

Documentation

# Core principles

The Valkyrie Engine makes it easy for game makers without programming experience to develop full-fledged retro text games without writing a single line of code. Instead, the engine makes use of JSON files to store the game state. In other words, to develop games within Valkyrie, all you need to do is to create and edit JSON objects using some text editor (we can highly recommend this for windows users: <https://www.microsoft.com/sv-se/store/p/json-editor/9nblggh5plsg>)

In this chapter, core principles of the engine are described in detail. Please read this before reading the section “Get started”, as this chapter provides information crucial to understanding how to create games within the engine.

## Rooms

Valkyrie games are all based around the concept of *rooms.* A room most commonly models a real-world room, but could also be considered a general container for the player. In other words – a room is a game world entity in which the player can be located at any point in time.

A game always consists of *at least one* room. This is where the player spawns – *the spawn room*. This room always have the unique id 0, this cannot be changed. Apart from this, the spawn room can be setup just like any other room (more on how to setup rooms in upcoming chapters)

## Doors

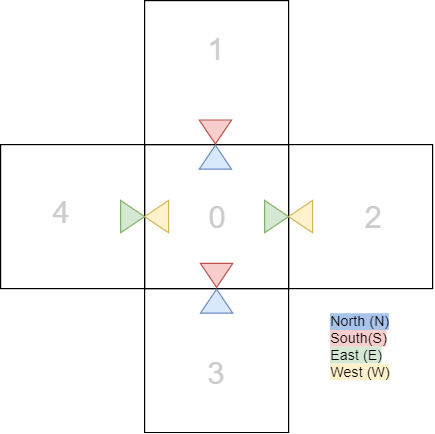
Doors, without surprise, connects two\* rooms. In the demo version of the engine, doors are always both ways. In other words, if the player enters though a door, he or she can always venture back through it and end up at the same place, just as in the real world. However, in upcoming versions this might change to enable some mind bending (and probably very confusing) gameplay.

Doors can be locked by *keys* (one or more). Keys are either already held by the player or picked up by the player in the game. If a door starts as locked, the corresponding key(s) must be placed within the game world, to prevent unsolvable puzzles. The game world setup is validated on initialization to prevent this from happening.

Keys can be placed in *containers* or in the rooms themselves (as rooms are just a special kind of container).

## Orientation and directions

When creating a game with Valkyrie, it’s important to think about the game world layout. It’s highly recommended that a map is drawn first, before adding the rooms configuration. Consider the drawing on the next page, illustrating a simple game world consisting of five rooms, where the middle room is the spawn room.



Each arrow color represents a direction, labeled as in the map legend. Each arrow represents the doors themselves within the game.

When creating the doors, it’s important to label these correctly. If you add the northern most door in room 0, connect it to room 1 with the direction tag **NORTH,** and so forth. The directions can then be used by the player to issue *commands,* for example “open the door to north”. See the section about Input Parsing to read more about the topic.

Of course, if the player stands in any of the other rooms, apart from spawn, there is no need to specify a door by direction, “open the door” would be enough.

It should also be noted that the game supports two more directions not depicted in the map; Up (U) and Down (D).

A closely related topic to door directions is “*player orientation*”. This is a built-in feature that gives the player the ability to turn by 90 degrees within a room. Then, the player can simple use the command “open the door in front of me” or similar, instead of specifying a direction. The game core uses a special kind of mapper, the “orientation mapper”, to map between simple command keywords such as “to the right” or “in front of” to actual game world directions. As with all features that are language based, it must be configured via the Json files.

## Actions

A fundamental aspect of the Valkyrie Engine is the use of *actions.* In short words, an action in the game is a triggered response with the effect that something happens. A good and trivial example is the *DisplayText* action – it simply displays some text to the player.

An action is either *Global* or *Localized*. Global actions can occur at any time, in any room, within the game world. Localized actions however are strongly tied to some *context,* as an example some specific room.

See a complete list of built in actions **here**

## Trigger

A trigger is a game entity that causes an action to be performed. The trigger is often based on some user input. A standardized trigger is the single keyword “help” that when typed displays some customizable help text to the user. The keyword “help” is simply a trigger for the DisplayText action mentioned above.

## Conditions

Most triggers have *conditions* attached to them. If the conditions are met, the action is triggered. If not, an alternate action (FallbackAction) is triggered instead. Fallbacks cannot (as of the demo version of the engine) have attached conditions.

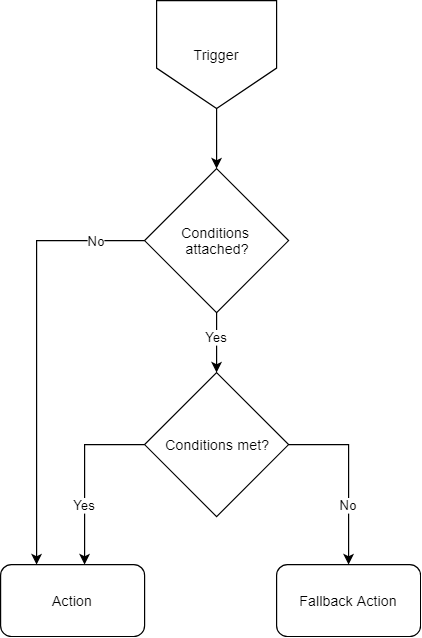
## Events

A game event is the combined process of a trigger, condition and action. An (conditional) event can be described simply by this language invariant function:

function Trigger()

If (conditions are met) then Action() else FallbackAction()

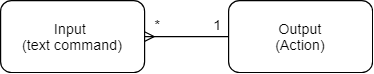
The following flow chart also depicts the whole event, with and without attached conditions



## Input resolving

All input from the user consists of text commands. The Valkyrie Engine uses *input resolvers* to process the user input. The resolver can be used when there is no 1:1 correspondence between the user and the game action (i.e. more complex situations than the built in “help” event)

The resolvers rely heavily on *mappings.* A *mapping* is a game object that maps the user input(s) to an *Action*. To keep it simple, the following relation of the mapping is established:



The mappings in turn relies on *rules*. The Valkyrie Engine consists of several built-in rules to make the creation of mappings an easier task. Consider the following example of how one would create a mapping for letting the user unlock doors.

{

"map" :

{

"comment": "Mapping to resolve the action UnlockDoor with 3",

"id" : 17,

"action" : 19,

"scope" : "LOCALIZED",

"mappings" :

[

{

"input" : ["unlock door", "use key", "use key on door", "unlock door with key", "unlock with key"],

"rule" : "EXACT"

},

{

"input" : ["unlock", "use key"],

"rule" : "FUZZY"

},

{

"input" : ["unlock", "door", "key"],

"rule" : "ALL"

}

]

}

}

This map contains three mappings. All perform the same task (i.e triggers the same Action), but using three different rules:

**EXACT**: Trigger the Action if the user inputs any of the phrases. The user input must contain the whole string exactly as it typed here.

**FUZZY**: Trigger the Action if the user inputs any of the words / phrases listed. For example, in this case the input string

“unlock the elephant with the hacksaw”

Would suffice. However, this is clearly not recommended due to illogical effects.

**ALL**: Must match all keywords / phrases, but the order is not important.

Also note that this example creates a LOCALIZED map – it can only be used in a specific context. If one wants to create a more general solution the following can be used:

{

"map" :

{

"comment": "Mapping to resolve the action UnlockDoor",

"id": 2 ,

"action" : 7,

"scope" : "GLOBAL",

"mappings" :

[

{

"input" : ["unlock", "door"],

"rule" : "ALL"

}

]

}

}

Note that both these examples are contextual, even though the second one is global. They both assume that the door referred to is already known (*context*).

## Contextual input resolving

Consider the following text displayed to the player, triggered by the OnRoomEnter-trigger, when the player enters an unvisited new room in the game

You enter the small, poorly lit chamber. To your left, there is a small, rusty hatch in the wall. To your right there is a great, wooden door. You can always go back, but you feel the way forward is either through the hatch, or through the door. What do you do? (You are currently facing north)

The developer need to design the interactions in a non-ambiguous way. With just the examples above, the game cannot properly resolve which action to take (or, in other words, which door \*(or hatch) to unlock or open). Assume for the following examples that the doors are both unlocked.

First, the *orientation mapper* must be properly configured for the game language. The following example shows a simple mapper configuration that is useful in the above example:

# A complete list of actions and their properties

## DisplayText

Properties

Text (string)

Path (xpath)

## DisplayTextAndQueryPlayer

Displays a text and queries the user for input. Requires an InputParserDescriptor to resolve user input and to produce a resulting Action.

Text (string)

Path (xpath)

InputParserDescriptor (descriptor that resolves the user input)

## GameStart

## GameOver

# A complete list of triggers and their properties