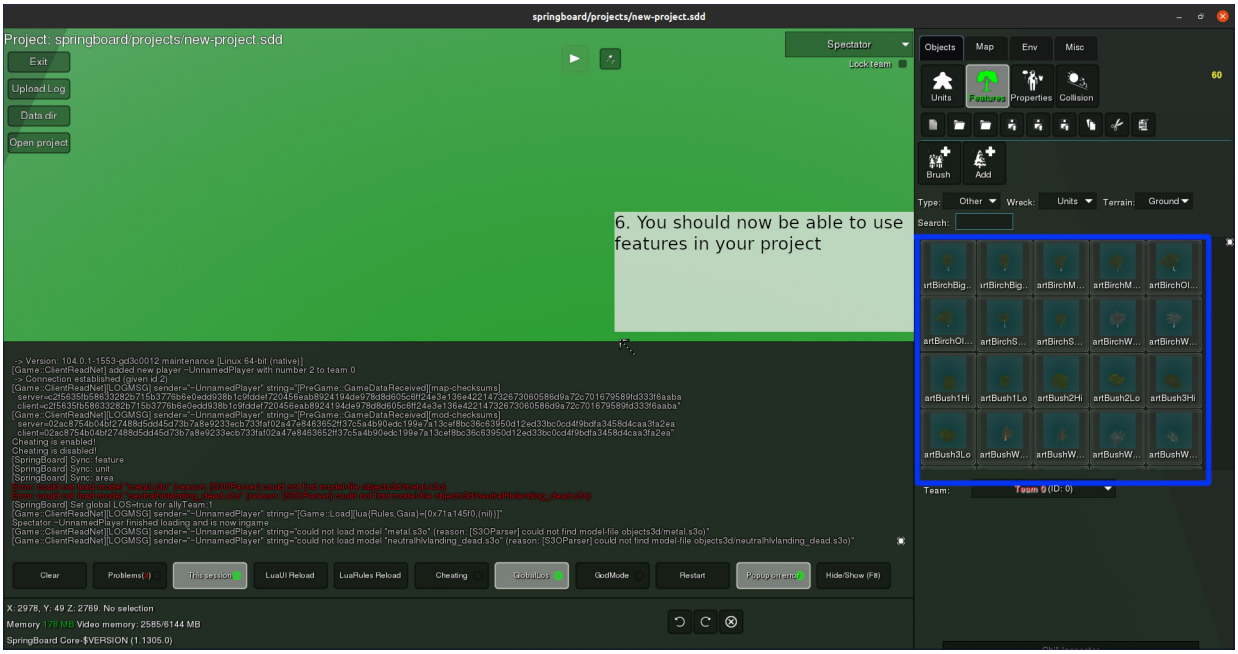
Below is an example of how to import `SpringFeatures` into a SpringBoard project.

1. Download SF from https://github.com/Spring-Helper-Projects/spring-features/archive/master.zip extract it and open the folder.
2. Open an existing project or create a new one. Then, open its folder using the "Open Project" button.
3. Copy features, objects3d and unittexture folder to your project. You might want to remove the features you don't need later.
4. Press F8 for console window to show

5. Restart
6. You should now be able to use features in your project







5.6.2 Exporting as Spring archive (recommended)

Exporting your project using the "Spring archive" export option will include all features and is the recommended way to export a SpringBoard project.

5.6.3 Export manually

Manual export is only recommended if you want to customize the export process (perhaps you want to compile the map yourself using a third-party tool). In that case you can follow the below tutorial.

s11n export should be used if you want to export game objects (units, features, etc.) and load them in your standalone map. Install s11n as you would normally:

1. Copy the s11n and LCS folders to the `libs/s11n` and `libs/LCS` folders of the map (create destination directories as necessary).
2. Copy s11n_gadget_load.lua from the `s11n` folder to `LuaGaia/Gadgets/` of the map folder.

Then setup s11n to load your exported objects:

1. Copy your exported s11n model file to map's `mapconfig` folder.
2. Copy s11n_load_map_features.lua to map's `LuaGaia/Gadgets/` folder.
3. Set the file path to your s11n model file in the newly copied `s11n_load_map_features.lua`

Including feature defs, compiling the map and setting mapinfo.lua is outside the scope of this guide. Please consult the Spring MapDev pages for that.

5.7 Assets

Assets can be added to SpringBoard in order to include new art for editing. They should be added in the `springboard/assets` folder, and each asset pack should have its own directory structure.

SpringBoard supports the following asset types:

- **Brush patterns** */brush_patterns*. These should be black images with an alpha determining the value.
- **Materials** */materials*. Materials consist of multiple different textures (`diffuse`, `specular` and `normal`), which should be contained in different image files, in the `name_$texType.png` format, e.g. `cement_diffuse.png` and `cement_specular.png`.
- **Skyboxes** */skyboxes*. These should be `.dds` skybox textures.
- **Detail textures** */detail*. These are the usual Spring detail textures.

A set of core assets are available. You can download them either via the launcher's `Asset Download` option (recommended), or manually via a direct link. In case of a manual download, you need to extract them to `springboard/assets/core/`.

5.8 Extensions

SpringBoard support editor extensions. They should be created in separate folders, in the `springboard/exts` folder, and they consist of two subfolders:

- **ui**. This is where you should place all strictly unsynced extensions, like the Editor GUI and States.
- **cmd**. This folder should contain files that should be shared by both synced and unsynced extensions, like Command and Model.

5.8.1 Example

We present a full example of a SpringBoard extension consisting of *ui* and *cmd* modules. This example is located in the exts/example folder of the repository.

First we will define the UI elements, given in the ui/example.lua file. At the top of the file, we will include the *Editor* class and make a new *ExampleEditor* subclass out of it, with which we will define our custom *Editor*.

```
SB.Include(Path.Join(SB.DIRS.SRC, 'view/editor.lua'))

ExampleEditor = Editor:extends{}
```

We will then register the newly defined class to make it accessible in the SpringBoard interface.

```
ExampleEditor:Register({
    name = "exampleEditor",
    tab = "Example",
    caption = "Example",
    tooltip = "Example editor",
    image = Path.Join(SB.DIRS.IMG, 'globe.png'),
})
```

Then in the init method, we will define the fields. We create two *NumericFields*: *example* and *undoable*, and we add them to the *Editor*.

```
function ExampleEditor:init()
    self:super("init")

    self:AddField(NumericField({
        name = "example",
```

(continues on next page)

(continued from previous page)

```
        title = "Example:",
        tooltip = "Example value tooltip.",
        width = 140,
        minValue = -10,
        maxValue = 5,
    }))

    -- Note: as we are setting the value in synced only, we won't see the effect of␣
↪undo in the editor.
    -- Consider using game rules if you want to be able to read in the UI as well.
    self:AddField(NumericField({
        name = "undoable",
        title = "Undoable:",
        tooltip = "This value can be used with undo/redo.",
        width = 140,
        minValue = -3,
        maxValue = 12,
    }))

    local children = {
        ScrollPanel:New {
            x = 0,
            y = 0,
            bottom = 30,
            right = 0,
            borderColor = {0,0,0,0},
            horizontalScrollbar = false,
            children = { self.stackPanel },
        },
    }

    self:Finalize(children)
end
```

To handle field changes, we will create an *OnFieldChange* method, and when fields change, we will create and execute appropriate *Commands*.

```
function ExampleEditor:OnFieldChange(name, value)
    if name == "example" then
        local cmd = HelloWorldCommand(value)
        SB.commandManager:execute(cmd)
    elseif name == "undoable" then
        local cmd = UndoableExampleCommand(value)
        SB.commandManager:execute(cmd)
    end
end
```

We also want to group all changes for the *UndoableExampleCommand* into a single undo/redo command on the command stack, and for that purpose we use the *SetMultipleCommandModeCommand* command.

```
function ExampleEditor:OnStartChange(name)
    if name == "undoable" then
        SB.commandManager:execute(SetMultipleCommandModeCommand(true))
    end
end
```

(continues on next page)

(continued from previous page)

```
function ExampleEditor:OnEndChange(name)
    if name == "undoable" then
        SB.commandManager:execute(SetMultipleCommandModeCommand(false))
    end
end
```

We also need to define the two commands. This is done in separate files, in the cmd folder, which makes the Commands accessible from both unsynced (GUI) and synced (execution). The *HelloWorldCommand* is rather simple, and it just prints out a single line of text.

```
HelloWorldCommand = Command:extends{}
HelloWorldCommand.className = "HelloWorldCommand"

function HelloWorldCommand:init(number)
    self.number = number
end

function HelloWorldCommand:execute()
    Spring.Echo("Hello world: " .. tostring(self.number))
end
```

The *UndoableExampleCommand* is slightly more complicated as it also has a value that can be changed. In the *:unexecute()* method we revert it to its previous value.

```
UndoableExampleCommand = Command:extends{}
UndoableExampleCommand.className = "UndoableExampleCommand"

local value = 0
function UndoableExampleCommand:init(number)
    self.number = number
end

function UndoableExampleCommand:execute()
    Spring.Echo("Setting value: " .. tostring(self.number))
    self.old = value
    value = self.number
end

function UndoableExampleCommand:unexecute()
    Spring.Echo("Reverting to: " .. tostring(self.old))
    value = self.old
end
```

Note: Displaying a synchronized value in the GUI requires additional steps. Depending on how this value is kept, things like RulesParams can be used. Refer to the Spring documentation for details: https://springrts.com/wiki/Lua_SyncedCtrl#RulesParams https://springrts.com/wiki/Lua_SyncedRead#RulesParams

5.8.2 Extensions used in games

Zero-K's metal spot extension.

This extension describes how the ObjectBridge API can be used to create new, custom editors for game world objects.

5.9 Game-specific modules

Game-specific modules can be created to customize the editor for a specific game. It can include *extensions* and *meta-programming*, as well as settings and widget/gadget overrides.

This is achieved by creating mutators which depend on SpringBoard-Core and the actual game. The modinfo.lua mutator for Balanced Annihilation is given below:

```
return {
  name = 'SpringBoard BA',
  shortname = 'SB_BA',
  game = 'SpringBoard BA',
  shortGame = 'SB_BA',
  description = 'SpringBoard for Balanced Annihilation',
  version = '$VERSION',
  mutator = 'Official',
  modtype = 1,
  depend = {
    -- Order matters. Putting game second ensures its widget/gadget handler is loaded

    'rapid://sbc:test',
    --'SpringBoard Core $VERSION',

    'rapid://ba:test',
    --'Balanced Annihilation $VERSION',
  },
}
```

It is possible to use both the rapid dependencies (such as `rapid://sbc:test`, `rapid://ba:test`) as well as for local versions (`SpringBoard Core $VERSION`, `Balanced Annihilation $VERSION`), which are more useful for development.

Note: It's important to first include SpringBoard Core and then the game.

5.9.1 Configuration

General configuration is given in the `sb_settings.lua` file. It allows to configure the following behavior:

- `startStop`: Position of the startStop button.
- `OnStartEditingSynced`: Function to be executed in synced when editing starts.
- `OnStopEditingSynced`: Function to be executed in synced when editing stops.
- `OnStartEditingUnsynced`: Function to be executed in unsynced when editing starts. This is commonly used to disable widgets that would get in the way of editing.
- `OnStopEditingUnsynced`: Function to be executed in unsynced when editing stops. This is commonly used to reenable widgets that were disabled for editing.

Balanced Annihilation version of the file is given in sb_settings.lua

5.9.2 Editor mode handling

Sometimes it's necessary to have the game's widget or gadget act differently depending on whether it's currently being edited or not. It is possible to read the current state from the GameRules' `sb_gameMode` parameter, like: `Spring.GetGameRulesParam("sb_gameMode")`. It can return three values:

- `"dev"`: SpringBoard is currently in edit mode and game mechanics (especially those depending on time) shouldn't be happening.
- `"test"`: SpringBoard is currently being tested and game mechanics should work as normally. Additionally debug information can be printed out and it should be possible to return to the Editor (e.g. by clicking on the *Stop* button).
- `"play"`: SpringBoard's is currently being played, like a normal scenario. All game mechanics should work as usual, with no debug/development information printed. It is not possible to return back to the Editor.

Example of how this can be handled is given for a SpringBoard EVO gadget.

5.9.3 Examples

Repositories:

- https://github.com/Spring-SpringBoard/SpringBoard-BA
- https://github.com/Spring-SpringBoard/SpringBoard-ZK
- https://github.com/Spring-SpringBoard/SpringBoard-EVO
- https://github.com/Spring-SpringBoard/SpringBoard-S44

5.10 Video tutorials

Basic introductory tutorials:

- Introduction
- Scenario editing
- Meta-programming

5.11 Hot keys

General

Hotkey	Action
Ctrl-Z	Undo
Ctrl-Y	Redo
Ctrl-A	Select All
Ctrl-T	Select objects of same type as current selection
Ctrl-Shift-T	Select objects in view, of same type as current selection
Ctrl-X	Cut
Ctrl-C	Copy
Ctrl-V	Paste
Del	Delete current selection objects
Ctrl-S	Save
Ctrl-Shift-S	Save as
Ctrl-O	Open
Ctrl-I	Import
Ctrl-E	Export
1-9	Enter into a current editor's edit mode
Esc	Close dialog
Enter	Confirm dialog

Map editing

5.12 Feature comparison

Here we present a comparison between SpringBoard and other scenario editing software.

5.12.1 Spring tools

The most popular alternative for a scenario editor in the SpringRTS ecosystem is the Zero-K Mission Editor (ZKME) (link). It is primarily designed for Zero-K, but it has basic support for other games with similar mechanics.

Comparing SpringBoard to ZKME:

- ZKME runs as an external tool, while SpringBoard runs in-engine. This allows SpringBoard to offer WYSIWYG kind of editing, and also use camera controls that players are already familiar with.
- ZKME doesn't support expressions in functions and actions. This makes it hard to write complex conditions and actions, which limits expressibility.
- ZKME doesn't support variables which makes writing logic that depends on state difficult.
- ZKME only runs on Windows, while SpringBoard supports all platforms that work with the SpringRTS engine.
- ZKME isn't extensible, while SpringBoard has meta-programming which can be used to expose custom functionality to the GUI.

5.12.2 Other tools

Starcraft 2 and Warcraft 3 scenario editors are editors for the popular Starcraft 2 and Warcraft 3 games made by Blizzard entertainment.

Comparing them to SpringBoard:

- They are made for one game specifically, while SpringBoard can be used for any game using the SpringRTs engine.
- They are released under a proprietary license, and need to be paid to be used. SpringBoard is free and open source.
- They offer extensibility in terms of coding, while SpringBoard supports it with meta-programming. The SpringBoard approach allows extensions to be created by more experienced developers and they are seamlessly integrated within the editor, which makes them usable by novices.

SpringBoard, ZKME and Starcraft 2/Warcraft 3 editors all use a event/condition/action system, and that seems to be the most popular approach for defining scenarios.

5.13 API

Full API can be found at API.

5.13.1 View

The API describing the View elements is described in the various Field pages and and the Editor class.

5.13.2 State

The State API consists of an Abstract State class which should be subclasses when implementing new behaviors, and the State Manager class that can be invoked to read and modify current state.

5.13.3 Command

The Command API is split into three parts, the base Command interface, the CompoundCommand class for grouping multiple commands into a single one and the the CommandManager class for invoking execution of commands.

5.13.4 Model

The model API is given by the ObjectBridge interface. More detailed s11n serialization API is also included.