CardEngine

About

The CardEngine framework is an add-on to the Godot game engine which provides basic functionality to develop card based games. Like Godot it comes under the MIT license, and the source code is available on GitHub. The main goal of this project is to provide a good amount of flexibility while not requiring a lot of code to be written. With it you can create games inspired by Hearthstone, or Slay the Spire.

NOTE: the framework is still in very early stage, expect things to not function out of the box and everything is subject to change.

Features

- **Integrated UI**: CardEngine's UI is integrated inside the Godot Engine Editor.
- **Flexible card database design**: CardEngine stores card inside databases using a flexible but simple structure.
- **Straightforward Database query**: CardEngine allows to write database query in a straightforward manner.
- **Advanced in-memory card management** : CardEngine supports in-memory card filtering and sorting.
- **Powerful layout system**: CardEngine provides a code-less container widget creation tool.
- **Advanced animation tools**: CardEngine offers an easy way to create complex card animations.
- **Many quality-of-life tools**: CardEngine comes with other tools to help creating card-based game.

Databases

Overview

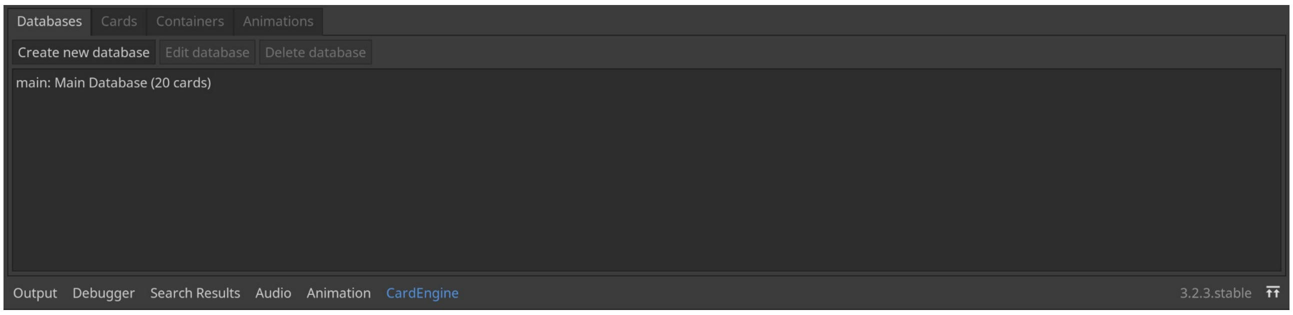
CardDatabase class: `res://addons/cardengine/database/database.gd` (source)
DatabaseManager class:
`res://addons/cardengine/database/database_manager.gd` (source)

CardEngine stores card's data inside databases. You can create one or more databases depending on your need. Usually one should be sufficient and easier to manage.

CardEngine database are file based and can be found inside the `_private/databases` folder in the format `<database ID>.data`. Such files should not be edited by hand.

Databases are identified by an unique ID which is defined during creation and cannot be changed. Databases also have a name for display purpose, also defined during creation, it can be changed.

Creation



main: Main Database (20 cards)

To create a database press the "Create new database" button. Fill in the ID and the name. The ID cannot contain spaces, can only contain a-z A-Z 0-9 characters and cannot start with a number. The name can be any non-empty string of characters.

Deletion

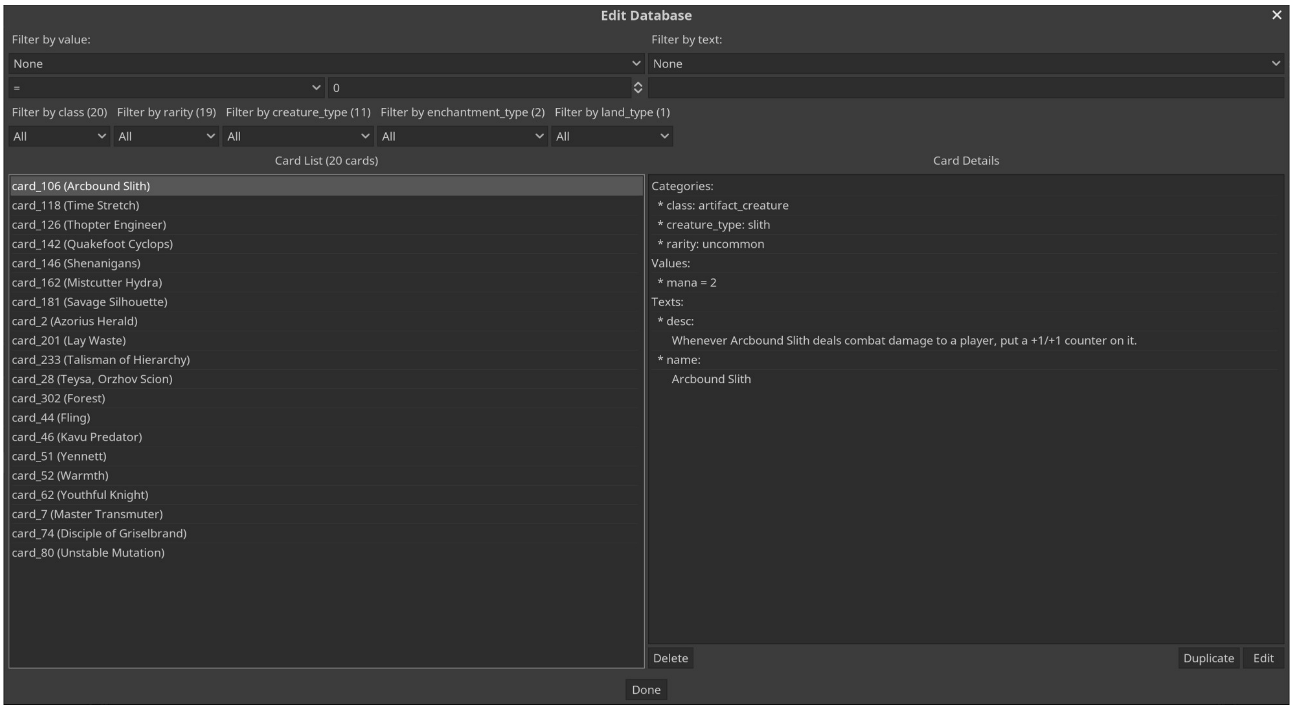
To delete a database, select it in the list and press the "Delete database" button. You will have to validate the delete with the confirm dialog. Be careful, deleting a database is not reversible.

Changing ID and name

The ID cannot be changed.

The name can be changed by double-clicking the database in the list.

Editing the database



card_106 (Arcbound Slith)
card_118 (Time Stretch)
card_126 (Thopter Engineer)
card_142 (Quakefoot Cyclops)
card_146 (Shenanigans)
card_162 (Mistcutter Hydra)
card_181 (Savage Silhouette)
card_2 (Azorius Herald)
card_201 (Lay Waste)
card_233 (Talisman of Hierarchy)
card_28 (Teysa, Orzhov Scion)
card_302 (Forest)
card_44 (Fling)
card_46 (Kavu Predator)
card_51 (Yennett)
card_52 (Warmth)
card_62 (Youthful Knight)
card_7 (Master Transmuter)
card_74 (Disciple of Griselbrand)
card_80 (Unstable Mutation)

Categories:
 * class: artifact_creature
 * creature_type: slith
 * rarity: uncommon
Values:
 * mana = 2
Texts:
 * desc:
 Whenever Arcbound Slith deals combat damage to a player, put a +1/+1 counter on it.
 * name:
 Arcbound Slith

The dialog

At the top of the dialog box you find the filters which allow you to find the cards you are looking for. Filters are generated automatically to reflect the content of the database.

On the left you find the list of cards in the database. The cards are identified by their ID, the only mandatory data. If cards have a "name" text it is displayed along the ID between parenthesis. Card in the list can be selected by single click.

On the right you find the card details pane. When selecting a card, it displays the data attached to the card. No interaction on the data is possible here. Instead you find two buttons at the bottom to delete or edit a card.

At the bottom you find the "Done" button to close the dialog.

Creating a card

You don't create cards from this dialog, refer to the "Cards" chapter for more details.

Editing a card

To edit a card, select the card in the list on the left then press the "Edit" button a the bottom of the right pane. You will be transferred to the "Cards" page where the actual edit will take place, refer to the "Cards" chapter for more details.

Deleting a card

To delete a card, select the card in the list on the left then press the "Delete" button a the bottom of the right pane. A confirmation will be asked, be careful this is not a recoverable action.

Duplicating a card

To duplicate a card, select the card in the list on the left then press the "Duplicate" button at the bottom of the right pane. As there cannot be cards with the same ID, you will presented with a dialog to enter an ID for the duplicate. Once a proper ID entered you can press "Confirm" at the bottom of the dialog, you should see your duplicated card added to the list.

Cards

Overview

For convenience reason a card is split into multiple layers. The data layer which is what the databases manipulate. The visual layer which is a Godot Node2D to integrate into the scene system. The instance layer which helps with managing the card in-memory.

Data layer (CardData class)

`res://addons/cardengine/card/card.gd` (source)

The structure

The card data structure is designed to be simple and yet flexible. There is 3 types of data, categories, values and texts. It does not seem much but I think this enough to cover most of the use case.

Properties

Cards have the following properties:

- `id: String` is the card's unique identifier inside the database
- `source_db: String` is the database's unique identifier from where the card comes from

Categories

Categories help you define what is your card, is it an attack card? Or is it a defense card? Is it magical? Or is it a weapon? A category is composed of two parts, a meta-category and the category itself. The meta-category gives you the ability to type your categories to help create better classification. An example of meta-category would be "rarity". The category is the value given to a meta-category for a card. An example of categories, for the meta-category "rarity", would be "common", "uncommon", "rare" and "epic".

You can access the categories by using the function `CardData.get_category(meta_category)`. This function return the category as a string set for the given meta category, if the meta category does not exists for the card, it returns an empty string.

Values

Values are a way to give numerical values to a card. Numerical value are usually at the center of the gameplay of many card games. As computer only understand numbers, the game logic manipulate them to determine what a card does. A value is composed of an ID and a numerical value. An example of value would be "mana", to define the mana cost of the card.

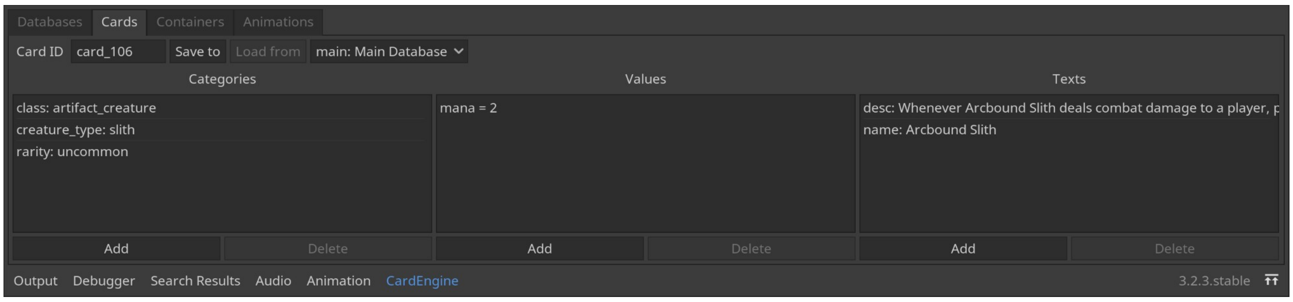
You can access the values by using the function `CardData.get_value(id)`. This function return the value as an integer for the given ID. To avoid error message if you are not sure what values your card has, check first with the function `CardData.has_value(id)`.

Texts

Texts are a way to add textual information to a card. Text is not so central for the internal working of a card game but are very important for the player to understand what a card does. They are mostly a visual support for the two data type above. A text is composed of an ID and a textual value. And example of text would be "name", to define the display name of the card

You can access the texts by using the function `CardData.get_text(id)`. This function return the text as a string for the given ID. To avoid error message if you are not sure what texts your card has, check first with the function `CardData.has_text(id)`.

Editing



Card ID: card_106
main: Main Database

Categories:
class: artifact_creature
creature_type: slith
rarity: uncommon

Values:
mana = 2

Texts:
desc: Whenever Arcbound Slith deals combat damage to a player, p
name: Arcbound Slith

To edit cards data, CardEngine provides a UI (see above). At the top you find the toolbox. The toolbox start with a field where you fill in the ID of the card. The ID is unique and the only required data needed to create and manipulate a card. Next the "Save to" button allows you to write the card in the selected database. If the card has been correctly written in the database, a success message is displayed. Following is the "Load from" button which allows you, when the ID is filled in, to read the card from the selected database. If the card cannot be read from the database, an error message will be displayed. Finally the combo box allows you to select the database on which you want to work on.

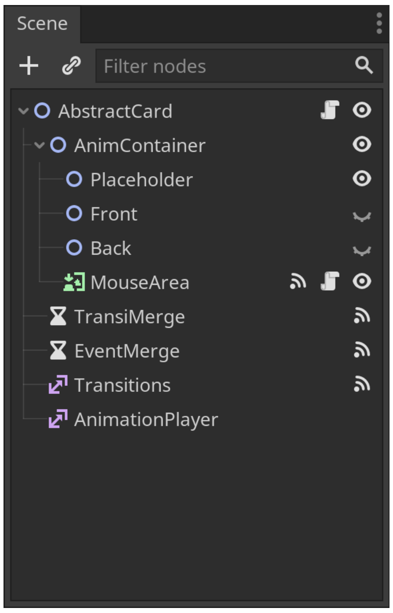
In the main area, there is 3 columns, each one correspond to a data type seen above. Each column behave the same way. At the top the list display the data already attached to the card, it is empty at first. Single click to select an item. Double-click to edit an item. At the bottom, the "Add" button allows you to create a new item, and the "Delete" button allows you remove the selected item.

Note: no modification to the card is save to the database until you press the "Save to" button.

Visual layer (AbstractCard scene)

`res://addons/cardengine/card/abstract_card.tscn` (source)

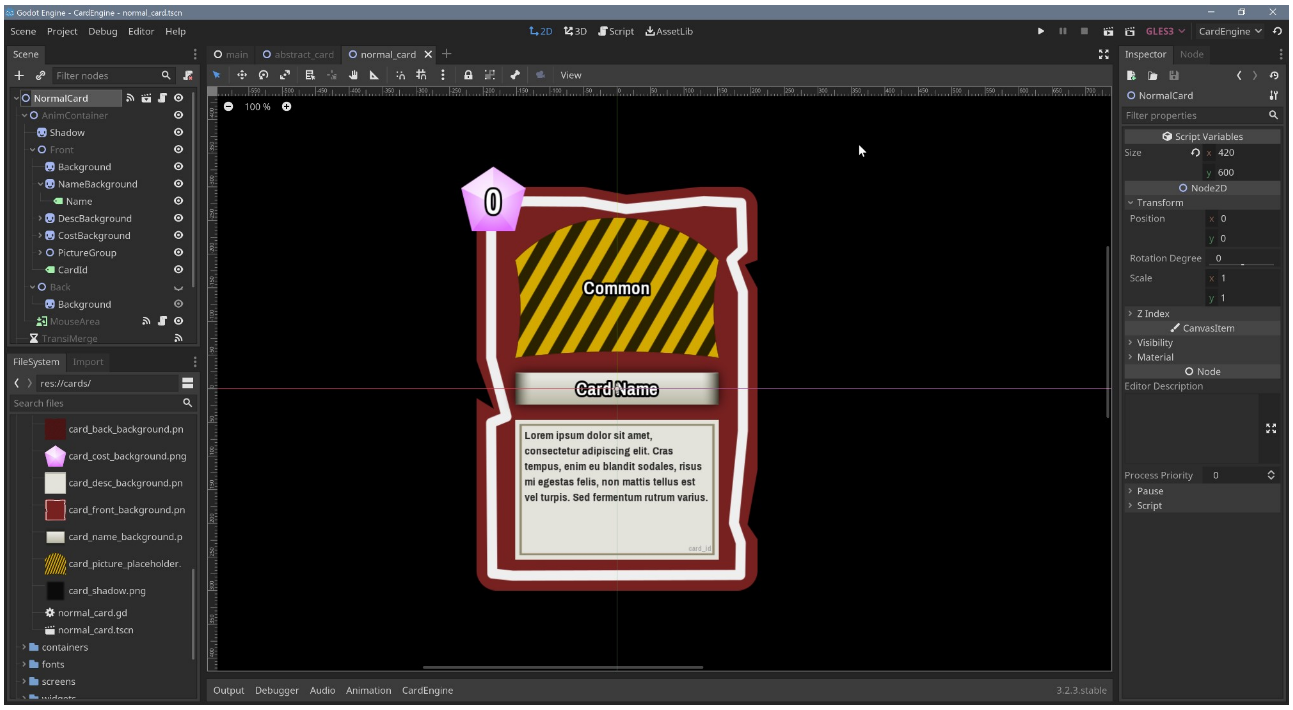
Introduction



The visual layer describes how the card data are displayed. To integrate best with the Godot scene system, a card's root node is a Node2D and the structure can be describe as follow:

- **AnimContainer** is a Node2D to which the animations are applied. The reason to have this node is to make animations independent of how the card is transformed by its container.
    - **Placeholder** is a Node2D which is the root of what is displayed when the visual layer is not linked to the data layer (more details below)
    - **Front** is a Node2D which is the root of what is displayed on the front of the card
    - **Back** is a Node2D which is the root of what is displayed on the back of the card
    - **MouseArea** is a Control which capture all the mouse interaction for the card
- **TransiMerge** is a Timer which keeps only the latest modification to the card made by its container in a given time
- **EventMerge** is a Timer which keeps only the relevant events resulting from player interaction in a given time
- **Transitions** is a Tween which animate transformation of the card by its container
- **AnimPlayer** is a Tween which play the animations defined by its container

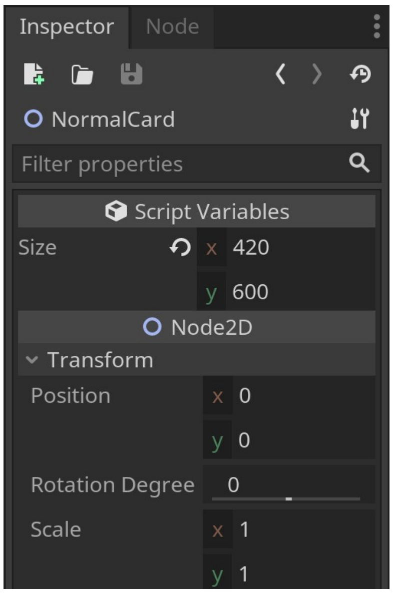
Creation



Here is the process to create your card scene:

1. Open the "Scene" menu at the top left of the Godot Editor and choose "New inherited scene"
2. Look for `abstract_card.tscn` in the `res://addons/cardengine/card` folder and click the "Open" button
3. Rename the root node to whatever you deem appropriate, like "NormalCard" and save the scene. You can save the scene where you want but for good practice save it as `res://cards/<name>/<name>_card.tscn` (ex: `res://cards/normal/normal_card.tscn`)
4. To be able to customize the behavior of your card, you need to have a script. To do so, right-click the root node and choose "Extend script" and save.

Once the scene is created you can start the customization process. You can place any Node2D or Control derivative nodes under the "Front" and "Back" node to create your visual. Common elements to the front and back can be placed directly under "AnimContainer".



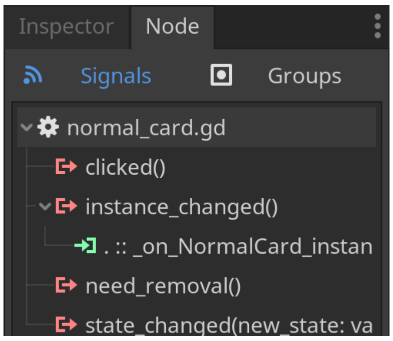
Size x 420 y 600

Position x 0 y 0

Rotation Degree 0

Scale x 1 y 1

An important step is to set the "Size" property to define the horizontal and vertical size in pixels of your card. It is important because containers need this information to calculate their layout.



normal_card.gd
clicked()
instance_changed()
. :: _on_NormalCard_instan
need_removal()
state_changed(new_state: va

Once your visual is ready there is a good chance that you want to display your data using it. To do so, connect the signal `instance_changed` to your card script. You can then access the data with a call to `instance().data()`. Here an example from the demo:

```
func _on_NormalCard_instance_changed() -> void:
    var data := instance().data()

    _card_id.text = data.id

    if data.has_text("name"):
        _name.text = data.get_text("name")

    if data.has_text("desc"):
        _desc.text = data.get_text("desc")

    if data.has_value("mana"):
        var val = data.get_value("mana")
        if val >= 0:
            _cost.text = "%d" % val
        else:
            _cost.text = "X"

    for child in _picture_group.get_children():
        child.visible = false

    if data.has_meta_category("rarity"):
        if data.get_category("rarity") == "common":
            _common.visible = true
        elif data.get_category("rarity") == "uncommon":
            _uncommon.visible = true
        elif data.get_category("rarity") == "rare":
            _rare.visible = true
        elif data.get_category("rarity") == "mythic_rare":
            _mythic_rare.visible = true
    elif data.has_meta_category("class"):
        if data.get_category("class") == "basic_land":
            _basic_land.visible = true
```

Note on Placeholder

The visual layer comes with a placeholder which is meant to position cards in a container without having to attach an instance to it immediately. It is especially useful when dragging card around to show where they will be dropped. Usually you should not have to worry about it, placeholders are managed by the containers.

Instance layer (CardInstance class)

`res://addons/cardengine/store/card_instance.gd` (source)

The instance layer is the last piece of this small puzzle. An instance is used to manipulate a card's data in-memory. There is 2 reasons for instances to exist:

- As seen previously, the card's ID is unique in the database, but when playing multiple copy of the same card can exist. Therefore we need an additional information to identify cards in-memory, in this case the reference, accessible using the `CardInstance.ref()` function.
- As part of many card game gameplay, cards get modified over time, by other cards or other game mechanics. Instances allows you to store such modifiers for each card, as even copy of the same card can have different modifiers.

An instance store a reference to the data, that you can access using the `CardInstance.data()` function.

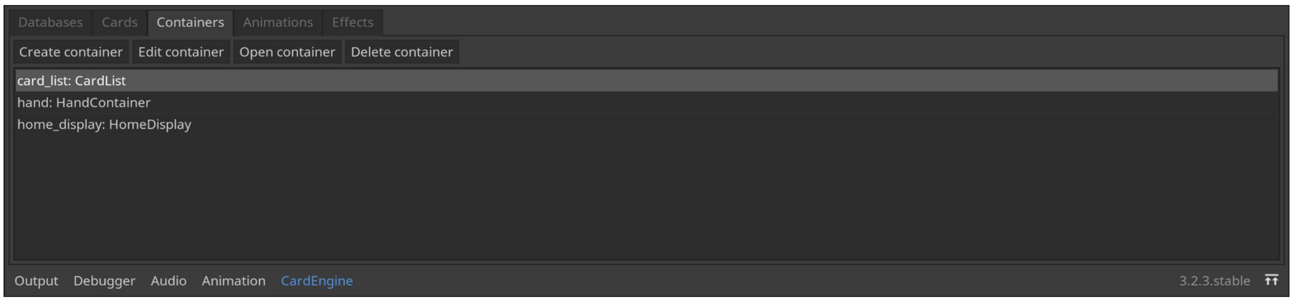
Containers

AbstractContainer scene:
`res://addons/cardengine/container/abstract_container.tscn` (source)
ContainerManager class:
`res://addons/cardengine/container/container_manager.gd` (source)

Overview

Containers are a way to display multiple cards from a store with many options for the layout and animations. Where containers really shine is that they come with a UI which allows you to create them without coding.

Managing containers



card_list: CardList
hand: HandContainer
home_display: HomeDisplay

Creation

To create a container press the "Create container" button. Fill in the ID and the name. The ID cannot contain spaces, can only contain a-z A-Z 0-9 characters and cannot start with a number. The name can be any non-empty string of characters.

Changing ID and name

The ID cannot be changed.

The name can be changed by double-clicking the container in the list.

Deletion

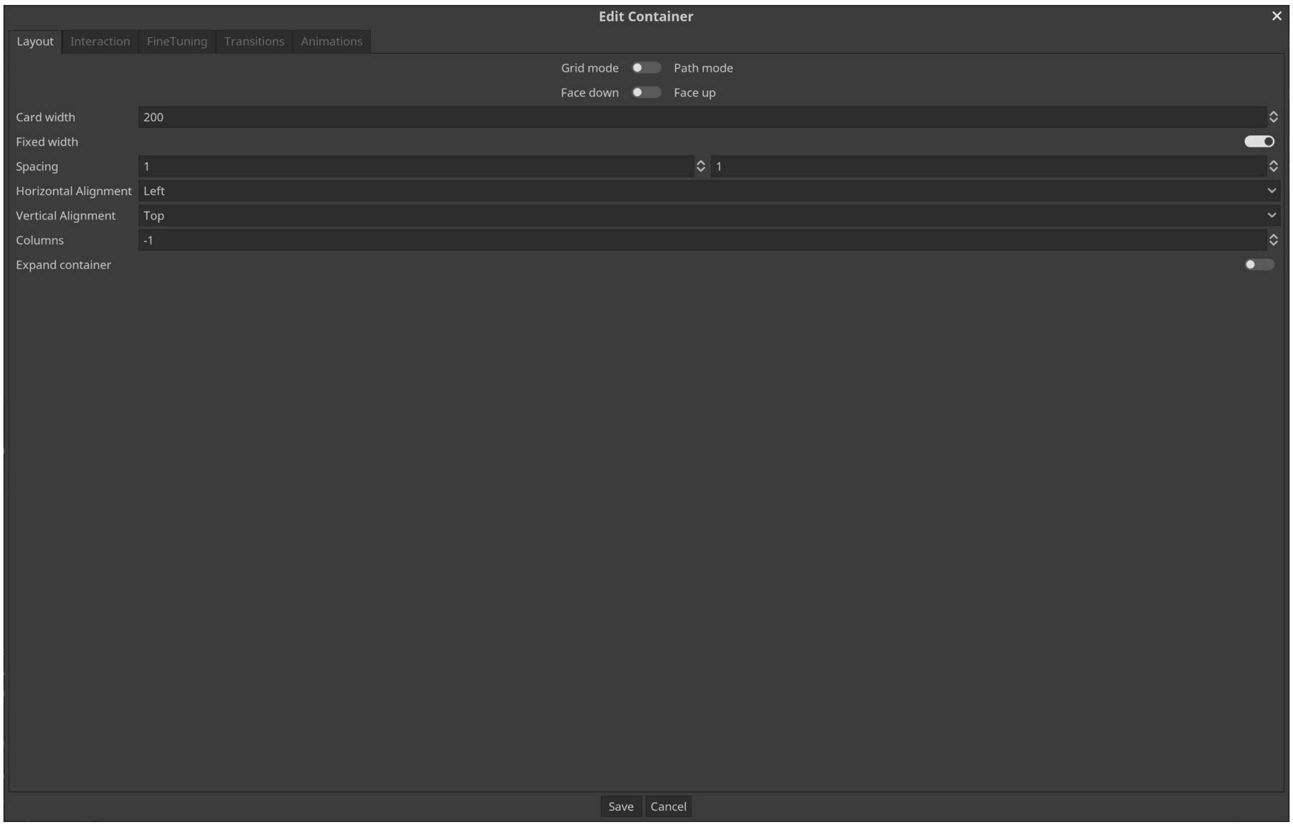
To delete a container, select it in the list and press the "Delete container" button. You will have to validate the delete with the confirm dialog. Be careful, deleting a container is not reversible. Note that only the "private" part of the container is deleted, to completely delete your container, you have to also delete the corresponding folder `res://containers/<container ID>` manually.

Opening

You can open your container scene in the Godot editor by pressing the "Open container" button.

Edit containers

To start editing your container press the "Edit container", you will be presented with the following dialog window:



Grid mode / Path mode
Face down / Face up
Card width: 200
Fixed width
Spacing: 1 | 1
Horizontal Alignment: Left
Vertical Alignment: Top
Columns: -1
Expand container

<u>Important</u>: *no* modification is saved until you press the "Save" button at the bottom of the dialog.

Layout tab

The layout tab is where you choose how cards are positioned in the container.

The following properties are common to both modes

<u>Mode</u>

Containers can follow 2 modes for their layout. The "Grid mode" which positions the cards in rows and columns. The "Path mode" which positions the cards along a path that you draw (see Customizing containers below).

<u>Face</u>

You can choose which card face the container presents you by default.

<u>Width</u>

When you turn on "Fixed width", cards will all have their width set to "Card width". Otherwise the card width is calculated to fit in the grid width or on the path length.

<u>Horizontal spacing</u>

Defines the spacing of the card on the horizontal axis as a ratio of the card's width. A value of 1 means cards will touch each other but without overlapping. A value below 1 means cards will overlap, example: 0.5 means cards will overlap by a half. A value greater than 1 means there will be space between cards, example: 1.25 means there will be a space a quarter the width between them.

The following properties are only applicable to the "Grid mode"

Vertical spacing

Same as the horizontal spacing but on the vertical axis.

Alignments

In case "Fixed width" is on, there is the possibility cards will not fill all the container space. "Horizontal alignment" defines how cards are positioned horizontally in this case. "Vertical alignment" is the same but vertically.

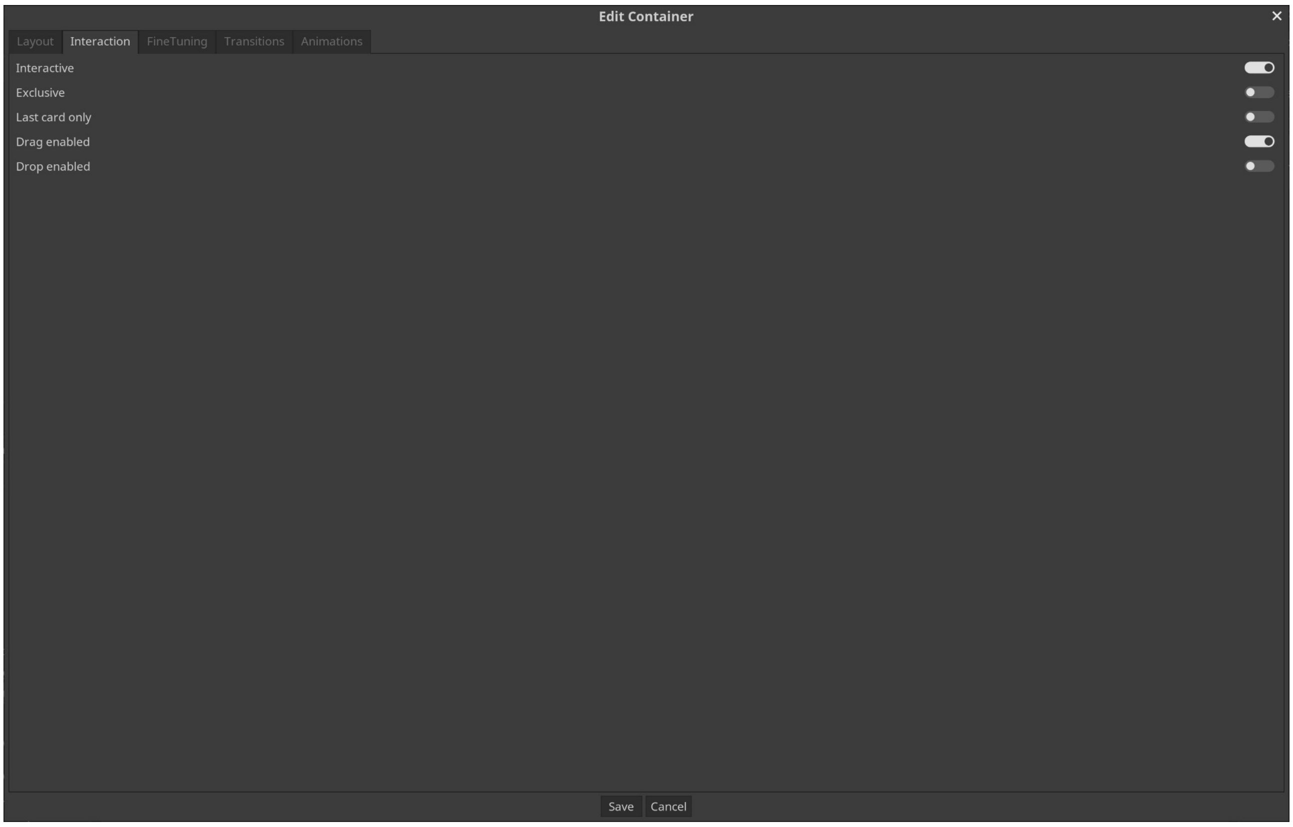
Columns

It defines the number of columns the container will use to layout the cards. The number of rows being then defined by the number of cards divided by the number of columns rounded up. Example: with 10 cards and 3 columns, you will have 4 rows, the last one with only 1 card.

Expand containers

If on, the container will adjust its size so no cards is outside its boundaries. The container width will only expand if fixed width is on. This option is particularly useful when placing the container inside a ScrollContainer to activate the scrolling.

Interaction tab



Interactive: on
Exclusive: off
Last card only: off
Drag enabled: on
Drop enabled: off

<u>Interactive</u>

You can deactivate interactivity for cards inside a container. It is mostly useful to create static card display that only need an idle animation.

<u>Exclusive</u>

When activated the focused card will be the only one to receive user inputs. This is useful when cards overlap and you don't want cards behind to gain focus.

<u>Last card only</u>

On interactive container you can tell that only the last card, usually the one on top, will receive user inputs.

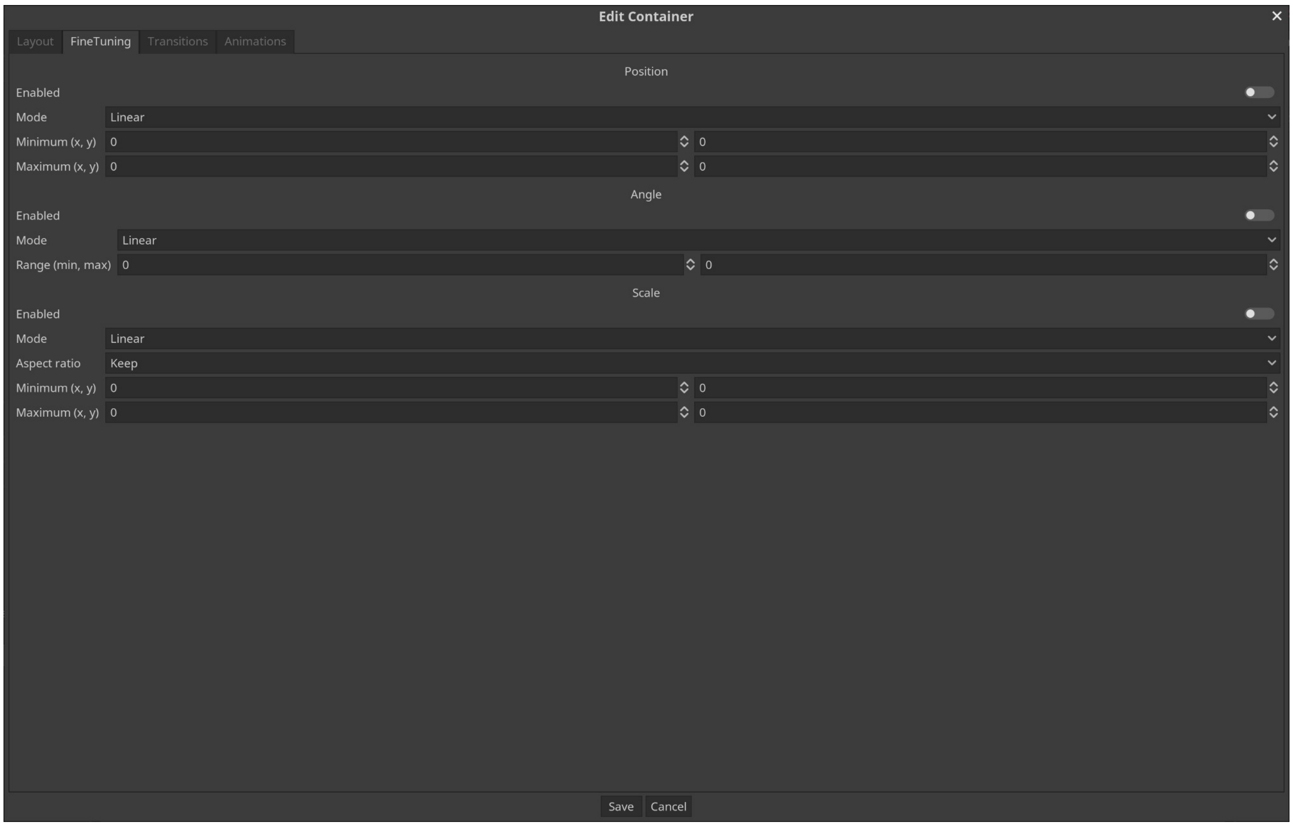
<u>Drag enabled</u>

Enable the container to be a source of drag&drop actions.

<u>Drop enabled</u>

Enable the container to be a receiver of drag&drop actions.

FineTuning tab


Edit Container

Layout | FineTuning | Transitions | Animations

Position

Enabled

Mode: Linear

Minimum (x, y): 0 0

Maximum (x, y): 0 0

Angle

Enabled

Mode: Linear

Range (min, max): 0 0

Scale

Enabled

Mode: Linear

Aspect ratio: Keep

Minimum (x, y): 0 0

Maximum (x, y): 0 0

Save | Cancel

The fine-tuning tab helps you add variation to your layout. You can adjust the position, scale and rotation independently but they all work in the same manner.

You can enabled/disabled them as needed.

Mode

In "Linear" mode the values are interpolated from the minimum at the first card to the maximum at the last card.

In "Symmetric" mode the values are interpolated from the minimum at the first card to the maximum in the middle and back to the minimum for the last card. Note that in case the number of cards is even, the middle is between 2 cards, meaning that no card will actually reach the maximum.

In "Random" mode each card receive a value picked randomly between the minimum and maximum.

Range

A range is compose of two values, a minimum and a maximum. In case of position and scale the value is a 2D vector to treat the horizontal and vertical value independently. In case of the rotation the value is given in degree.

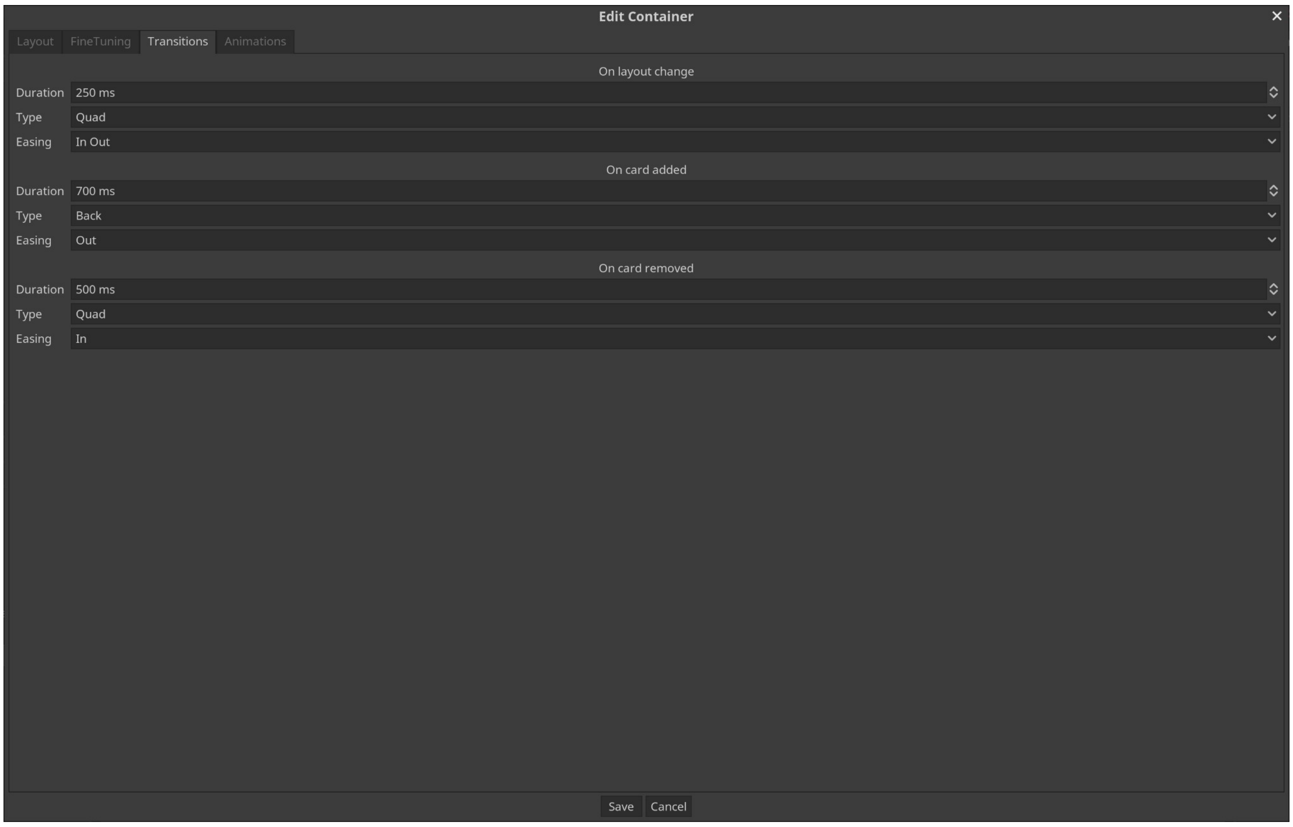
Note: values are relative to the values set by the layout, position and rotation will be added and the scale will be multiplied.

Aspect ratio (scale only)

In "Keep" mode the horizontal scale is applied to the vertical scale.

In "Ignore" mode the horizontal and vertical scales are calculated independently.

Transitions



On layout change

Duration 250 ms
Type Quad
Easing In Out

On card added

Duration 700 ms
Type Back
Easing Out

On card removed

Duration 500 ms
Type Quad
Easing In

The transitions tab has to do with changes to the layout and how to animate them. Currently CardEngine supports 3 types of change:

- On layout change: is when a card is already in the container and a change to the layout is triggered, during event like a resizing, sorting or filtering.
- On card added: is when a card was not present previously on the container and has to be added to it, happens when cards are added to the store or filters are removed.
- On card removed: is when a card was present in the container but no-longer is, it happens when cards are removed or filters are added.

Note: the card added and removed transition rely on an external node to function see Customizing containers below.

Duration

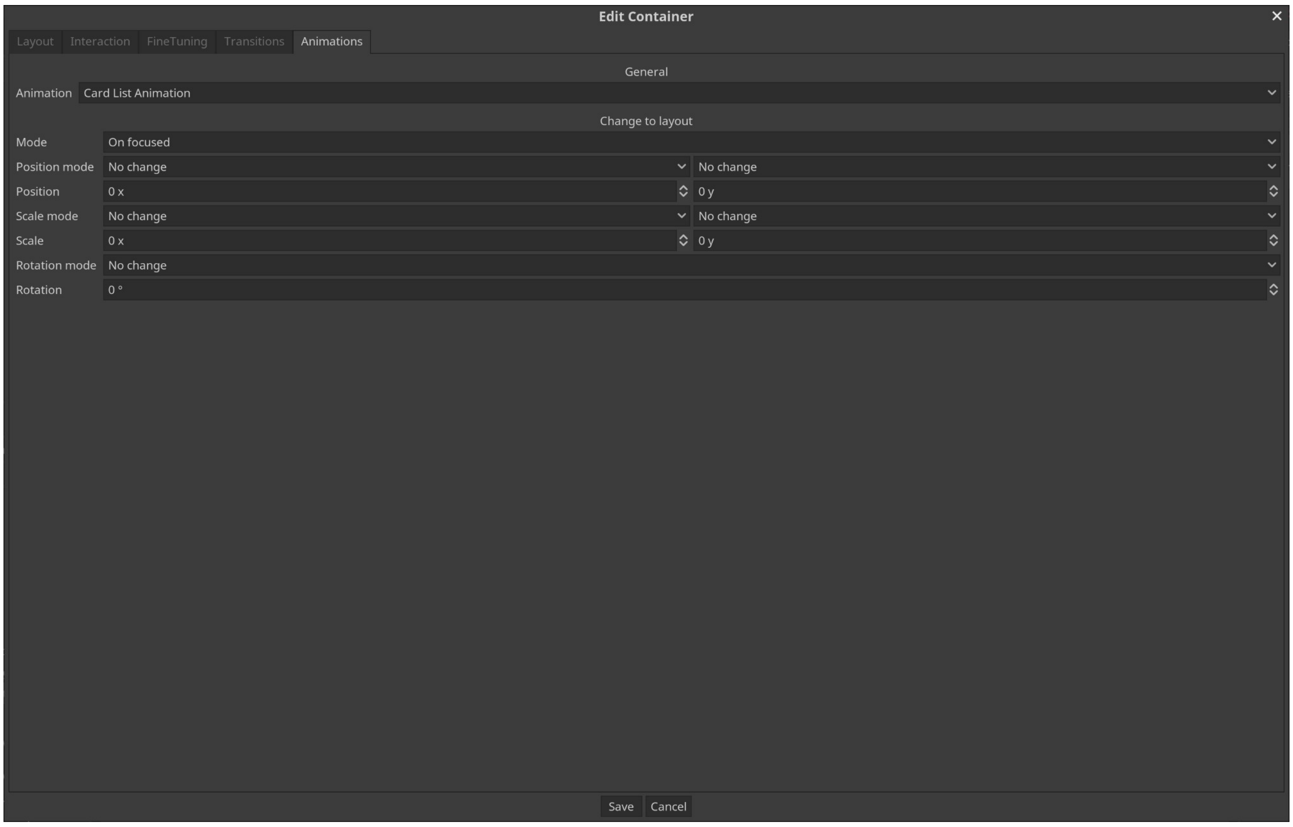
Defines the duration of the transition in milliseconds.

Type and Easing

Defines the curve the transition will follow to smooth out the movement (see this cheat sheet for reference).

Note: transitions are applied to position, rotation and scale all at once.

Animations



General

Animation	Card List Animation	

Change to layout

Mode	On focused	
Position mode	No change	No change
Position	0 x	0 y
Scale mode	No change	No change
Scale	0 x	0 y
Rotation mode	No change	
Rotation	0 °	

The animations tab is about how the container will animate its cards based on interaction with the mouse.

Animation

You can choose from the list the one animation you want. If you don't want any animation, just select "None".

It is also possible to make some adjustments to the layout to help make your animations more predictable especially in the case you made some fine-tuning.

Mode

Defines when this adjustments will take place. Currently it can be "On focused" (mouse hover) or "On activated" (mouse pressed). If "On focused" is selected adjustments will remain through the active phase too.

Adjustments can made to the position, scale and rotation. They all function on the same model.

Mode

"No change" means no adjustment will be made.

"Relative change" means the adjustment will be added (position, rotation) to, or multiplied (scale) with, the value set by the layout.

"Absolute change" means the adjustment will totally overwrite the value set by the layout.

Value

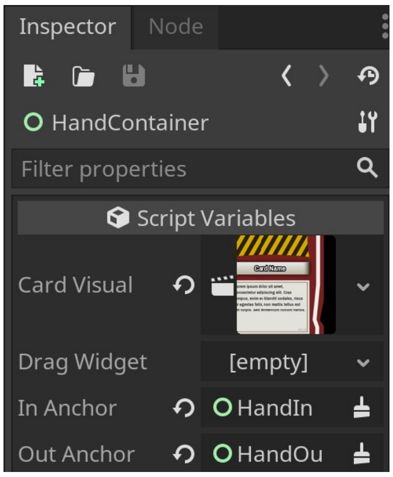
For position and scale the horizontal and vertical value can be adjusted independently.

For the rotation the value is in degree.

Customizing containers

As seen above you can already do a lot from the edit dialog window, but there is still some tweaks to make by editing the container scene. You have two choices on how you customize:

- By editing the public container scene that you can find in `res://containers/<container ID>`
- By editing an instance of the scene when you add your container in another scene



Inspector Node

HandContainer

Filter properties

Script Variables

Card Visual

Drag Widget [empty]

In Anchor HandIn

Out Anchor HandOu

Card Visual (mandatory)

Use this property to set the card you want to be used in the container. This must be a scene inheriting from AbstractCard (for more details).

Drag Widget

Use this property to set the widget that can be used instead of the card in a drag and drop interaction. This must be a scene inheriting from Control (example: `res://containers/card_list/card_list_drag.tscn` source).

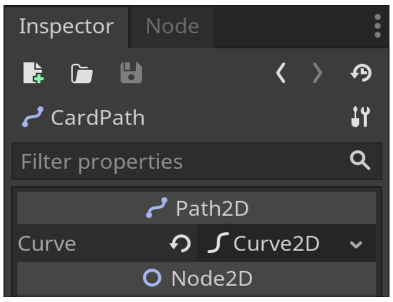
In Anchor

Use this property to set a node that will be used for the added card transition (see Transitions above). The node can be a Node2D or a Control. Its position, scale and rotation will be used to define the initial transform for newly added cards.

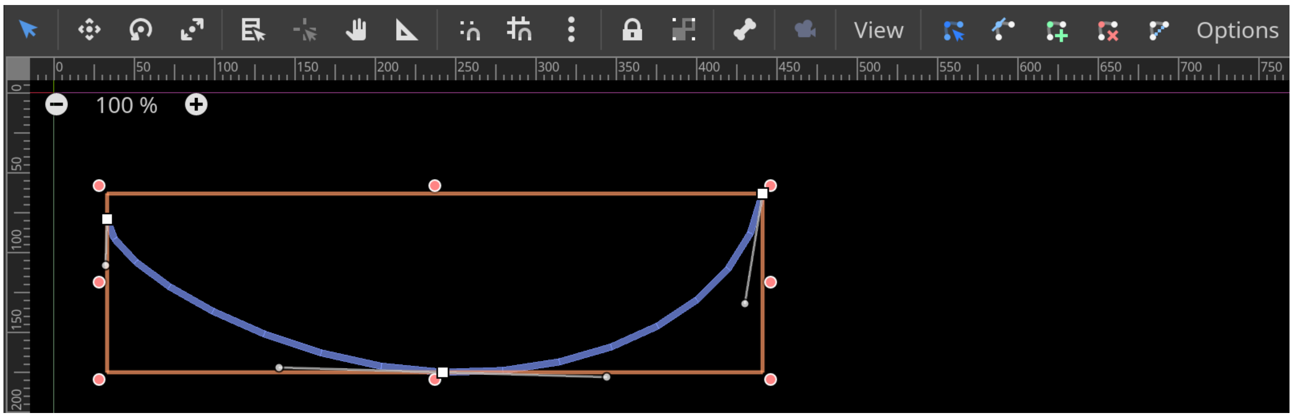
Out Anchor

Same as In Anchor but for removed cards.

Note: due to limitation in the Godot API, the in and out anchors should be siblings of the container so they have the same origin. This problem is most notable for the position, if the anchors and containers are not siblings, there is a risk that the cards appearing, or disappearing, would not do so at the intended position.



In case you have setup your container in "Path mode", you have to create a path for your cards to follow. To do so, create a new Curve2D for the "Curve" property of the "CardPath" node of your container scene. You can now use the Godot Editor tools to draw your path:



<u>**Note**</u>: for you cards to follow the path as drawn on screen, make sure the CardPath transform is at its default values. Tips: move the points of the path and not the whole path.

Adding card to containers

The last step to have a fully functioning container is to add cards to display. This is the only part where you need to write some code. Cards in containers come from stores. Here an example from the demo, that is used to display 3 cards at the top-left of the main menu, the code is added to the menu screen scene `_ready` function:

```
var db = CardEngine.db().get_database("main")
var q = Query.new()
var store = CardPile.new()

var cards = q.from(["rarity:common"]).execute(db)

store.populate(db, cards)
store.shuffle()
store.keep(3)

_display.set_store(store)
```

<u>Quick explanation</u>

1. We retrieve the "main" database
2. We create a database query
3. We create a store

4. We retrieve all the "common" "rarity" cards from the db
5. We use the result to populate our store
6. We shuffle the store to randomize the order of the cards
7. We keep 3 cards
8. We set the store to be the container store

For a more in depth explanation on stores and queries click here.

Animations

AnimationData class: `res://addons/cardengine/animation/animation.gd` (source)
AnimationManager class:
`res://addons/cardengine/animation/animation_manager.gd` (source)

Overview

Animations are a way to animate cards while interacting with the player. An animation is organized as a chain of animation blocks (AnimationBlock class: `res://addons/cardengine/animation/animation_block.gd` source). Each block has 3 sequences (AnimationSequence class: `res://addons/cardengine/animation/animation_sequence.gd` source), the position sequence, the scale sequence and the rotation sequence. Sequences are an ordered list of steps (AnimationStep class: `res://addons/cardengine/animation/animation_step.gd` source). A step is composed of value (StepValue class: `res://addons/cardengine/animation/step_value.gd` source) and a transition (StepTransition class: `res://addons/cardengine/animation/step_transition.gd` source).

Hopefully, you don't need a deep understanding of what all these classes do. CardEngine has a UI to edit animations, see below.

The chain

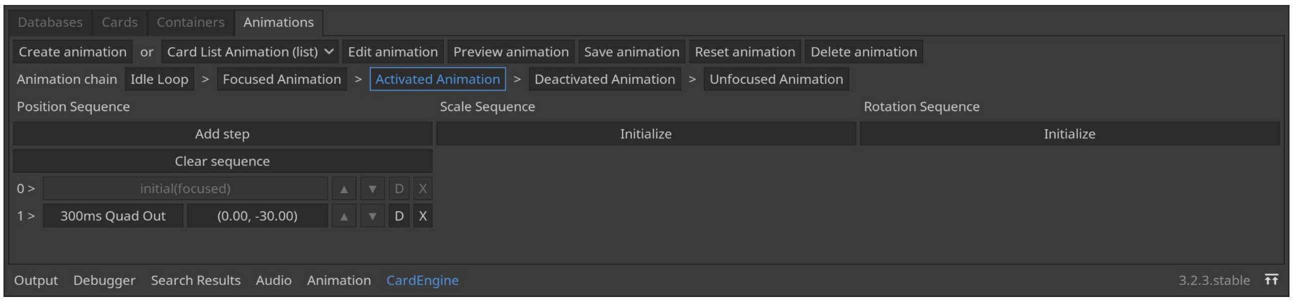
But first, it is important to understand how the animation chain works. The chain property comes from the fact that the start of one block is dependent of how the previous block ended. The chain is built has follow:

- Idle loop: this animation block is played in a loop when the card is not interacted with. It starts and stops at the origin.
- Focused animation: this animation block is played once when the mouse enter the card area. It starts at the origin and stops at its last step.
- Activated animation: this animation block is played once when the mouse is pressed. It starts at the last step of the focused animation and stops at its last step.

- Deactivated animation: this animation block is played once when the mouse is released. It starts at the last step of the activated animation and stops at the last step of the focused animation.
- Unfocused animation: this animation block is played once when the mouse exit the card area. It starts at the last step of the focused animation and stops at the origin. At this point, when this animation block finishes, the chain returns to the idle loop.

Each block, and each sequence of each block, is optional. Note that if there is no focused animation block/sequence the last step is the origin, the same for the activated animation block/sequence.

Managing animations



Creation

To create an animation press the "Create animation" button. Fill in the ID and the name. The ID cannot contain spaces, can only contain a-z A-Z 0-9 characters and cannot start with a number. The name can be any non-empty string of characters.

Changing ID and name

The ID cannot be changed.

The name can be changed by pressing the "Edit animation" button.

Preview

You can have a preview of your animation at any moment, even before saving, by pressing the "Preview animation" button. A dialog window will appear with a card playing your animation in the middle. You can use your mouse to go through the animation chain.

Saving

You can save your animation by pressing the "Save animation" button.

Important: no modification will be saved until your press this button.

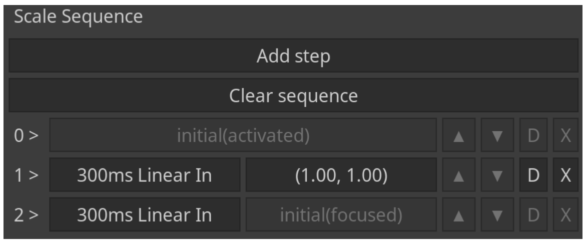
Reset

If you made unwanted change that are still not saved, you can press the "Reset animation" button to return to the previous saved version.

Deletion

To delete an animation, select it in the list and press the "Delete animation" button. You will have to validate the delete with the confirm dialog. Be careful, deleting an animation is not reversible.

Edit animations

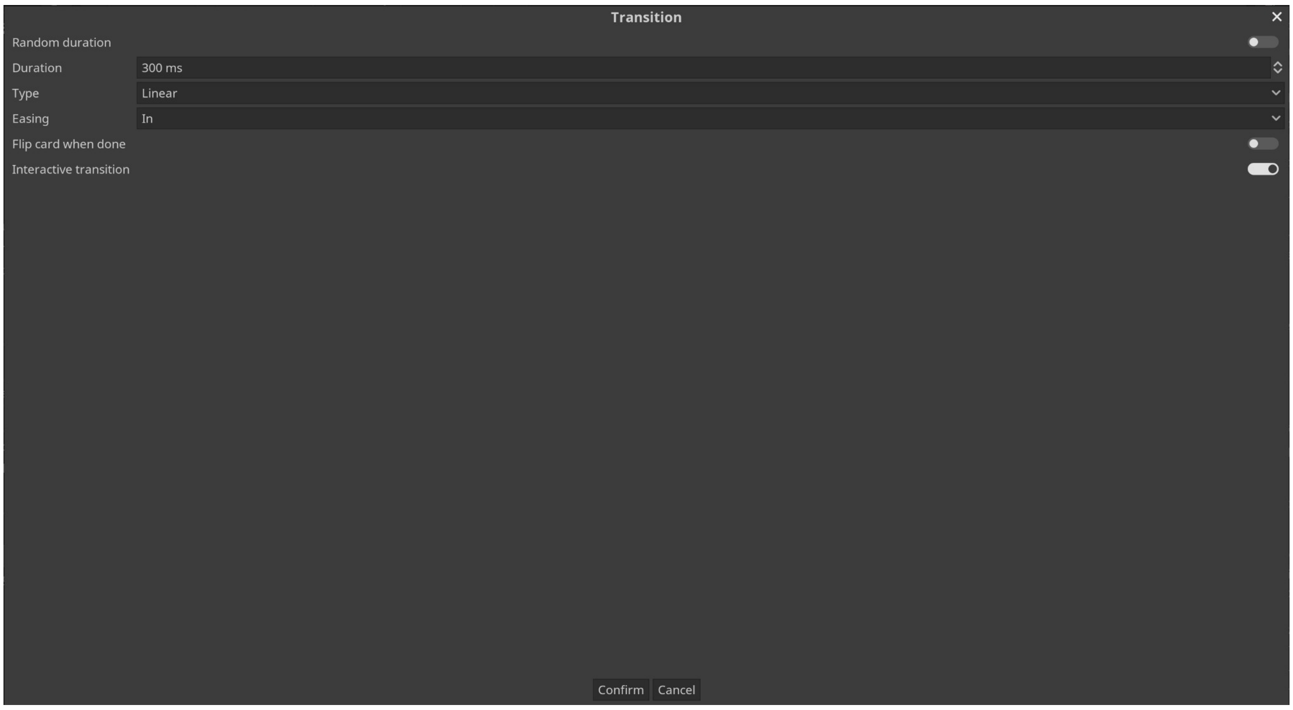

Scale Sequence
Add step
Clear sequence
0 > initial(activated) ▲ ▼ D X
1 > 300ms Linear In (1.00, 1.00) ▲ ▼ D X
2 > 300ms Linear In initial(focused) ▲ ▼ D X

To start editing your animation, first choose the animation block you want to edit. You will be presented with the 3 empty columns, 1 for each sequence. You can initialize the sequence you want to use by pressing the "Initialize" button in the corresponding column. Depending on the selected animation block, the initial sequence varies. Usually a sequence starts at an initial value, and stops at another initial value, that cannot be changed or removed. The actual value of the initial value, and the presence of an initial value at the end, is depending on where the block is in the chain. The starting and ending initial values can be different also.

You can add as many steps as you want with the "Add step" button. Tips: don't make your animations too long to avoid the feeling of a sluggish UI. The "Clear sequence" button removes every steps in the sequence, including those that are not removable individually.

Each line represents a step. Each step has two main buttons, except for the step 0, the left button is to change the transition and the right the value. Each step is followed by 4 tool buttons. The "▲" and "▼" buttons allow you to change the order of the steps by moving them up and down. The "D" button is for duplicating the step, which is added at the end of the sequence, but before the initial value if present. The "X" button deletes the step, deleting a step is not reversible.

Transition editing


Transition

Random duration

Duration 300 ms

Type Linear

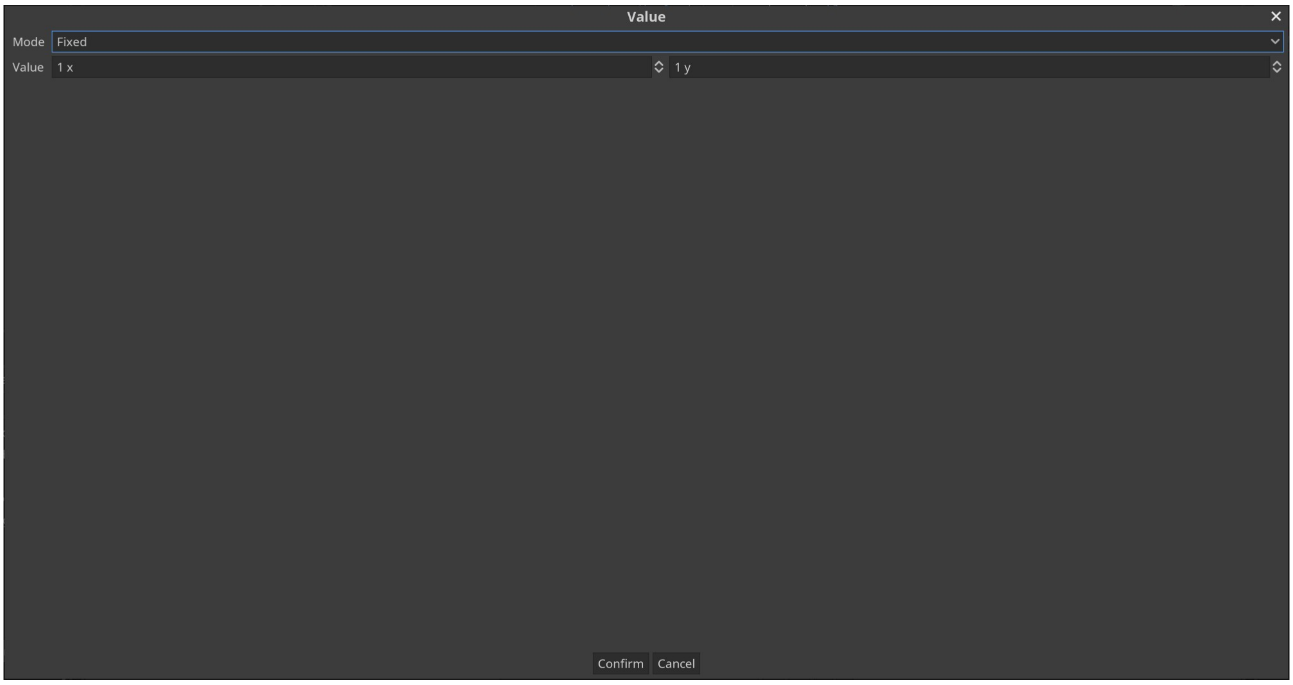
Easing In

Flip card when done

Interactive transition

A transition helps you smooth out the change of values between the previous step and this step, over a given duration. You can activate the "Random duration" so a random duration will be picked between a minimum and a maximum. You can change the duration in milliseconds with "Duration". With "Type" and "Easing" you control the smoothing curve (see here for reference). Additional features are available for creating more complex animations. You can tell CardEngine to flip the card at the end of the transition. You can disable interactivity during a transition if you want a step to go through no matter the input from the player.

Value editing


Value

Mode Fixed

Value 1 x 1 y

A value defines the state you want your step to reach at the end of the transition. There is 3 modes to define a value:

- Initial: tell the step to go back to the initial value as defined by the position of the block in the chain
- Fixed: tell the step to use the given value
- Random: tell the step to use a random value between a given minimum and maximum

With "Value" you defined the value for the "Fixed" mode or the minimum for the "Random" mode.

With "Range" you defined the maximum for the "Random" mode.

Note: values are relative to the values defined by the layout.

Stores

Overview (AbstractStore class)

`res://addons/cardengine/store/abstract_store.gd` (source)

Stores help you managed the card in-memory. Contrary to a database, a store can have multiple copy of the same card and modify them on the fly, thanks to `CardInstance` (learn more here). In addition, stores keep meta-information on cards.

Typically, stores is what you will be using in your scripts to manipulate cards to create your gameplay. CardEngine offers specialized stores like deck, pile and hand, for basic gameplay features which are common to most card games.

Signals

As store are usually used in coordination with visual objects to display their state, either be a Container or other Node, it is a good idea to use signals to keep the visual updated (Containers do it automatically). You have two choices on how you want to manage this, you can used the specialized signals `card_added`, `cards_added`, `card_removed`, `cards_removed`, `filtered` or `sorted`. Or you can use the generic `changed` signal which is always emitted for every change made to the store.

Properties

Stores have the following properties:

- `save_id: String` contains the ID of the store as saved on disk (read-only)
- `save_name: String` contains the name of the store as saved on disk (read-only)

Adding

You can add cards to a store using either the "add" functions or the "populate" functions:

- Use "add" functions if you already have CardInstance objects on hand, you can use `add_card(CardInstance)` or `add_cards(CardInstance[])`. Tip: you can use CardInstance.duplicate() if you want modification to the added instance to not affect the original instance
- Use "populate" functions to create new CardInstance objects from database entries, you can use `populate(CardDatabase, ID[])` or `populate_all(CardDatabase)`, the first creates an instance for each ID of the list, the second creates an instance for each database entry

Utility

Stores offer some basic utility functions:

- `count()` returns the number of cards in the store (affected by the filter)

- `cards_count()` returns a `Dictionary` with the number of cards per ID, IDs being the keys and numbers the values
- `count_for(ID)` returns the number of card with the given ID
- `is_empty()` returns true if there is no card in the store (affected by the filter)
- `has_card(reference)` returns true if the store contains an instance with the given reference (affected by the filter)

Saving and loading

See this page

Retrieving

You can retrieve cards using the following functions:

- `cards()` returns all the cards as an array of `CardInstance` (affected by the filter)
- `get_card(int)` returns the card at the given index or `null` if the index is out of bound (affected by the filter)
- `get_first()` returns the first card in the store or `null` if the store is empty (affected by the filter)
- `get_last()` returns the last card in the store or `null` if the store is empty (affected by the filter)

Finding

You can find cards using the following funtions:

- `find(ID)` returns an array of `CardInstance` with the given ID or an empty array if not found
- `find_first(ID)` returns the first `CardInstance` with the given ID or null if not found
- `find_last(ID)` returns the last `CardInstance` with the given ID or null if not found

Removing

When it comes to removing cards you can use the following functions:

- `clear()` removes all the cards and the filter
- `remove_card(reference)` removes the card with the given reference
- `remove_first()` removes the first card
- `remove_first(ID)` removes the first card with the given ID
- `remove_last()` removes the last card
- `remove_last(ID)` removes the last card with the given ID
- `keep(int)` only keeps the first given N cards

Transferring

You can transfer cards between stores either by moving them or by copying them.

Moving

Moving cards means that the cards will be removed from the store once transferred:

- `move_cards(AbstractStore)` moves all the cards from one store to the other
- `move_card(reference, AbstractStore)` moves a card instance with the given reference from one store to the other
- `move_random_card(AbstractStore)` moves a random card from one store to the other

Copying

Copying cards means that the cards transferred to the other store will be different instances:

- `copy_cards(AbstractStore)` copies all the cards from one store to the other
- `copy_card(reference, AbstractStore)` copies the card instance with the given reference from one store to the other
- `copy_random_card(reference, AbstractStore)` copies a random card from one store to the other

Filtering

As seen above some functions are affected by a filter. You can set a filter on your store, with `apply_filter(Query)`, so only some cards remain accessible without removing them from the store. Look at Query below to learn how to write a filter.

Sorting

Store can be sorted using one or more criteria at once. You can sort your store by calling `sort(Dictionary)`. You build your sort criteria as a GDScript Dictionary following this rules:

- For categories the entry key is of the form `"category:<meta-category>"` and the value is an array of categories in the order you want them to appear, example: `sorting["category:rarity"] = ["common", "uncommon", "rare", "mythic_rare"]`
- For values the entry key is of the form `value:<value ID>` and the value is boolean where true is for ascending order and false is descending order, example: `sorting["value:mana"] = true`
- For texts the entry key is of the form `text:<value ID>` and the value is boolean where true is for ascending order and false is descending order, example: `sorting["text:name"] = true`

You can combine as many criteria as needed, they will be treated in the order you add them.

Meta-information

Stores also keep data about what kind of data its cards have:

- About categories: with `categories()` you get all the different meta-categories and how many cards have it, and per meta-categories you get all the categories and how many cards have this category as a Dictionary, with `get_meta_category(String)` you get the same information but just for one meta-categories
- About values: with `values()` you get all the different value ID as an Array
- About texts: with `texts()` you get all the different text ID as an Array

Example:

```
# Print the number of cards with a rarity meta-category
var categs = store.categories()
print("Count %d" % categs["rarity"]["count"])
# Or
var rarity = store.get_meta_category("rarity")
print("Count %d" % rarity["count"])


# Print all the different rarity categories
var rarity = store.get_meta_category("rarity")
for categ in rarity["values"]:
  print("Category %s" % categ)


# Print the number of cards with a common rarity
var rarity = store.get_meta_category("rarity")
print("Count %d" % rarity["values"]["common"])


# Print all the different value IDs
var values = store.values()
for value in values:
  print("Value %s" % value)


# Print all the different text IDs
var texts = store.texts()
for text in texts:
  print("Text %s" % text)
```

Card Deck (CardDeck class)

`res://addons/cardengine/store/card_deck.gd` (source) (API not finished)

A deck is usually a list of cards which the player is building before or during a game by selecting cards from a database.

Card Pile (CardPile class)

`res://addons/cardengine/store/card_pile.gd` (source) (API not finished)

A pile is usually a list of cards from which the player can draw cards. Piles also have the ability to be shuffled, that will change the order the cards are drawn.

- `draw()` removes the last cards from the store and returns it (emit the `card_drawn` signal)
- `shuffle()` randomizes the order of the cards in the store (emit the `shuffled` signal)

Card Hand (CardHand class)

`res://addons/cardengine/store/card_hand.gd` (source) (API not finished)

A hand is usually a list of cards from which the player choose the card to play.

- `play_card(reference, AbstractStore)` removes the card instance with the given reference and move it to the other store if present (emit `card_played` signal)

Query (Query class)

`res://addons/cardengine/database/query.gd` (source)

Queries have 2 uses, selecting cards from a database or filtering a store, but in both cases they are built the same way. There is 3 types of statements:

- From statements to select the cards by categories
- Where statements to select the cards by values
- Contains statements to select the cards by texts

You build your query around arrays of comma separated statements, such that each element of the array will be interpreted as logical ***or*** statement, and each comma separated as logical ***and*** statement.

Building from statements

A from statement is built using the following structure `<meta-category>:<search string>`. The search string can contain * and ? as wildcards for respectively a string and a single character.

Building where statements

A where statement is build using the following structure `<value ID> <comparison operator> <number>`. There is 5 comparison operators supported: =, <=, <, >= and >. The number is any integer, positive or negative.

Building contains statements

A contains statement is build using the following structure `<text ID>:<search string>`. The search string is case insensitive.

Examples

```
# Cards must be of "class" "creature" and the "creature_type" must start with
"human_", or the "class" must be "sorcery"
var q = Query.new()
q.from(["class:creature,creature_type:human_*", "class:sorcery"])
```

```
# Cards must have a "mana" value superior or equal to 2 and strictly inferior to
5
var q = Query.new()
q.where(["mana >= 2,mana < 5"])


# Cards "name" must contain "time" or "master"
var q = Query.new()
q.contains(["name:time", "name:master"])


# You can combine all statements in one line
var q = Query.new()
q.from(["class:creature,creature_type:human_*", "class:sorcery"]).where(["mana
>= 2,mana < 5"]).contains(["name:time", "name:master"])


# You can use the query to populate a store from a database
var db = CardEngine.db().get_database("main")
var q = Query.new()
var store = CardPile.new()

var result = q.from(["class:creature,creature_type:human_*",
"class:sorcery"]).where(["mana >= 2,mana < 5"]).contains(["name:time",
"name:master"]).execute(db)

store.populate(db, result)


# You can use the query to filter a store
var db = CardEngine.db().get_database("main")
var q = Query.new()
var store = CardPile.new()

store.populate_all(db)

q.from(["class:creature,creature_type:human_*", "class:sorcery"]).where(["mana
>= 2,mana < 5"]).contains(["name:time", "name:master"])

store.apply_filter(q)
```

Drag and drop

Overview

Drag and drop is a key component of any card game. CardEngine relies on Godot own drag and drop implementation, especially the following functions of the Control node:

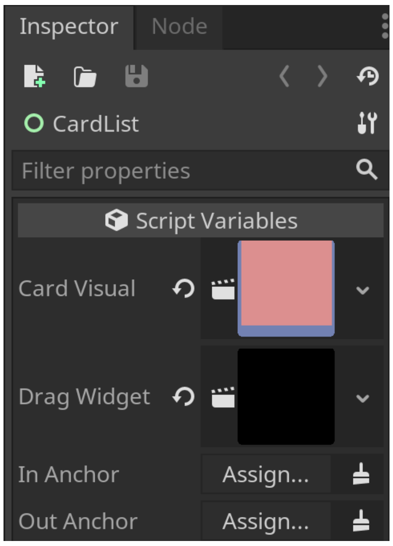
- `get_drag_data()`: documentation
- `can_drop_data()`: documentation
- `set_drag_preview()`: documentation

Most of the complexity is however hidden and and requires minimal actions from you. Containers, cards visual layer and the DropArea widget take care of all the details for you.

Container

Containers can be source for a drag and drop action. To activate this feature you have to make sure drag is enabled for your container, see here for more details. This is all you have to do, however by default the drag action will move the card itself with the mouse cursor. If you want your drag action to have a different visual, you can set a widget for that (see below).

Drag widget


Inspector Node

CardList

Filter properties

Script Variables

Card Visual

Drag Widget

In Anchor Assign...

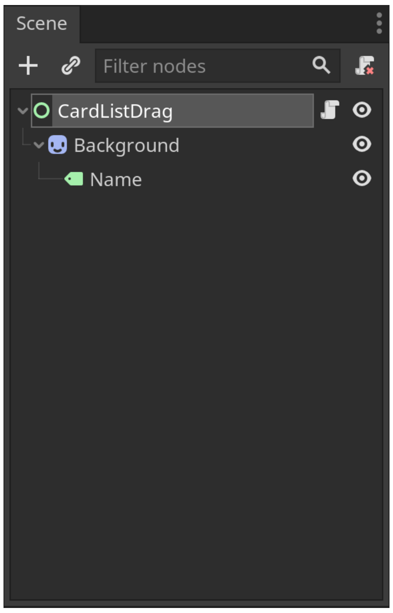
Out Anchor Assign...

Assigning a drag widget

You define which widget you want to use using the "Drag Widget" property of your container.

To do so, open the container scene file under the `res://containers/<container ID>.tscn` and select the root node, you will have access to the property in the Inspector. Load the widget using the property editor menu.

Creating a drag widget



A drag widget must be a control based scene. Create a new scene using the "Scene" menu of the Godot editor and choose "User Interface" as a root node. You can then build your widget using any Control or Node2D based nodes. If you want to customize your widget depending on the card being dragged, attach a new script to the root node. In the `_ready()` function you can connect to the `drag_started()` signal of the `GeneralManager` singleton accessible through `CardEngine.general()`. Then you can retrieved the dragged card using the `CardEngine.general().get_dragged_card()` function. Here a complete example from the demo:

```
extends Control

onready var _name = $Background/Name


func _ready() -> void:
    CardEngine.general().connect("drag_started", self, "_on_drag_started")


func _on_drag_started():
    var card = CardEngine.general().get_dragged_card()

    _name.text = card.data().get_text("name")
```

DropArea

DropArea scene: `res://addons/cardengine/core/drop_area.tscn` (source script)

Usage

DropArea is very simple to use, this is a Control based Node. It can be added in any scene as you would add any UI elements. You can then make your script react to a card being dropped into the area by using the `dropped(card, source, on_card)` signal, where:

- card (CardInstance) is the card being drop into the area.
- source (String) is the ID of the container from where the card is coming from.
- on_card (CardInstance) if the DropArea contains cards (typically a container) this gives the card on which the dragged card is being dropped.

Here an example from the demo:

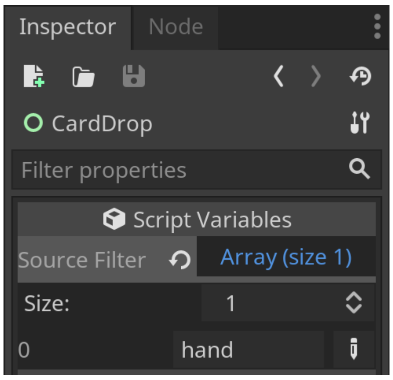
```
func _on_CardDrop_dropped(card: CardInstance, source: String, on_card: CardInstance) -> void:
    var card_mana = card.data().get_value("mana")

    if _mana >= card_mana:
        _hand.play_card(card.ref(), _discard_pile)

        if card_mana < 0:
            _mana = 0
        else:
            _mana -= card_mana

        _update_mana()
```

Filter

By the dragged card

You might not want all cards being dropped into your area, you can set a filter by calling the `set_filter(filter)` function of `DropArea`. The filter is of type `Query`, find out how to write queries at the bottom of this page.

By the source container



You can also filter the cards being dropped by the container they are coming from. To do so, add the allowed container IDs in the "Source Filter" array property of DropArea. If the array is empty the DropArea will accept cards from any source. You can also use the `set_source_filter()` function to set this filter in script.

Special cases

Container to container

Containers can be receivers for a drag and drop action. To activate this feature you have to make sure drop is enabled for your container, see here for more details. You can retrieve the container's DropArea by using the `get_drop_area()` function, it is particularly useful if you need to set some filters on it (see above).

Once activated your container will emit the `card_dropped(card, source, on_card)` signal, connect to it to decide what to do with the card, here an example from the demo:

```
func _on_TokenGrid_card_dropped(card: CardInstance, source: String, on_card: 
CardInstance) -> void:
    if _tokens.count() >= MAX_TOKENS:
        return

    var card_mana = card.data().get_value("mana")

    if _mana >= card_mana:
        _hand.play_card(card.ref(), _discard_pile)
        _tokens.add_card(CardInstance.new(card.data()))

        if card_mana < 0:
            _mana = 0
        else:
            _mana -= card_mana

        _update_mana()
```

Effects

Overview

AbstractEffect class: `res://addons/cardengine/effect/effect.gd` (source)
EffectInstance class: `res://addons/cardengine/effect/effect_instance.gd` (source)
EffectManager class: `res://addons/cardengine/effect/effect_manager.gd` (source)
AbstractModifier class: `res://addons/cardengine/effect/modifier.gd` (source)

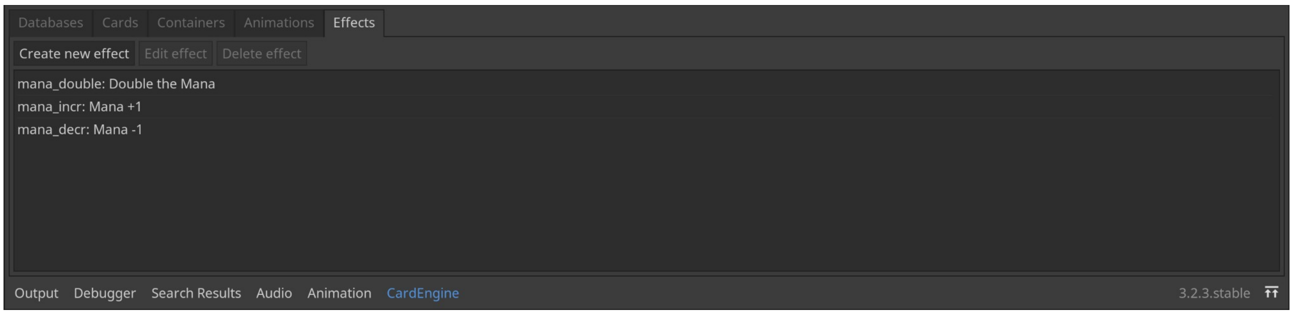
CardEngine offers the possibility to temporarily modify cards' data through the effects system. Effects apply modifiers to the cards of your choosing. Modifiers make change to the data that can be reverted. This is useful when, for example, cards affect other cards, or if you have artifacts which modify cards behavior.

Effects are divided in two layers, the logic layer (AbstractEffect) and the instance layer (EffectInstance). In the logic layer, effects are identified by an unique ID and are what you managed in the editor. As an effect can be applied multiple time, the instance layer helps keeping tracks of what is applied and where.

Effects are not what is directly applied to cards, instead effects add modifiers. The reason is to make it easier for you to create effects that can change multiple data at once.

Effects management is a mix of UI driven operation and scripting. This might change in the future to a more UI driven system.

Creation



mana_double: Double the Mana
mana_incr: Mana +1
mana_decr: Mana -1

To create an effect press the "Create new effect" button. Fill in the ID and the name. The ID cannot contain spaces, can only contain a-z A-Z 0-9 characters and cannot start with a number. The name can be any non-empty string of characters.

Change ID and name

The ID cannot be changed.

The name can be changed by double-clicking the effect in the list.

Deletion

To delete an effect, select it in the list and press the "Delete effect" button. You will have to validate the delete with the confirm dialog. Be careful, deleting an effect is not reversible.

Usage

Applying effects

Before you can apply an effect, you have to instantiate one using the `instantiate("<effect ID>")` function of `EffectManager`. The manager is accessible using the the CardEngine singleton `fx()` function.

Once you have an instance you can apply the effect to a store, using the `apply(store)` function or to a given card, using the `affect(card)` function.

Here is an example from the demo, which apply the "mana_incr" effect to the cards in hand:

```
var fx = CardEngine.fx().instantiate("mana_incr")
fx.apply(_hand)
```

Displaying changes

To make the visual layer of your cards reflect changes made by effects, you have to connect to the signal `modified()` from the instance layer. Example from the demo:

```
func _on_NormalCard_instance_changed() -> void:
    instance().connect("modified", self, "_on_instance_modified")
    _update_data()


func _on_instance_modified() -> void:
    _update_data()
```

Note: the instance changed signal is emitted when a new instance is linked to the visual layer, while the instance modified signal is emitted when the data layer is changed.

The modified data are accessible using the `instance().data()` function. You can access the unmodified data using the `instance().unmodified()` function. To learn more about how to display data, [check the Visual layer section of this page](). Example on how to use both data to change the color of the mana cost text depending on if the modified cost is higher or lower:

```
    var val = instance().data().get_value("mana")
    var orig = instance().unmodified().get_value("mana")

    if val > orig:
        _cost.add_color_override("font_color", Color("ff0000"))
    elif val < orig:
        _cost.add_color_override("font_color", Color("00ff00"))
    else:
        _cost.add_color_override("font_color", Color("ffffff"))
```

Cancelling effects

To cancel an effect you have to call the `cancel()` function of `EffectInstance`. This will remove all the modifiers linked to this effect instance from the affected cards.

Edition

Editing an effect requires to write some code. You can find the effects in the `res://effects` folder. Alternatively, you can click the "Edit effect" button to open the corresping script. For each created effect you have a corresponding script following this naming convention `<effect ID>.gd`. To start editing, open the script with your usual GDScript editor. Example of an effect script when newly created:

```
extends AbstractEffect


func _init(id: String, name: String).(id, name) -> void:
    pass


# Override this to limit the affected cards or leave null to affect all the
cards
func get_filter() -> Query:
    return null


# Override this to returns an array of modifiers applied by this effect
func get_modifiers() -> Array:
    return []
```

Available modifiers

CardEngine has built-in modifiers which you can use with any effects:

- ValueChange: add a given amount to a given value `ValueChange.new(mod_id, stackable, value_id, amount)`
- ValueMultiplier: multiply a given value by a given multiplier, the result is floored `ValueMultiplier.new(mod_id, stackable, value_id, multiplier)`
- ValueSetter: set a given value to the given number `ValueSetter.new(mod_id, stackable, value_id, number)`
- More modifiers to come...

Custom modifiers

If you cannot find a built-in modifier that fits your needs, writing a custom one is straightforward. You have to create a new class extending the `AbstractModifier` class, either in a new script, if you need to reuse it in multiple effects, or as an inner class of the effect, the quickest way. Here an example of a custom modifier, created as an inner class of an effect, which modify the name of a card if present.

```
class CustomModifier extends AbstractModifier:
    func _init(id: String, stackable: bool).(id, stackable) -> void:
        pass
    func modify(card: CardData) -> void:
        if card.has_text("name"):
            var name = card.get_text("name")
            card.set_text("name", "%s (modified)" % name)
```

Adding modifiers

To add modifiers to an effect you have to override the `get_modifers()` function of the effect. This function returns an array of modifiers instance. Instantiating a modifier is done by calling `new()` on the desired modifier class name, example: `ValueChange.new("incr", true, "mana", 1)`. The first parameter is the modifier's ID, important to keep track which modifier has been applied by other effects. The second parameter is to say if the modifier is stackable, a stackable modifier can be applied multiple time instead of just once for non-stackable. Additional parameters can follow depending on the instantiated modifier. Example:

```
func get_modifiers() -> Array:
    var modifiers := []

    # Adds the buit-in ValueChange modifier
    modifiers.append(ValueChange.new("incr", true, "mana", 1))
    # Adds the custom modifier CustomModifier
    modifiers.append(CustomModifier.new("custom", false))

    return modifiers
```

Setting up a filter

If you want your effect to only affect certain cards you can override the `get_filter()` function. This function returns a Query, to learn how to write queries go at the bottom of this page.

Example:

```
func get_filter() -> Query:
    var filter := Query.new()
    # Effect only affects cards with a mana value above or equal zero
    return filter.where(["mana >= 0"])
```

Going deeper

Once familiar with the concepts above, you can explore the source code of the game screen from the demo to get a better understanding on how all this is working together. Reading the source code of the visual layer implementation "normal_card" is also a good idea.

User data

Overview

There are many ways you might want to save user data, in binary, in XML or in JSON, locally or online. CardEngine cannot offer all this possibilities out-of-the-box. Instead CardEngine has interfaces which make it easier to adapt it to your needs. At the moment, there is only one interface to manage the player's stores.

UDStores

Class: `res://user_data/ud_stores.gd` (source)

This class offers the following functions, accessible through the `UserStores` singleton:

- `saved_stores()` returns a Dictionary containing the saved stores with IDs as keys and names as values
- `load_store(ID, AbstractStore)` loads the saved store with the given ID into the given store
- `save(ID, name, AbstractStore)` save the store data with the given ID and name

This interface comes with a default implementation bases on the `ConfigFile` built-in Godot class. However, the following functions can be customized to your needs:

- `_get_stores()`

```
# Retrieves the list of all saved stores
# Customized this function to your need
# Returns a Dictionary:
# {
#   "<store id>": "<store name>",
#   "<store id>": "<store name>",
#   "<store id>": "<store name>",
#   ...
# }
func _get_stores() -> Dictionary:
  ...
```

- `_get_store(ID)`

```
# Retrieves the store with the given ID
# Customized this function to your need
# Returns a Dictionary:
# {
#   "id": "<store id>",
#   "name": "<store name>",
#   "cards": [
#     {"id": "<card id>", "source": "<database id>},
#     {"id": "<card id>", "source": "<database id>},
#     {"id": "<card id>", "source": "<database id>},
#     ...
#   ]
# }
func _get_store(id: String) -> Dictionary:
  ...
```

- _post_store(ID, name, Array)

```
# Saves a store with the given ID, name and cards
# Customized this function to your need
# Cards Array:
# [
#   {"id": "<card id>", "source": "<database id>},
#   {"id": "<card id>", "source": "<database id>},
#   {"id": "<card id>", "source": "<database id>},
#   ...
# ]
func _post_store(id: String, name: String, cards: Array) -> void:
  ...
```

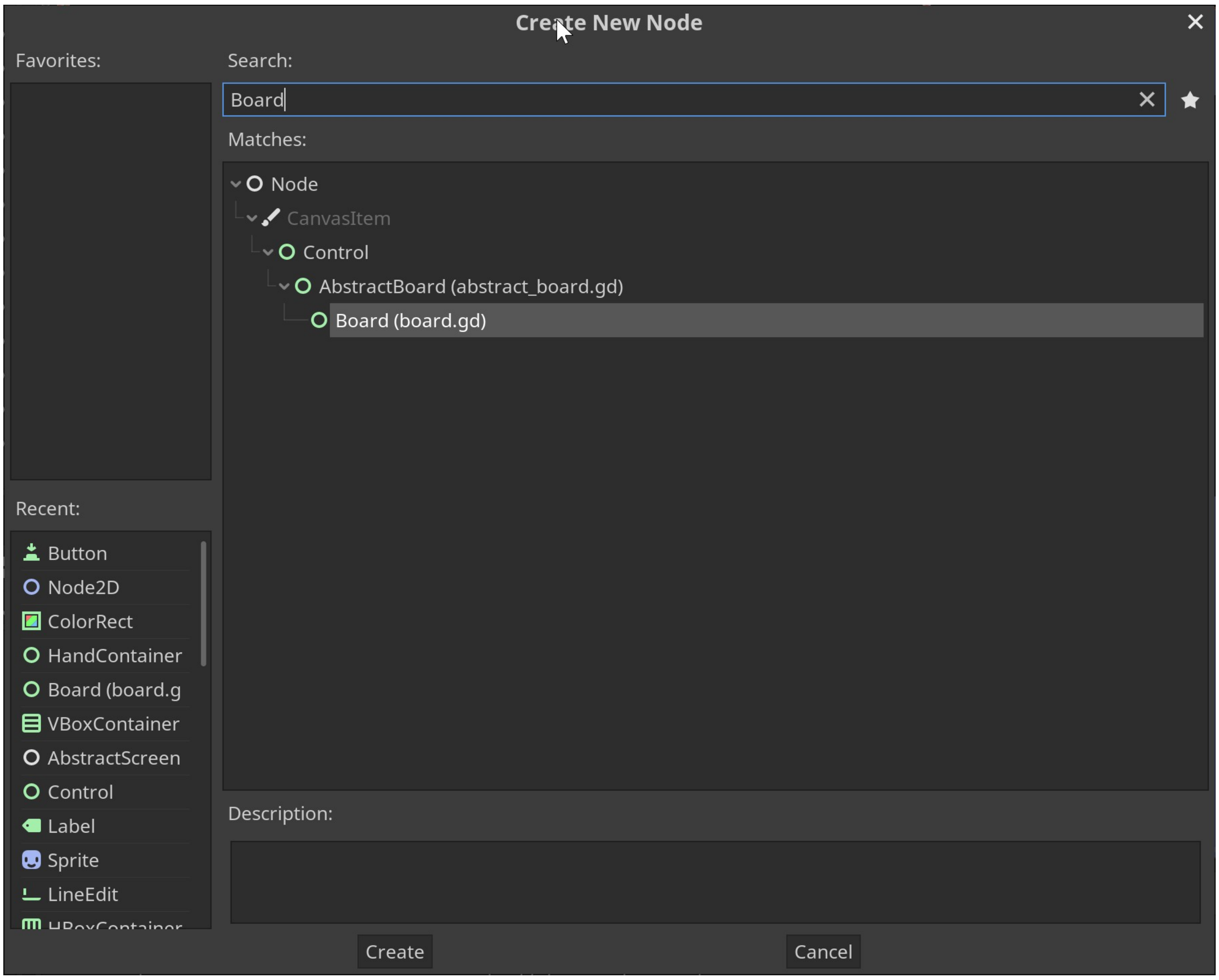
Board widget

Overview

Board class: `res://addons/cardengine/container/board.gd` (source)

In a basic approach each container works independently, they don't know from where cards are coming and where they are going. The Board widget allows to link multiple containers together allowing to have seamless transitions when cards move from one to another.

Usage



Using the Board widget is very simple, just add it to your scene as you do for any other node. Then you can place as many containers on it as you need as children nodes. That's all you have to do.

Note: at the moment for the Board widget to work properly, containers have to be direct children of the board.

Demo

Please check out the demo to see how it works, by pressing the Board Demo button in the main menu or looking at the source code.

Utilities

Overview

CardEngine provides some utilities to help with game development in general. At the moment the following tools are available:

- The screen system helps with organizing your game.
- The pseudo-random number generator helps with creating seeded game.

Screen system

AbstractScreen class: `res://screens/abstract_screen.gd` (source)

Introduction

The screen system is a very simple way to switch scene. It works with the combination of the main scene, scenes inheriting from AbstractScreen and a signal.

The main scene

Main scene: `res://main.tscn` (script)

The main scene contains two components essentials to the screen system:

- A dictionary called `_screens` where you defined all the available screens and give them an ID.
- A function called `change_screen` which switch from one screen to another. You don't call this function directly instead this function is connected automatically to the `next_screen` signal of the loaded screen.

Creating a screen

To create a screen, simply create a scene which inherits `AbstractScreen`, and save it in the screens folder for consistency. To be callable from other screens, add your screen to the `_screens` dictionary of the main scene (see above). Example:

```
var _screens = {
    "menu": preload("res://screens/menu/menu_screen.tscn"),
    "builder": preload("res://screens/builder/builder_screen.tscn"),
    "game": preload("res://screens/game/game_screen.tscn"),
    "board": preload("res://screens/board/board_screen.tscn")
}
```

Switching screen

You can easily switch from one screen to another from script by emitting the `next_screen` signal, passing the ID of the screen to load. Example:

```
func _on_MenuButton_pressed() -> void:
    emit_signal("next_screen", "menu")
```

Pseudo-random number generator (PRNG)

PseudoRng class: `res://addons/cardengine/rng/pseudo_rng.gd` (source)

Introduction

Many games which have randomness in them rely on a seed that is fed to a PRNG, this is the case with most of Rogue-like or Rogue-lite types of games. This give the impression of randomness while being deterministic given the same seed. CardEngine offers a utility class to help with this. Godot also offers a PRNG but you can only have one generator at a time. CardEngine overcomes this limitation and allows for unlimited generators to work in parallel.

API

- `set_string_seed(str_seed: String) -> void`: seeds the PRNG using a string of characters, this helps making sharing seeds more user-friendly.
- `set_seed(random_seed: int) -> void`: direct seeding of the PRNG.
- `generate() -> int`: generates a random integer number and returns it.
- `randomi() -> int`: alias for the generate function for API consistency.
- `randomf() -> float`: generates a random float number between 0.0 and 1.0 (included) and returns it.
- `random_range(from: int, to: int) -> int`: generates a random integer number between from and to (included) and returns it.
- `randomf_range(from: float, to: float) -> float`: generates a random float number between from and to (included) and returns it.
- `random_vec2_range(from: Vector2, to: Vector2) -> Vector2`: generates a random 2D vector between from and to (included) and returns it.