PCA 3D ADVENTURE STUDIO

Text Parser

Typed commands, useful words and responses that make sense



Build a working command, then follow it into play

Learn every part of Text Parser Designer, connect a command-entry box, give objects useful synonyms, and build custom commands with scripts or visual nodes. Follow a complete workshop example from the first Look response to two-object interactions.

Version 0.86.3.4 · Windows, macOS and Linux

No practice project or extra downloads required.

How to use this guide

If you have already followed the scene, character, GUI or verb tutorials, open that project and use a familiar object. You can also use an existing demo. The workshop pictured here is a small example made with the bundled Clockmaker artwork; it is not a project you need to download.

Work through Chapters 1–9 for the basic connection: accepted words → an input box → the object's actions. Continue with custom commands and scripts in Chapters 10–13, then add two-object and inventory interactions in Chapters 14–16.

Chapters 17–20 cover room descriptions, walking, language choices, troubleshooting and a complete test pass. Use the clickable contents or PDF bookmarks when you need one particular field.

The numbered lists are steps to carry out in the editor. Tables explain exact settings or compare expected results. Screenshot close-ups show real controls; the surrounding text supplies any values that are too long to fit inside a field.

Some examples intentionally show a failed match or an unfinished command. Their captions say so. Keep those tests separate from a working game setup.

Apply designer changes, save the project and start a fresh game preview. Command Preview and GUI Designer preview are useful checks, but neither replaces a gameplay test.

You can use your own object names and IDs. Where a script uses an exact ID such as workshop or clock_wound, substitute the corresponding ID from your project.

Screenshots follow version 0.86.3.4. The workflow is shared across desktop platforms. Test touch keyboards and other device-specific input on the devices you plan to support.

Contents

Click a chapter or page number to jump to it. PDF bookmarks also include the smaller section headings.

How to use this guide . . . 2

1. What the text parser does . . . 4

2. Open the designer and understand its controls . . . 5

3. Give an object a useful vocabulary . . . 8

4. Teach the game more verbs . . . 9

5. Remove filler words without losing meaning . . . 11

6. Read Command Preview properly . . . 12

7. Put a command-entry box in the game . . . 14

8. Submit your first command in play . . . 16

9. Make a typed verb change the game . . . 18

10. Add a complete custom command . . . 20

11. Make a Help command with visual nodes . . . 23

12. Make a Help command with GDScript . . . 25

13. Check the room and puzzle state in a custom handler . . . 27

14. Use one object on another . . . 29

15. Include inventory commands . . . 32

16. Talk to characters and refer to regions . . . 33

17. Look around and walk by command . . . 35

18. Write vocabulary and feedback players can use . . . 37

19. Find the cause when a command fails . . . 38

20. Finish with a complete test pass . . . 40

1. What the text parser does

The text parser lets a player type an instruction such as **look at the clock** instead of choosing Look and clicking the clock. It connects typed words to the commands and responses you have already made for your game.

You do not need to write a new script for every sentence. Give Look a few useful synonyms, give the clock a few useful names, and several sentences can lead to the same response.

The player types	The game uses
look at the clock	Look + Workshop Clock
examine the timepiece	Look + Workshop Clock
please inspect the clock!	Look + Workshop Clock
wind up the clock	Wind + Workshop Clock
help	A custom command that calls a script

Words, commands and responses

A **verb** is the action: Look, Talk, Use or Wind. A **target** is the thing receiving that action: the clock, Elias or a brass key. A **synonym** is another accepted word or phrase for the same record. A **response** is what you author in the target's actions.

For an ordinary one-object command, PCA 3D cleans the sentence, finds a verb, finds a target and passes them to the interaction system. That system runs the target's response. The parser chooses the instruction; the actions decide what happens next.

A **custom command** works differently. You enter a complete phrase, such as **help**, and choose the script to run. It does not need an object or a verb record. Custom commands are useful for instructions, status checks and other commands that do not belong to one visible object.

> Start small: one object, one Look response and one working sentence. Add synonyms after that works. Add custom commands and two-object sentences later.

What it will not work out for you

The parser matches the vocabulary you supply. It does not correct misspellings, invent puzzle solutions or work out the meaning of any sentence a player might write. **Inspect** works when you add it as a synonym. **Insepct** is a different word.

It also does not provide a general command queue. Teach players to submit one instruction at a time. A sentence such as **take the key and open the door** is not a promise that both actions will run in order.

The examples in this guide use a small workshop, a stopped clock, a lantern, a brass key and Elias. You can follow them with objects from your own project. Keep your own IDs where they differ from the examples.

2. Open the designer and understand its controls

Open a game project. In the Database panel, select **Text Parser**. You can also use the workspace toolbar button whose tooltip begins **Text Parser**. If the toolbar is crowded, use its horizontal arrows to reach the remaining designers. If the Database panel is hidden, show it from **Panels**.

The designer has one project-wide set of parser settings. You do not create a separate parser record for each room.

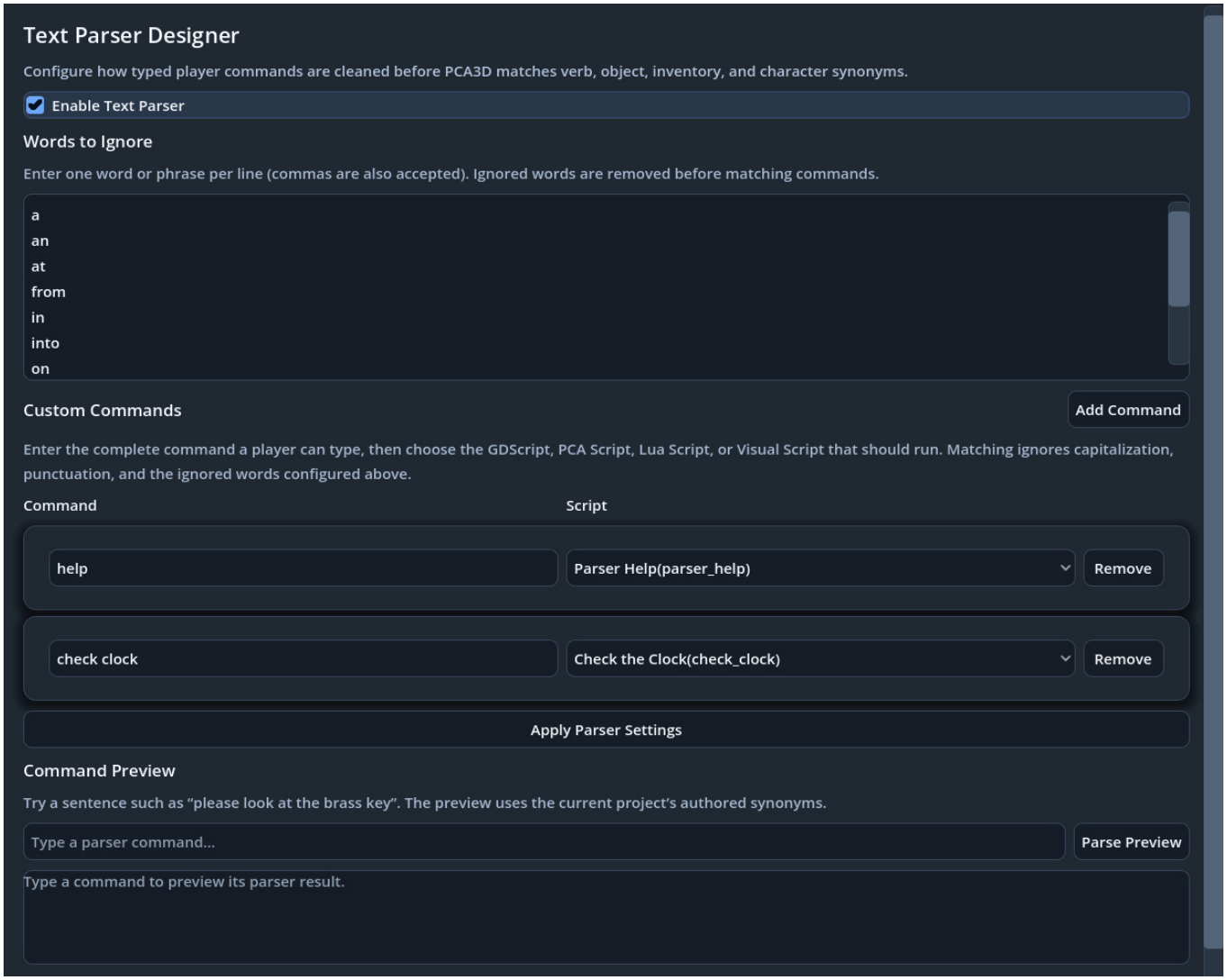

Figure 1. The Text Parser Designer in the running editor. This overview shows the main sections; the following pages use closer views.

Control	What it does
Enable Text Parser	Allows typed commands to run in the game.
Words to Ignore	Removes filler words or phrases before ordinary matching.
Custom Commands	Connects complete command phrases to scripts.
Add Command	Adds another phrase-and-script row.
Command	The complete phrase for that custom row.
Script	The script called when that phrase matches.
Remove	Deletes that custom command row. It does not delete the script.

Control	What it does
Apply Parser Settings	Accepts the current parser fields into the project.
Command Preview	A place to check how a sentence matches.
Parse Preview	Refreshes the result for the text you entered.

Apply, save and restart

1. Make a change to the ignored words, enabled checkbox or custom command rows.
2. Click **Apply Parser Settings**. Look for **Text parser settings applied.**
3. Click **Save Project** in the main toolbar. For a new project, choose its save location.
4. Start a fresh preview with **Test Scene** or **Play / Restart**.

Command Preview reads the current fields in this designer, including edits you have not applied yet. A preview result is therefore not proof that the running game has received those changes. Applying, saving and restarting removes that uncertainty.

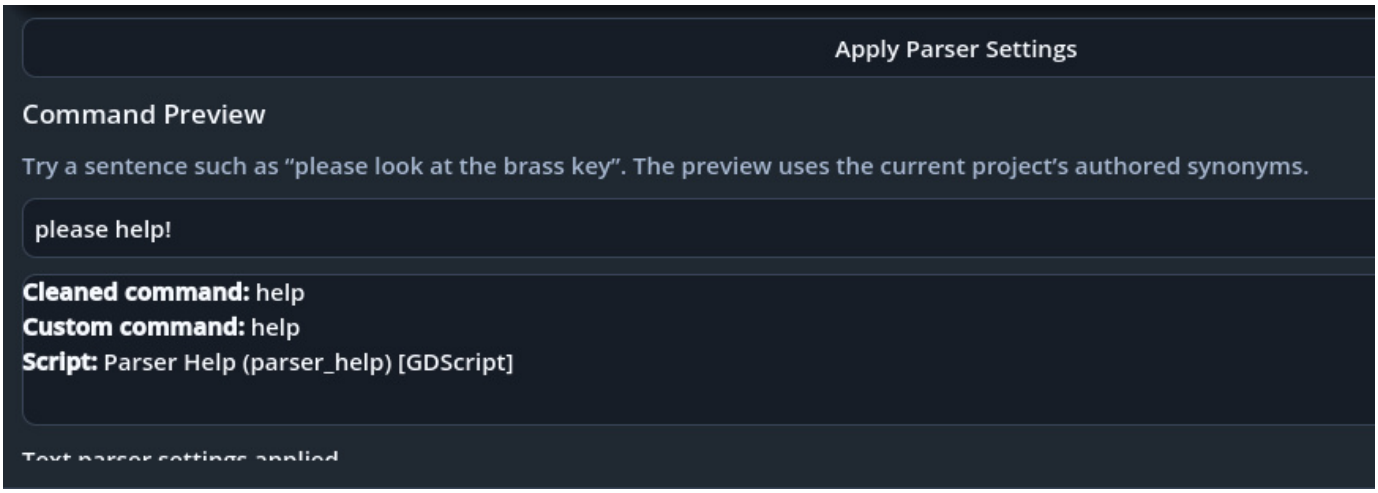

Figure 2. The accepted settings show a confirmation beneath Command Preview. Save Project is still a separate step.

Turning typed input off

Clear **Enable Text Parser**, apply the settings and test again. A submitted command is refused with **The text parser is disabled.** This checkbox controls execution. It does not remove a text box from your GUI or stop Command Preview from analysing words.

If the game should no longer offer typed input, also hide or remove the command-entry element in your HUD. Keep mouse controls available if the player still needs them.

3. Give an object a useful vocabulary

Begin with an object that is already visible in a working scene. The screenshots use the grandfather clock model from the bundled Clockmaker assets. Its scene record is named **Workshop Clock**, with the ID **clock**.

1. Open **Scene Designer** and select the object in the room or Scene Objects list.
2. Open the Inspector's **General** tab.
3. Keep a clear **Name**. This is the name people will recognise in the editor and game.
4. Find **Synonyms**. Enter **clock, timepiece, grandfather clock**.
5. Click **Apply Changes**, then **Save Project**.

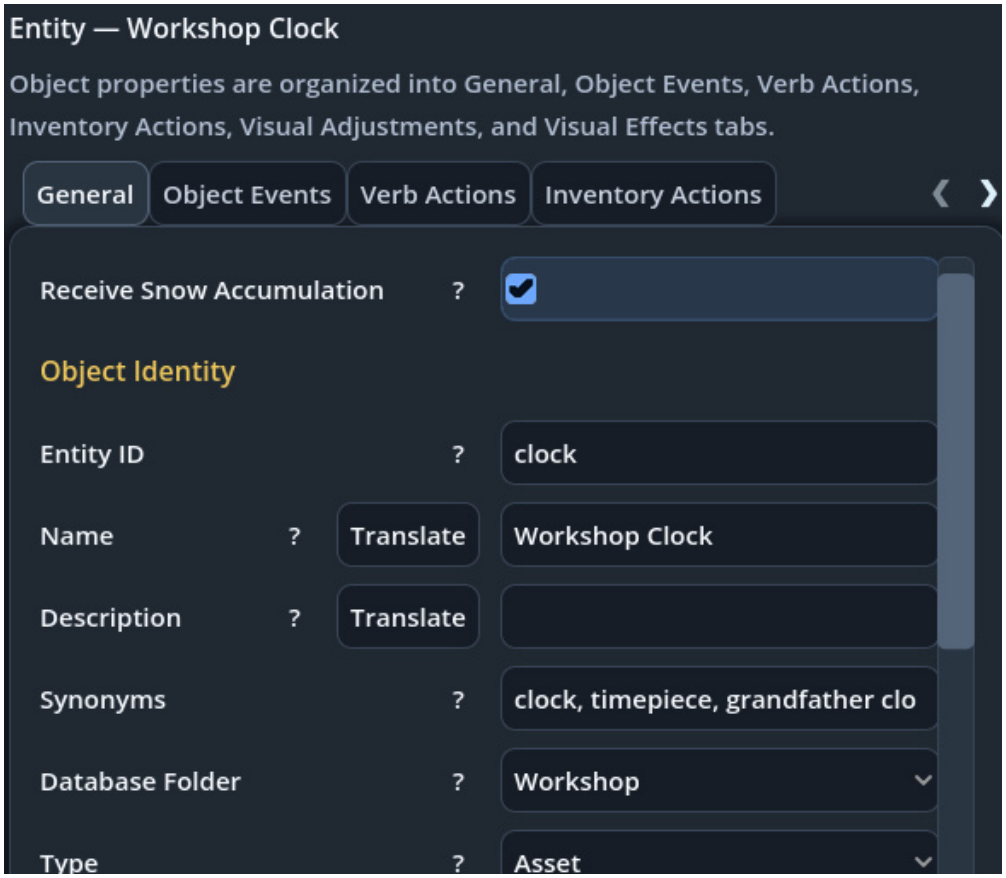

Figure 3. The scene object's identity and Synonyms field. These belong to the placed clock, rather than its imported model asset.

The record ID and name are also candidates for matching. Synonyms let you accept sensible alternatives without renaming the object. A player can use **timepiece** even though the editor still calls it Workshop Clock.

Add target words to the **scene object**, rather than expecting the imported model's asset name to supply them. An asset is the source artwork; the placed object owns this room's identity and interactions.

Make the clock answer Look

1. With the clock still selected, open **Verb Actions**.
2. Find the **Look Verb** group.
3. Set its action to **Show Message**. Enter **The workshop clock has stopped.**
4. Give the message enough time to read. The captured example uses seven seconds.
5. Apply the changes and save.

The same object can have a Description as well. Use Description for a simple fallback, but author a Look response when you want a particular message or more than one action. Do not confuse an object's event volume or collider with its parser vocabulary.

> Keep the first response obvious. A message is easier to check than a subtle sound, variable change or lighting adjustment. Once the sentence reaches the correct object, you can extend its actions.

4. Teach the game more verbs

The target's synonyms name the thing. Verb synonyms name the action. **Timepiece** belongs on the clock; **examine** belongs on Look.

1. Open **Verb Command Designer** and select **Look(look)**.
2. Keep **Verb ID** as **look** and **Display Name** as **Look**.
3. In **Synonyms (comma-separated)**, enter **examine, inspect, observe**.
4. Leave **Enabled** checked.
5. Click **Apply Changes** and save the project.

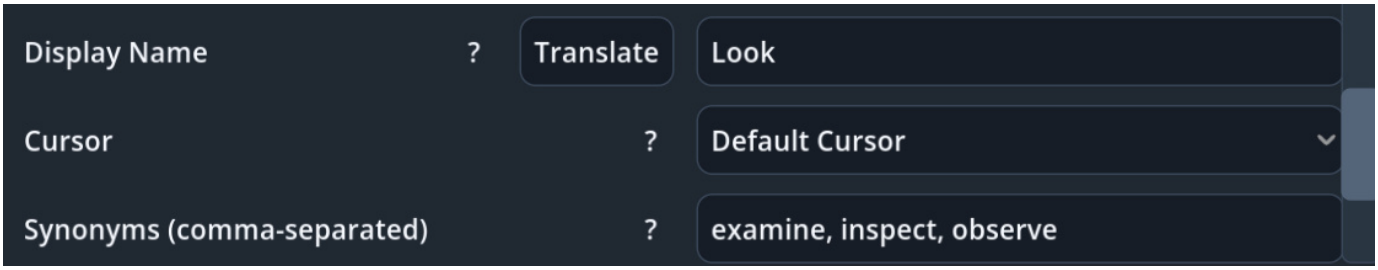

Figure 4. Look accepts examine, inspect and observe. The action remains the same Look command.

Return to Text Parser and try **please examine the timepiece**. The expected result is Look and Workshop Clock. You have changed the accepted wording without making a second Look response.

Add a new Wind verb

1. In Verb Command Designer, click **Add New**.
2. Give it the ID **wind** and display name **Wind**.
3. Add **wind up, crank** as synonyms.
4. Leave **Enabled** checked. A preposition is not needed for this one-object example.
5. Apply and save. You will add Wind's response to the clock in Chapter 9.

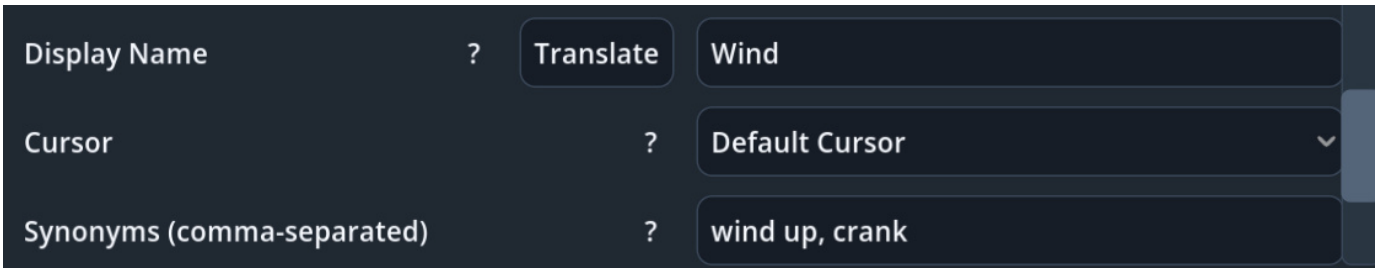

Figure 5. Wind is a separate command. Its multiword synonym wind up reaches the same Wind response.

Use a stable ID once other records refer to it. Changing an ID is more than changing the wording on a button. To improve what players read or type, edit the display name and synonyms first.

Multiword aliases and overlap

An alias can contain more than one word. Matching looks for those words together, in order. **Grandfather clock** is a two-word alias; **wind up** is another. A longer matching alias takes priority over a shorter one. If equal-length matches belong to different records, the first encountered record wins; the game does not ask the player which one they meant.

Avoid giving two objects the same vague alias, such as **door**, when both can be targets in the room. Prefer **archive door** and **street door**, and test the short wording separately. Do the same with verbs: do not assign **inspect** to both Look and Use.

Ignored words can break a multiword alias. If **at** is ignored, **look at** becomes **look** in the cleaned sentence. Use the core verb **look**, and let the ignored word disappear. Check any phrase whose meaningful words overlap the ignore list.

5. Remove filler words without losing meaning

Open Text Parser and find **Words to Ignore**. Enter one word or phrase on each line. Commas are also accepted as separators. Entries are normalised to lowercase and duplicates are removed when the list is read.

The usual English starting list is **a, an, the, to, at, with, on, in, into, from, please**. Your game can use a different list. These are project settings, not a fixed rule of English.

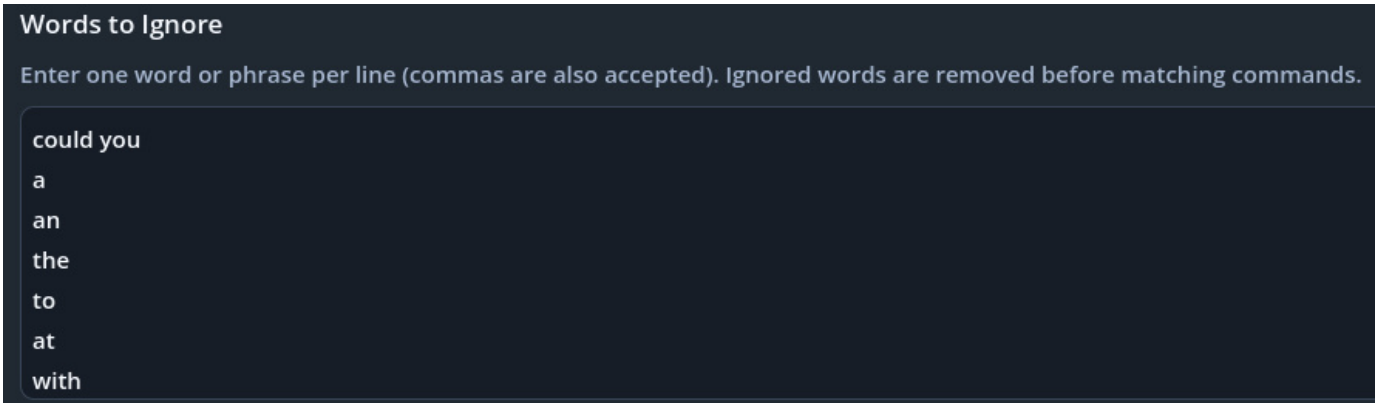

Figure 6. An added phrase, could you, sits in the same list as single ignored words.

Try an ignored phrase

1. Add **could you** on its own line.
2. Enter **could you please examine the timepiece!** in Command Preview.
3. Check that **Cleaned command** reads **examine timepiece**.
4. Apply the parser settings, save and restart the game before trying the sentence in play.

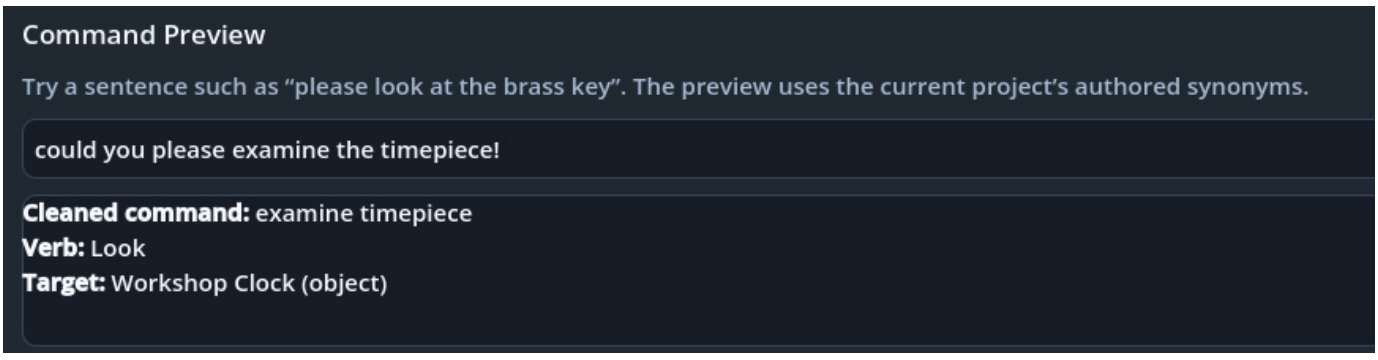

Figure 7. The sentence loses the filler phrase and still finds Look and Workshop Clock.

Whole words and complete phrases are removed. Ignoring **in** does not remove the letters in **wind**. Longer ignored phrases are tried before shorter ones, which helps when phrases overlap.

Choose the list carefully

Do not ignore the words that distinguish your puzzle objects. If you remove **red**, **look red key** can collapse to **look key**. If you remove **up**, your **wind up** alias loses one of its words. Keep meaningful adjectives, directions and action words unless you have tested the consequence.

An ignored word is removed wherever it matches. It is not restricted to the start of a sentence. Changing the list can affect ordinary commands and custom phrases throughout the project, so retest both.

Capitals and punctuation

Matching is case-insensitive. Common punctuation such as commas, exclamation marks and question marks is separated out; periods at the edges of words are trimmed. **LOOK AT THE CLOCK!** can therefore reach the same response as **look clock**.

Do not assume every symbol is discarded. Hyphens, apostrophes in names and unusual input deserve their own test. An apostrophe is treated as a separator, so a possessive name may split into more words than you expect. Add a simple alternative such as **clockmaker** if the full name is awkward to type.

6. Read Command Preview properly

Command Preview is the quickest way to find a vocabulary mistake. Type a sentence and read all three result lines. The result updates while you edit; **Parse Preview** also refreshes it.

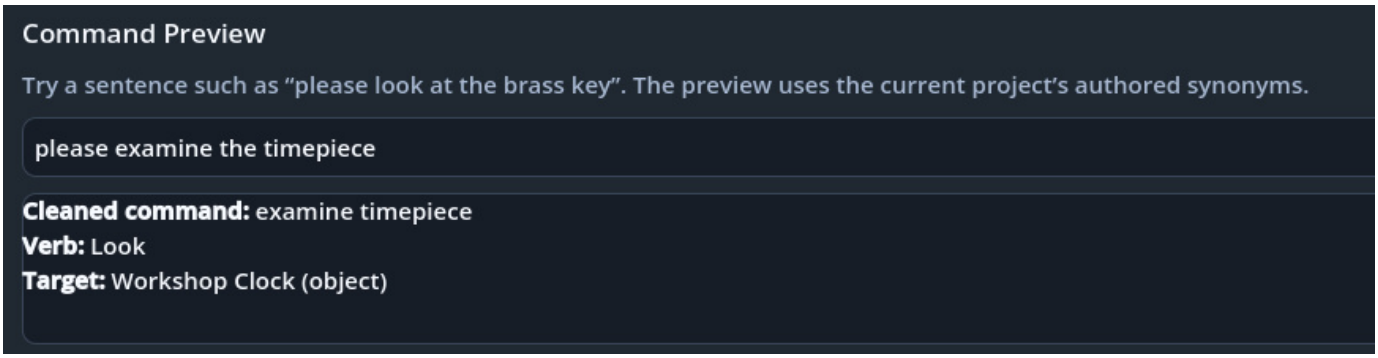

Figure 8. An ordinary command preview shows the cleaned words, recognised verb and recognised target.

Result	What to check
Cleaned command	Are the meaningful words still there after cleanup?
Verb	Did the intended verb win, rather than another matching alias?
Target	Did the intended object, character or inventory record win?

No verb matched

Polish the clock can find the clock while failing to find a verb. Either add Polish as a proper command with a response, or decide that the game should not accept that instruction. Adding **polish** to the clock's synonyms would name the object; it would not create a Polish action.

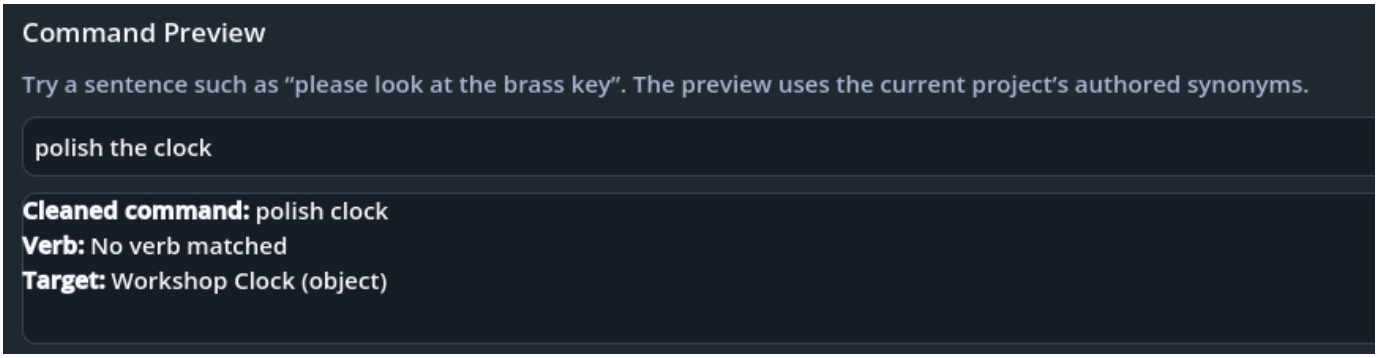

Figure 9. The clock is recognised, but polish has no matching verb in this example.

No target matched

Look unicorn recognises Look but finds no target. Check the target's Name, ID and Synonyms. Check that you are editing the intended record and have applied the changes. If an object is actually called **archive door**, the single word **archive** is not automatically its alias.

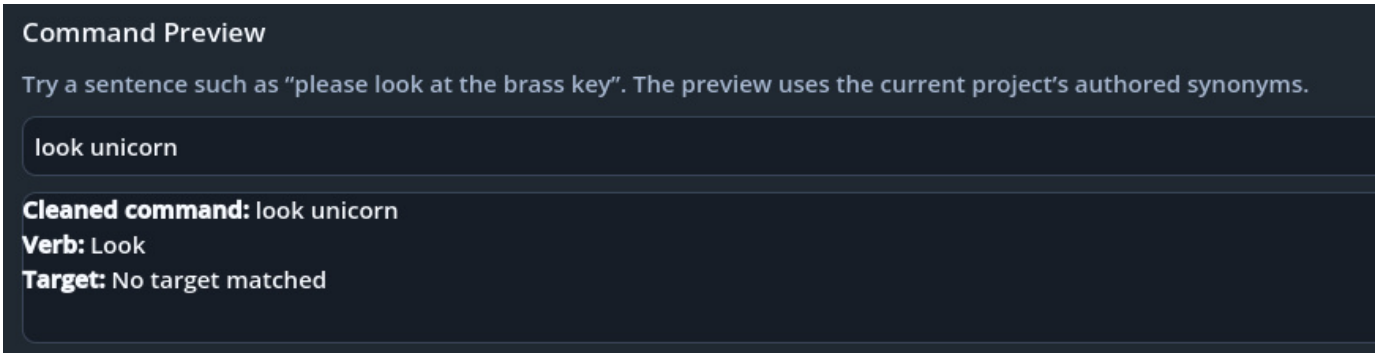

Figure 10. Look is recognised, while unicorn is not a target in the example.

A preview match is only a lookup result

Preview does not walk the player, run actions, add inventory or call the selected script. It also does not prove that the player can reach an object or that a puzzle condition is true.

This release's designer searches authored scene entities across the project and inventory definitions. Its ordinary preview is broader than the current game's room. It can also show an authored verb that is disabled for runtime use. Separate room regions and the two-object execution route are not fully represented by this simple three-line display.

The running game uses the current room's targets and its own interaction rules. Always finish with a game test, especially for objects in another room, disabled records, inventory and preposition commands.

Extra words are another reason to test. Ordinary verb-and-target lookup can find aliases inside a longer sentence without rejecting every unmatched word. A promising preview does not mean the parser understood the whole sentence as a grammatical instruction.

7. Put a command-entry box in the game

The Text Parser Designer does not create a player-facing input box. Add one to a GUI that is visible during gameplay. A main HUD is a convenient place to put it.

1. Open **GUI Designer** and select your gameplay HUD. The example calls it **Parser HUD**.
2. Click **Add Element** and choose a **Text Box**.
3. Give the element a clear name, such as **Command Entry**.
4. Position it inside the GUI's visible area. Make it wide enough for a normal sentence.
5. Select the element and open **Type Settings** in its properties.



Figure 11. The command-entry element is selected in GUI Designer. Its parser options are in Type Settings.

Set the text-box options

Field	Suggested starting value	Why
Placeholder	Type a command, then press Enter…	Explains the control before anything is typed.
Multiline	Off	Uses a single command line with Enter submission.
Clear Single-Line Text When Enter Is Pressed	On	Removes the submitted sentence, ready for the next one.
Run Text Parser Command When Enter Is Pressed	On	Sends the submitted sentence to the parser.
Return Focus To Text Box After Enter	On	Lets the player keep typing without clicking the box again.
Password / Cipher Mode	Off	Keeps ordinary commands readable.

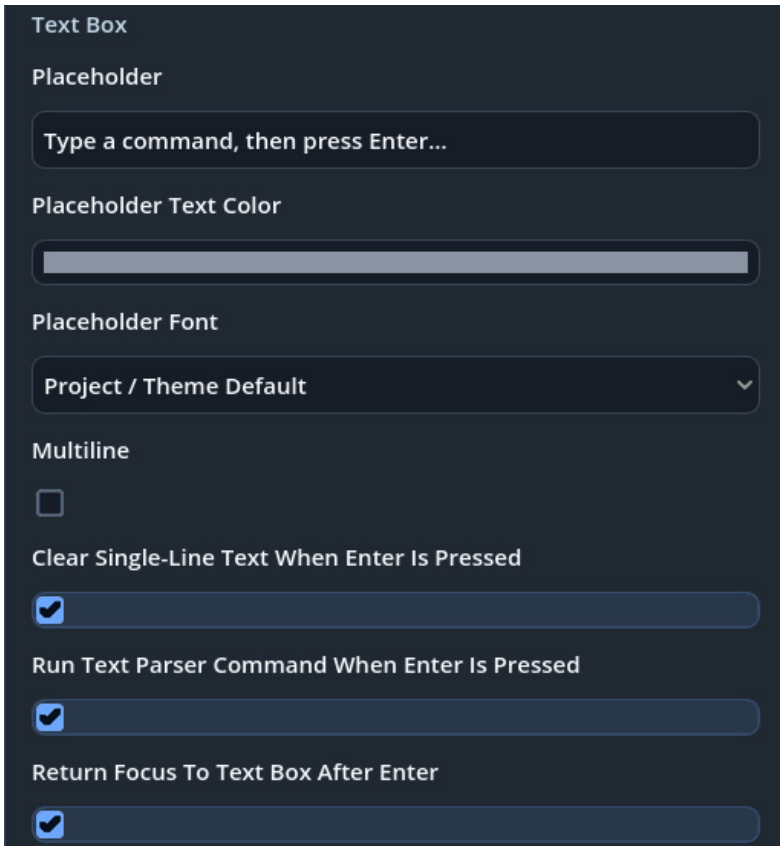

Figure 12. These are the three Enter options used by the example. Multiline is off.

Leave the element **Visible** and **Enabled**. Set readable text and background colours in its style properties. A dark input field against a dark scene still needs a visible edge and clear text.

GUI element property changes are edited live in the designer. Save the project after setting them. You do not need to add a Click event or write a submission script for the built-in Enter option.

Make sure the HUD is shown

If you created a new HUD, assign it as the project's **Main HUD** in the system-GUI settings. Otherwise, put the text box in the gameplay HUD your project already uses, or show that GUI through your existing game setup.

The GUI must be visible during play, not just open in GUI Designer. Check that a full-screen menu, cutscene overlay or another GUI is not covering it. Use the GUI's own design size and anchors so the command line stays on screen at your chosen game resolution.

The screenshot layout uses a 1280 × 720 game design size. The exact coordinates are not required. Put your input where it can be read and clicked without covering important scene targets.

8. Submit your first command in play

Save the project and use **Test Scene** for the room containing the clock. Test Scene is useful for checking one room without replaying the introduction. Later, use **Play / Restart** to check the normal startup path too.

1. Click the command-entry box.
2. Type **please examine the timepiece**.
3. Press **Enter**.
4. Confirm that **The workshop clock has stopped.** appears.
5. Check that the submitted text has cleared and the box is ready for another command.



Figure 13. The sentence is entered in the actual game preview, before submission.



Figure 14. After submission, the clock's Look response appears and the entry line is clear.

If the line clears but no action happens, clearing alone is not proof that parsing was enabled. Check both the parser checkbox and **Run Text Parser Command When Enter Is Pressed**, then inspect

the sentence in Command Preview.

Designer preview and gameplay are different

GUI Designer's preview is for checking presentation and control behaviour. It does not run the game events that make a parser command work. Use the actual running game to check submission, actions, inventory and room state.

The command label is optional

A label containing **{command}** can show the current interaction command. **{verb}** and **{target}** are also available for a label built around those parts. They use braces, as shown here.

These labels describe the current command and target selection. They are not a transcript of everything the player typed, and they are not the place where you enter parser synonyms. Keep an input box for typing and use messages, dialogue or your own GUI for responses.

Try submitting two commands in a row. Also click elsewhere, then click back into the box. Keeping focus after submission helps keyboard users; it does not guarantee the box keeps focus after every other GUI operation.

9. Make a typed verb change the game

Now give Wind a result that does more than display text. You need the Wind verb from Chapter 4, the clock target and a Boolean variable called **clock_wound**.

1. In **Variable Designer**, add a Boolean variable with ID **clock_wound** and default value **False**.
2. Return to Scene Designer and select Workshop Clock.
3. Open **Verb Actions** and find **Wind Verb**.
4. Make the first action **Set Variable**. Select **clock_wound** and set its value to **True**.
5. Add the next action as **Show Message**: **You wind the clock. A steady tick returns.**
6. Apply the object changes, save and start a fresh preview.
7. Type **wind up the clock** and press Enter.

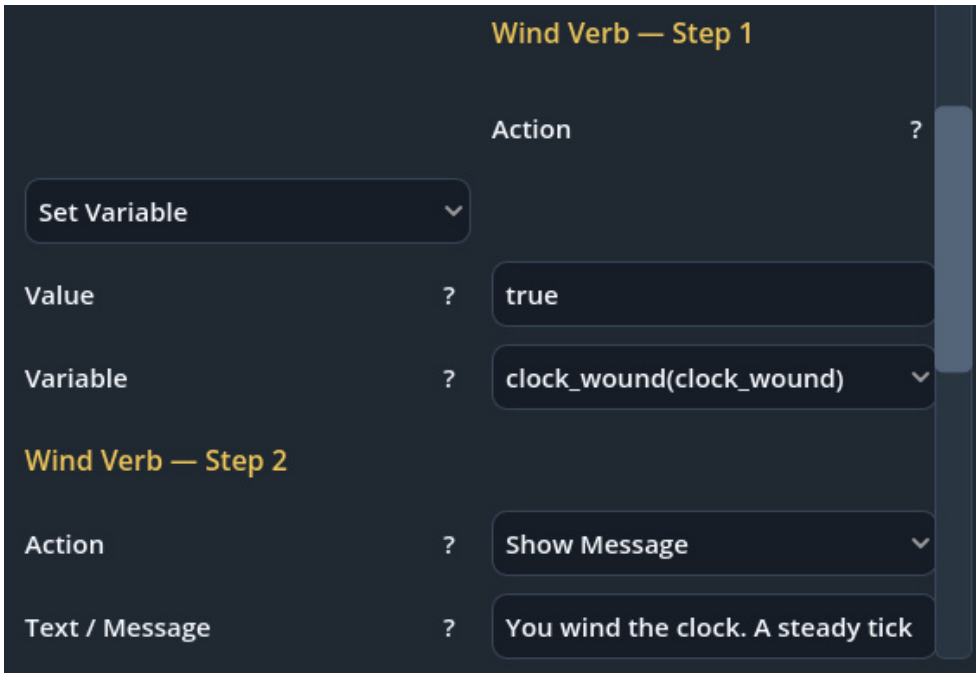

Figure 15. The clock's Wind response sets clock_wound, then shows the result. Scroll within Verb Actions to reach the Wind group.



Figure 16. The typed Wind command reaches the authored response in the game.

The screenshot's message describes the result. It does not itself start a clock animation or play ticking audio. Add those actions if you want the sound and movement to happen. A line of text and a sound effect are separate parts of the response.

Share the same response with mouse controls

The parser does not need a private copy of the clock's Wind actions. A GUI button with the action **verb:wind**, followed by the clock in **Verb Single Click** mode, can reach the same verb response.

This is a useful check: if a normal Wind interaction works and typed Wind fails, investigate the typed vocabulary and input connection. If both fail, inspect the target's actions and interaction settings.

The walkthrough uses **Verb Single Click** for this straightforward comparison. Context, Direct and Verb Direct have different mouse and inventory selection rules. Typed commands share interaction code, so test the mode your finished game will actually use.

Stop a one-time action from repeating

Setting a Boolean to True a second time does not automatically create a different response. If the clock should only be wound once, add an **If Variable** check before the actions. When clock_wound is False, wind it and set the variable. When it is True, show **The clock is already running.**

Use the same condition for mouse and typed interactions by keeping it in the clock's response. That way the puzzle behaves consistently whichever input the player chooses.

10. Add a complete custom command

A custom command connects a complete phrase to a script. The example **help** gives instructions without asking the player to name an object.

First create the script using Chapter 11 or 12. Then connect it:

1. Open **Text Parser**.
2. Under **Custom Commands**, click **Add Command**.
3. In **Command**, type **help**.
4. In **Script**, select **Parser Help(parser_help)**, or your equivalent script.
5. Click **Apply Parser Settings**, save and restart the preview.
6. Type **help** in the game.

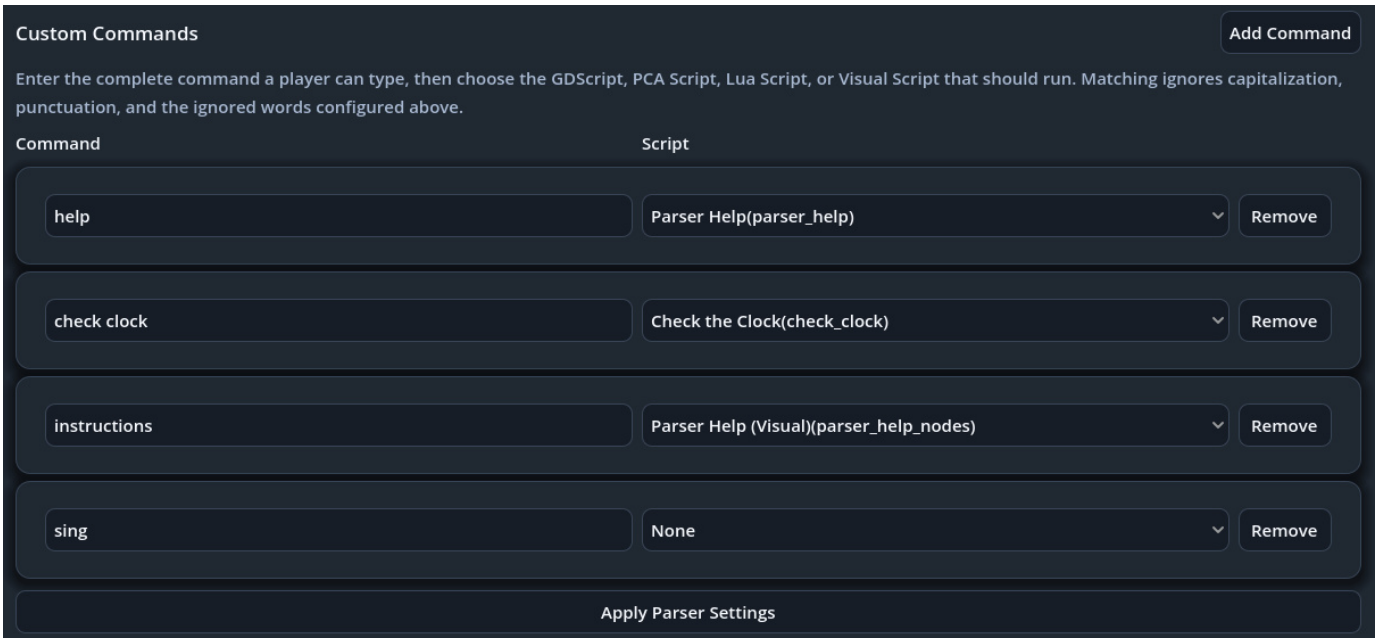

Figure 17. Custom rows pair a complete phrase with a script. A row with None selected still needs a handler.

The selector can offer GDScript, PCA Script, Lua, BASIC and Visual Script records from the project. Choose the actual record you made; the label includes its name and ID, with a language suffix where shown. Give scripts different IDs across the language collections so the selector and runtime do not have to resolve duplicate IDs.

Custom matching is exact after cleanup

The submitted sentence and the authored custom phrase are both cleaned before comparison. With **please** ignored, **please help!** can match **help**. The words that remain must be the same, in the same order.

Authored phrase	Player input	Custom match?
help	HELP!	Yes
help	please help	Yes, when please is ignored
help	help clock	No; an extra word remains
check clock	check the clock	Yes, when the is ignored
check clock	clock check	No; the order differs

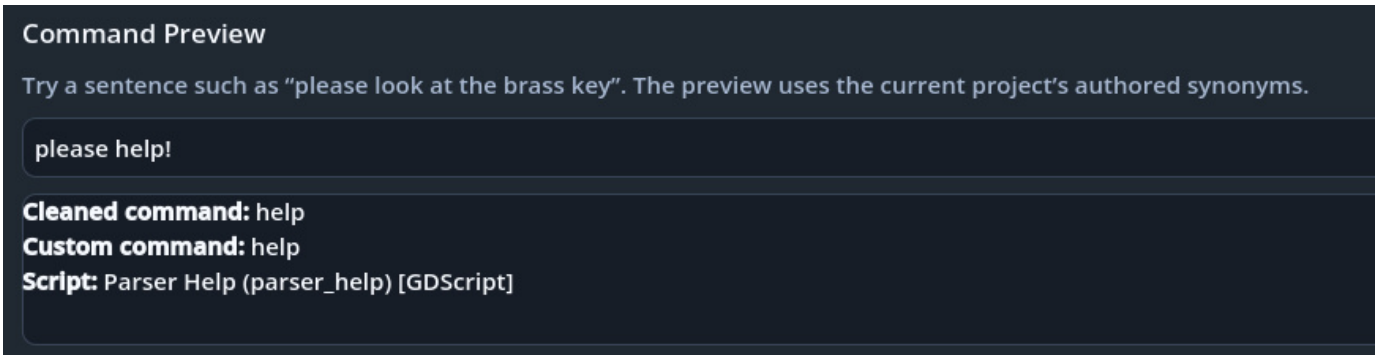

Figure 18. The preview identifies a custom command and its script instead of giving an ordinary verb-and-target result.

Order matters when commands overlap

Custom commands are checked before ordinary verb matching and before two-object commands. A custom **look clock** would therefore take ownership of that exact cleaned sentence. Use a custom phrase only when you want that override.

Avoid two rows that become identical after cleanup. **Help** and **please help** are duplicates when please is ignored. The first matching row wins; a later row does not add another action to it.

Remove, None and missing scripts

Remove deletes the row from the parser configuration; apply and save afterwards. It leaves the script record available elsewhere in the project.

None means no script is assigned. A named command left that way produces **That custom command has no Script assigned.** in play. A **Missing Script** label means the stored ID no longer resolves to an available choice. Choose a valid script or remove the command.

The project validator can catch an unfinished row before the game preview starts. In the screenshot, the test phrase **sing** has no script. Assign its handler or remove the row, apply the parser settings and test again.

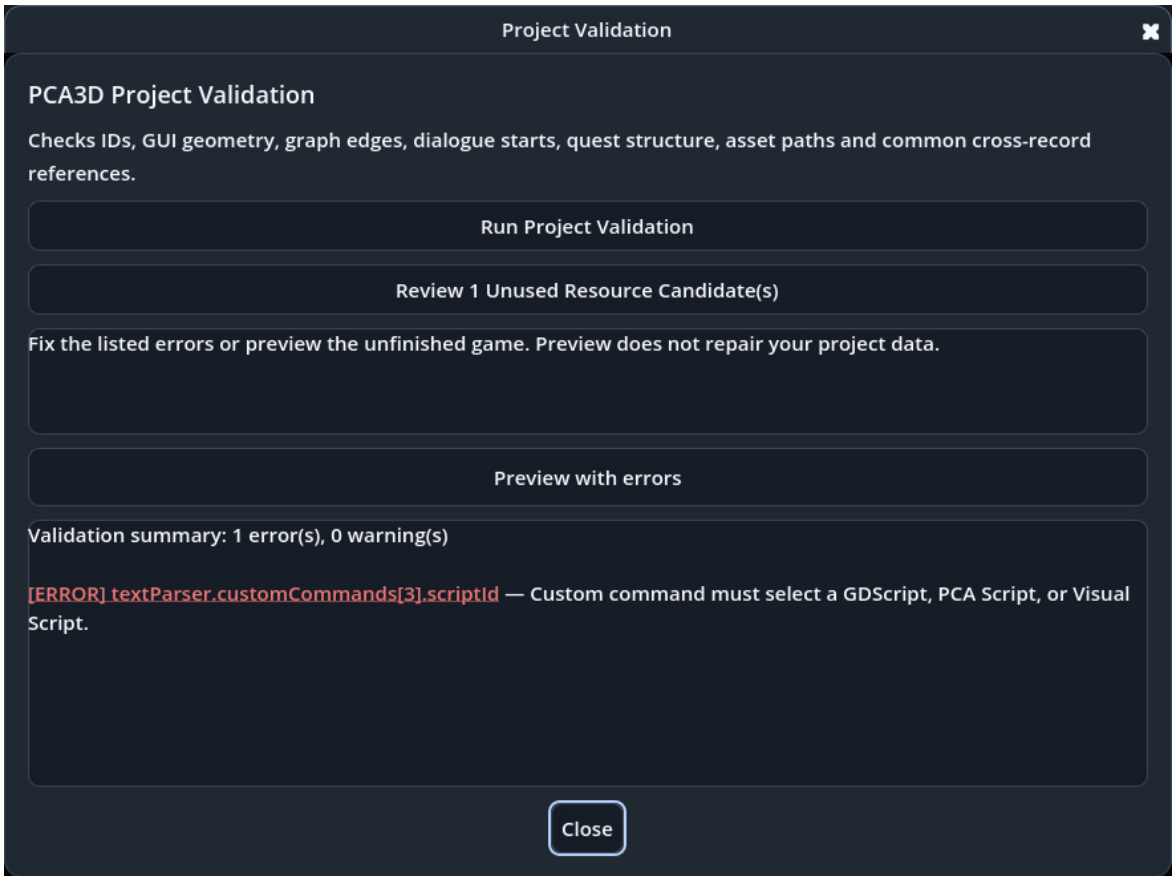

Figure 19. Project Validation catches a custom command with no assigned script. This is an intentionally incomplete row, not a working command.

11. Make a Help command with visual nodes

Use a Visual Script when you want to assemble the response without writing source code. The example remembers that Help was used, then shows a short instruction.

1. Add a Boolean variable **help_used**, with default value False, if you want to record whether help was requested. You can omit this variable and its node for a simpler message-only script.
2. Open **Visual Script Designer** and click **Add New**.
3. Name the script **Parser Help (Visual)** and give it an ID such as **parser_help_nodes**.
4. Add an **On Interact** entry node if the graph does not already have an entry.
5. Add **Set Variable**. Select help_used and choose True.
6. Add **Show Message**. Enter **Try examine the timepiece or wind the clock.**
7. Connect the entry's Next output to Set Variable's Flow input, then Set Variable's Next output to Show Message's Flow input.

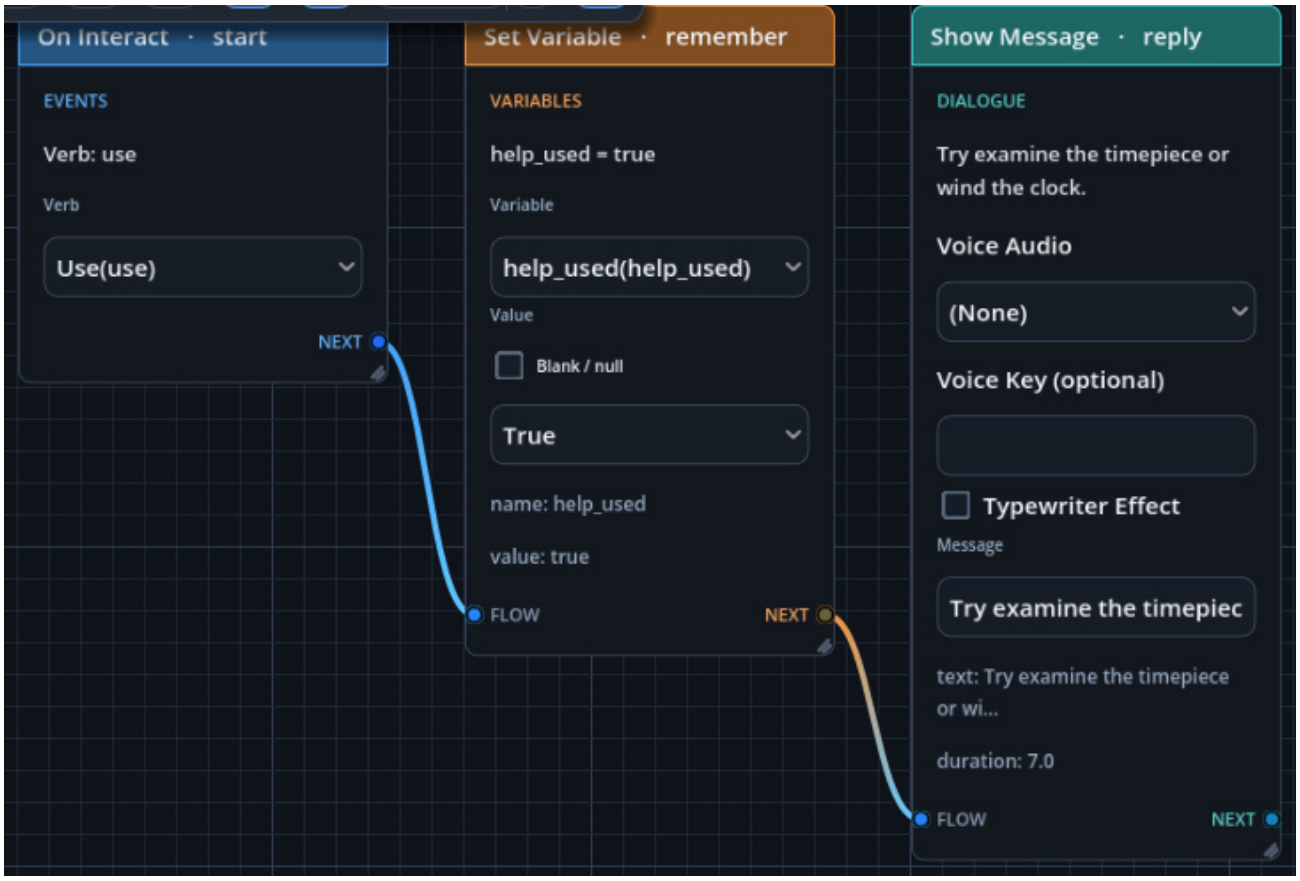

Figure 20. A working visual handler. Follow the connected flow from the entry to Set Variable and Show Message.

The parser calls the selected graph directly. You are not waiting for the player to click a scene object to start it. The entry node provides a start for the graph; its displayed verb does not make **instructions** an ordinary object command.

To keep both examples in one project, give this graph the custom phrase **instructions** and leave **help** connected to the GDScript example. Alternatively, select the visual handler for Help and use only that version. Do not add duplicate Help rows expecting both scripts to run.

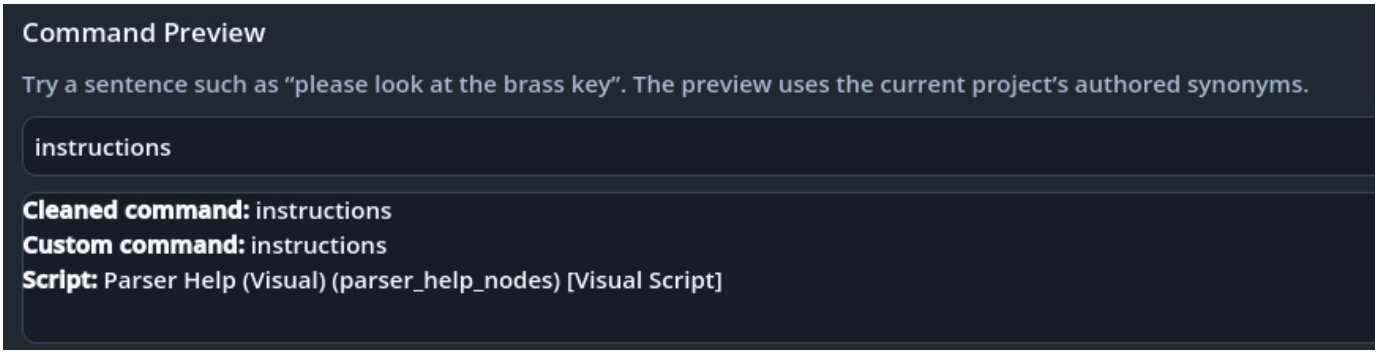

Figure 21. The instructions phrase is connected to the visual handler and is identified correctly in Command Preview.

Save the project and test the phrase in the game. You should see the instruction. The example graph was also checked to set help_used to True.

Add useful conditions

For a room-specific hint, put **If Current Room** before the message. Send its True and False outputs to suitable responses. For a puzzle-specific hint, use **If Variable** and read clock_wound.

A help command is available project-wide unless your own script restricts it. A room test belongs in the handler; selecting a script in Text Parser does not bind that custom command to the room currently open in the editor.

12. Make a Help command with GDScript

You can use a normal PCA 3D GDScript record as the handler. This version does the same job as the visual example. Choose one method to begin with; you do not need both.

1. Open **GDScript Designer** and add a new script.
2. Set the ID to **parser_help** and the name to **Parser Help**.
3. Use the source below. Keep the base-class line if you replace the starter source.
4. Click **Validate**, then **Apply Changes**, and save the project.
5. Select this script for the custom **help** row in Text Parser.

```
extends "res://scripts/runtime/script_base.gd"

func run(_command_context: Dictionary = {}) -> Variant:
    await SetVariable({"name": "help_used", "value": true})
    await ShowMessage({
        "text": "Try examine the timepiece or wind the clock.",
        "duration": 7.0
    })
    return true
```

Create help_used before using that version, or remove the SetVariable line if you only want a message. The leading underscore on _command_context tells GDScript that this example accepts the context without needing to read it.

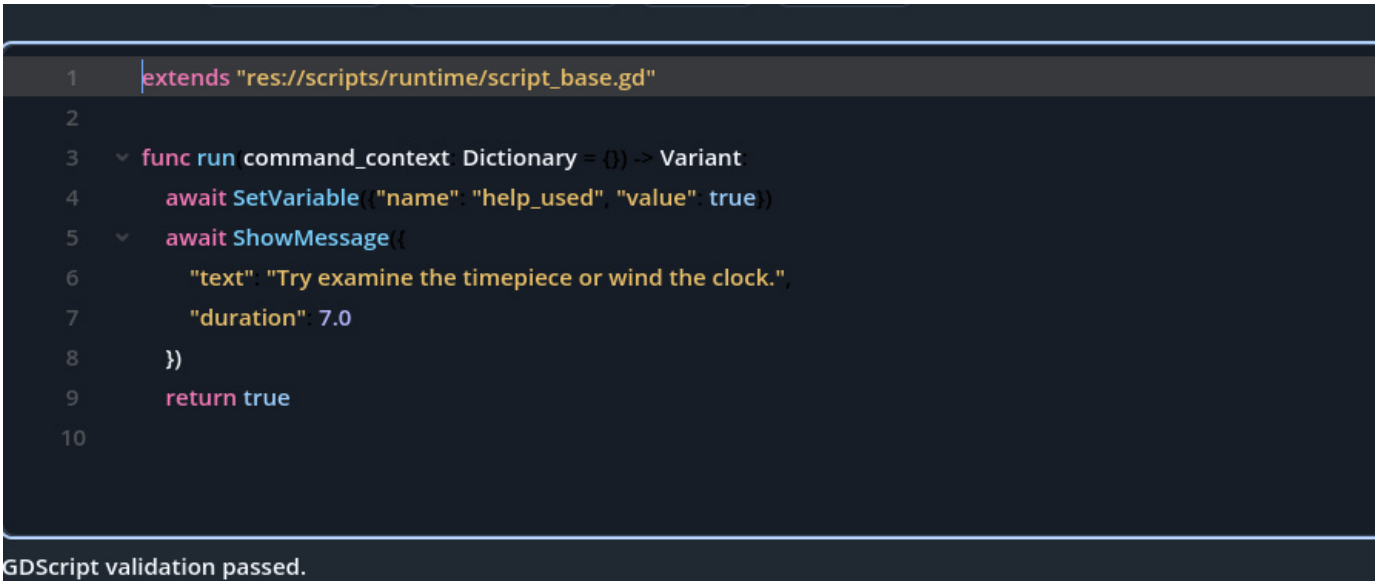

Figure 22. The Help handler in the engine's GDScript editor. The printed source above is provided at a more comfortable reading size.

The base class supplies **ShowMessage**, **SetVariable**, **IfVariable** and access to **Runtime**. These calls run normal game actions. The message text is a response authored by you; the parser does not write it.

What the parser passes to the script

A custom handler receives a context dictionary. If you want to read it, call the argument **command_context** and access its keys with get(). These four fields are supplied for a custom parser call:

Key	Example value	Use
source	text-parser	Identify where this call came from.
command	please help!	Read the original submitted text.
cleanedCommand	help	Read the cleaned command words.
customCommand	help	Identify the authored row that matched.

For example, **command_context.get("command", "")** reads the original string safely. These fields describe the custom-command call. They are not an automatically resolved object ID or a list of arbitrary parameters.

Full phrases are not parameter patterns

A custom row named **give [item]** is not a built-in wildcard that accepts any item. It is a phrase to compare. For object-based commands, use verbs, target synonyms and preposition interactions. For a more specialised grammar, read the original command in your own script and deliberately implement the extra rules you need.

Likewise, a whole-command handler is not automatically owned by the clock. Do not assume **this** means the noun the player typed. Use an explicit target ID or the context appropriate to the action that called the script.

13. Check the room and puzzle state in a custom handler

The **check clock** example reports whether the clock has been wound. It also refuses the command outside the workshop. This makes a useful distinction: matching a phrase and allowing its effect are separate decisions.

Create a GDScript record called **Check the Clock**, with ID **check_clock**. Use the following source, substituting your actual scene ID for workshop if necessary.

```
extends "res://scripts/runtime/script_base.gd"

func run(_command_context: Dictionary = {}) -> Variant:
    if Runtime.scene_id != "workshop":
        await ShowMessage({
            "text": "Try that in the workshop.",
            "duration": 7.0
        })
        return false

    var reply: String = "The clock is still silent."
    if IfVariable("clock_wound", true):
        reply = "The clock is ticking again."
    await ShowMessage({"text": reply, "duration": 7.0})
    return true
```

Validate, apply and save the script. In Text Parser, add **check clock** and choose **Check the Clock(check_clock)**. Apply and save those settings too.

Test both sides of the condition

1. Start a fresh game with clock_wound defaulting to False.
2. Type **check clock**. Expect **The clock is still silent.**
3. Type **wind the clock**. Expect the Wind response.
4. Type **check clock** again. Expect **The clock is ticking again.**
5. In a project with another room, try the command there and check the refusal message.

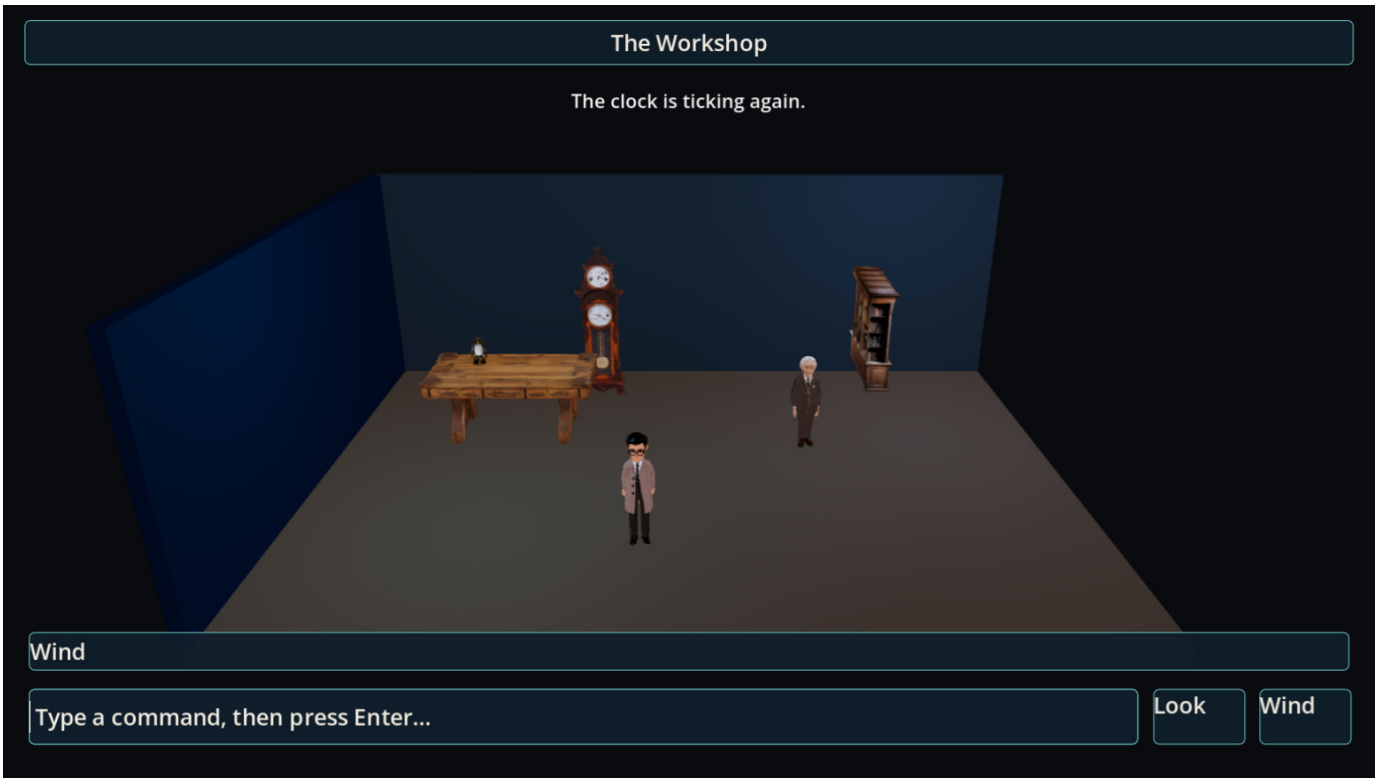

Figure 23. The same custom phrase gives a different answer after Wind has changed the variable.

The workshop state change and both clock replies were checked in the running example. The other-room branch is included for your own multi-room test; the screenshot example contains one room.

Calling the parser from GDScript

Code using PCA 3D's script base can deliberately submit a sentence through the runtime:

```
await Runtime.run_text_parser_command("look clock")
```

That submits a command; it does not return a promise that the puzzle succeeded. The normal matching and interaction rules still apply. Use it when you specifically want to reuse typed-command routing. For a precise internal game action, calling the intended action or script directly is usually

clearer.

Do not make the handler for **help** submit **help** again. That calls the same custom command recursively. Show the help response directly instead.

14. Use one object on another

For a sentence such as **use the lantern on the clock**, the game must identify two targets and know which response belongs to that pairing. This uses the verb's preposition settings and the first object's second-target responses.

There are three authoring steps: allow the preposition on Use, opt the lantern into that behaviour, and give the lantern a response for the clock.

Step 1: allow Use … on …

1. Open Verb Command Designer and select **Use(use)**.
2. Check **Allow Preposition**.
3. Enter **on** in **Preposition**.
4. Keep Use enabled. Apply and save.



Figure 24. Use allows on as a separator between the first and second targets.

Step 2: opt the first object in

Select the placed lantern in Scene Designer. Give it useful synonyms such as **lantern, lamp**. In **Verb Actions**, enable **Use Preposition · Use on**. This checkbox belongs to the first object in the command.

The order is **Use [lantern] on [clock]**. The lantern is the source; the clock is the second target. Swapping their order is a different interaction and needs its own suitable setup.

Step 3: author the pair's response

1. Still on the lantern's Verb Actions tab, scroll to **Second-Target Responses**.
2. Click **Add Second-Target Response**.
3. Select Use as the response's **Verb**.
4. Under **Allowed Targets**, select the scene clock. The stored reference in this example is **entity:clock**. The entity prefix identifies a scene target, rather than an inventory definition.
5. Add a **Show Message** result: **In the lantern light you can see the clock's winding socket.**
6. Apply the lantern changes and save.

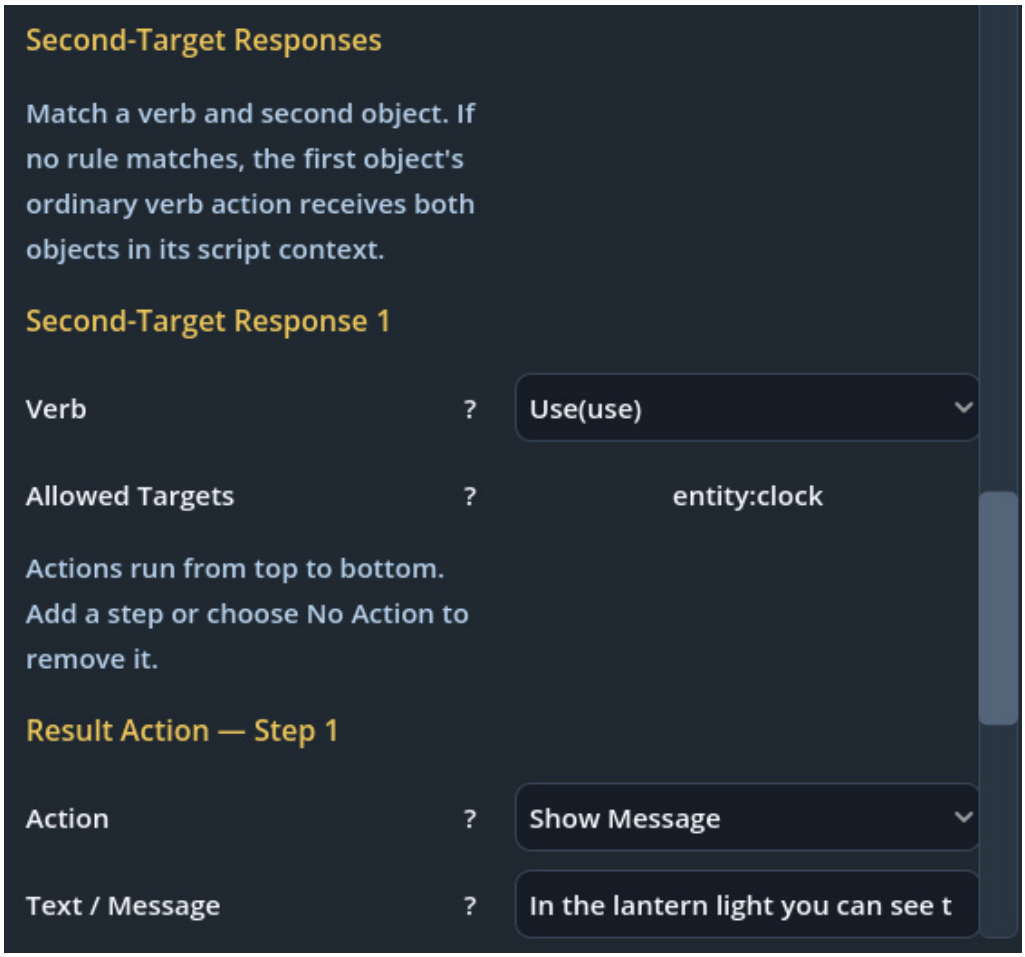

Figure 25. The response is authored on the lantern, with Workshop Clock as the allowed second target.

Start a fresh preview and enter **use the lantern on the clock**. The game should show the pair's result.



Figure 26. The two-object sentence reaches the lantern's response for the clock.

Prepositions and ignored words

It is normal for **on** to appear in Words to Ignore. The running two-object route reads the configured separator from the original command before applying ignored-word cleanup to the two noun phrases.

Use a clear verb-first sentence for this feature: **use the lantern on the clock**. In this release, a leading filler phrase before the verb can fail in the two-object route even when that phrase is ignored for ordinary commands. Do not use **please use…** as your first pair test.

Each side needs to name a complete target after filler words are removed. **Use lantern on** is incomplete. Extra unmatched noun words can also make the pair fail rather than silently selecting a partial target.

Preview is not the pair test

The designer's three-line preview does not display a complete source-and-target pair. It may show one ordinary target for this sentence. Use the running game to verify the second-target response, source opt-in, enabled targets and the action result.

If no specific second-target rule matches, the first object's ordinary verb actions can receive the pair context. Keep the rules explicit while learning the feature so a fallback does not hide a missing target selection.

15. Include inventory commands

An inventory definition has its own ID, name, synonyms, description and verb responses. A physical key lying in the room is a scene object; a key carried in the bag is an inventory item. They can use related words, but they are different records.

1. Open **Inventory Designer** and select or create **Brass key**, with ID **brass_key**.
2. Give it synonyms such as **key, winding key**.
3. Add a Description, such as **A small brass winding key.**
4. Add a Look response in **Verb Actions** if you want a particular action sequence.
5. Give the player the item through the game's normal starting-inventory or Add Inventory setup.
6. Save, run the game and try **look brass key**.



Figure 27. The carried key's Look response is reached by typed input in the example's Verb Single Click mode.

Use the carried key on the clock

On the inventory definition, enable the Use preposition behaviour and add a second-target response for Workshop Clock, just as you did for the lantern. The carried key is now the source. The clock remains a scene target.

Test **use the brass key on the clock**. In this example it answers **The brass key fits the clock. You turn it carefully.**

The two-object inventory route checks that the source item is actually carried and that the active character is allowed to carry it. Test with and without the item. A vocabulary match alone is not an inventory possession check.

Avoid an inventory-name collision

If a visible room object and an inventory definition both answer to **key**, a short one-target sentence can select a different record than you expect. Use distinct phrases while authoring, such as **key on bench** for the scene object and **brass key** for the carried item, then test the natural short wording deliberately.

Do not build puzzle conditions around the fact that Command Preview recognised an inventory name. Definitions can be recognised even when no copy is carried. Use **Has Inventory** or the appropriate inventory condition in the action or script that needs it.

Check your interaction mode

Verb Direct inventory handling is designed around selecting an item or target and choosing the action. It can therefore behave differently from a direct Look response in **Verb Single Click** mode. If a typed inventory command selects the item instead of showing the response you expected, check the project's interaction mode before rewriting the item's synonyms.

The screenshots and direct inventory Look test here use Verb Single Click. Repeat your final tests in whichever mode you ship, including double-click variants if you enable them.

16. Talk to characters and refer to regions

A character can be a target in the same way as the clock. Its record ID, name, character display name and synonyms provide possible matching words in the runtime.

Give Elias a Talk response

1. Select Elias in the scene or open the appropriate Character Designer record.
2. Add a simple synonym such as **clockmaker** alongside **elias**.
3. In **Verb Actions**, give Talk a response.
4. For a quick test, use **Show Message** with a short line. For a conversation, use **Start Dialogue** and choose an existing dialogue.
5. Apply, save and test **talk to elias** and **talk to the clockmaker**.



Figure 28. Talk to Elias reaches his Talk actions. The example uses a simple message so the parser connection is easy to check.

Recognising Talk does not create a conversation or choose dialogue content. You still need a Talk action. If Talk is found but no dialogue starts, inspect the character's response and the selected dialogue record.

For an already placed character, check whether you are editing its reusable definition or an instance override. Keep the accepted vocabulary and response on the record that the room is actually using.

Regions and other scene targets

The runtime can also look up named regions in the current scene. Give a region an ID, a clear name and useful synonyms, and author its Verb Actions when it should be interactive. A non-interactive region can supply a Look description; other verbs still need appropriate interactive behaviour.

The simple editor preview does not provide a complete check for separately stored room regions. Verify them in the game. Also keep player-facing targets distinct from invisible technical volumes: a trigger called **exit_guard** is not automatically a good noun for a player to type.

Try to accept the words shown in the room description and dialogue. If Elias tells the player to inspect the **workbench**, the bench should accept workbench. The player should not need to guess that the internal ID happens to be bench_03.

17. Look around and walk by command

Two built-in verbs have special parser handling: Look without an object can describe the room, and Walk can choose a destination.

Describe the room with Look

In the scene's **General** properties, write a **Room Description**. Save and run the scene. Type **look** without a noun. The runtime uses the room description; if there is none, it can fall back to a basic room-name message.



Figure 29. Bare look reads the room description rather than the clock's Look response.

The example description mentions a stopped clock. If you want the description to change after the repair, make that change part of your game logic or provide a state-aware custom command. Static text does not rewrite itself when a Boolean changes.

Walk to an object

Walk clock uses the matched scene entity's world position as a destination. It requests movement; it does not mean **run the clock's ordinary Walk Verb action**. The player still needs a working navigation setup.

For a predictable test, begin with a clear floor and a nearby object. A model origin inside a wall or outside the navigation area is a poor walking destination. Object placement, colliders and the

walkable area still matter.

Walk to coordinates in a 3D scene

The parser also accepts numeric destinations. Use this mainly for author testing or a game that deliberately exposes coordinates.

Sentence	Interpretation
walk 2 1	X = 2, Z = 1, with the player's current height
walk 2 0 1	X = 2, Y = 0, Z = 1 when parsed; walking is kept at the player's height

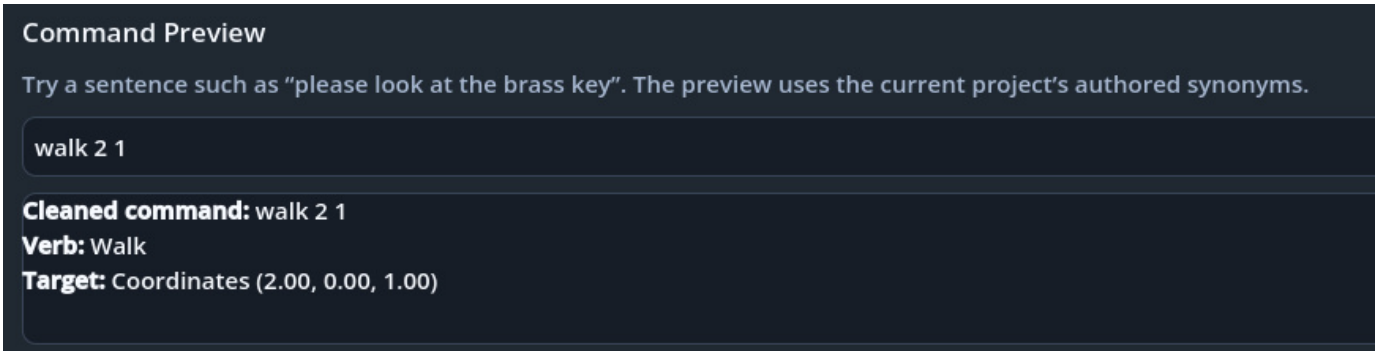

Figure 30. The ordinary preview can identify a numeric destination when no named target matches.

This is navigation, not teleportation. The player follows a walkable path. An unreasonable destination can be refused with **That destination is not on a walkable path.** A third number is not a way to make the player fly to another floor.

Use two or three numbers cleanly. With more numeric values, the current parser uses the last three. Avoid mixing a recognised object name and coordinates in the same walking sentence: a matched scene entity can take priority over the numbers.

The coordinate example was checked in the 3D workshop. Test your own room type and navigation arrangement before offering coordinate commands to players.

18. Write vocabulary and feedback players can use

The best command list usually begins with the words players can see. Use names from descriptions, item labels and dialogue. Then add a few ordinary alternatives that mean the same thing.

Thing in the game	Useful starting words
Workshop Clock	clock, timepiece, grandfather clock
Workbench	workbench, bench
Elias	elias, clockmaker
Look	look, examine, inspect
Take	take, get, pick up

Avoid adding dozens of loose synonyms at once. Every extra alias can collide with another verb or target. Add a small set, test it, and expand it when a real need appears.

Tell the player how to begin

Give the command box a useful placeholder. Add a Help custom command with a few examples. In the opening message or instructions, explain that the player can enter **look**, **look clock**, **talk to elias** and a suitable two-object sentence.

Teach the shape of a command before introducing unusual verbs. If a puzzle needs **wind**, place that word naturally in a description or conversation. A good puzzle should turn on a discovery or decision, rather than a struggle to find the one accepted word.

Translate the vocabulary as well as the response

Translation of dialogue and messages does not, by itself, design a complete parser vocabulary for another language. Review the accepted verbs, object names, synonyms, ignored words and custom phrases for every supported language.

Keep stable internal IDs. Add and test the words your players will actually type, including accents and common alternate spellings where appropriate. Do not assume the displayed translation of a name supplies every accepted input form.

The ignored-word list and custom command rows in this designer are project-wide. If you support several languages in one project, check that one language's ignored words do not erase meaningful words in another. A language-specific script can impose additional rules if your design needs them; the designer does not provide separate per-language tabs for those lists.

Test the same puzzle route after changing the game's language. Check the input, the response, fonts and the GUI layout. Device keyboards and Enter behaviour also need a check on the actual devices you intend to support.

19. Find the cause when a command fails

Work from the smallest test outward. First check the cleaned sentence, then the matched verb and target, then the target's response, and finally the current game's state. Changing everything at once makes a vocabulary problem look like a scripting problem.

What you see	Check this first
No verb matched	Verb ID, display name, synonyms and Enabled. Put action words on the verb.
No target matched	The placed target's name and synonyms; check the intended room and record.
The wrong target matched	Shared aliases and longer phrases. Give competing objects distinct nouns.
Preview works; the game differs	Apply, save and restart. Preview can see records outside the current runtime room.
Text disappears but nothing runs	The parser Enabled checkbox and the text box's Run Text Parser option.
A custom phrase does nothing useful	Assigned script, script errors and the handler's own conditions.
Help works but help clock does not	Custom commands require an exact cleaned phrase.
Use ... on ... fails	Global verb preposition, source opt-in, complete noun phrases and allowed target.
Inventory Look selects an item	Interaction mode, especially Verb Direct or its double-click variant.
A repaired clock repeats the repair	Add an If Variable check to the response.
Enter adds another line	Multiline is enabled, or the intended single-line input is not being used.
Another GUI receives the input	GUI visibility, enabled state, focus and overlays covering the field.

Understand the built-in refusal messages

Message	Meaning in this workflow
The text parser is disabled.	The runtime parser setting is off.
Please enter a verb and a target.	Words were entered, but cleanup removed all of them. A truly blank submission is ignored.
I don't recognize the verb in “…”	No ordinary verb matched and no custom command took the sentence.
What do you want to look?	Look matched, but the sentence did not supply a recognised target.
Where do you want to walk?	Walk needs a usable named destination or coordinates.
Please name both objects.	The preposition route could not resolve complete names on both sides.
Those objects do not interact.	The pair is invalid or has no suitable response.
That custom command has no Script assigned.	A custom phrase matched a row with no handler.

The wording above follows this release. Messages are clues about the route that failed, rather than proof of a particular authoring mistake. Check the corresponding settings before adding more synonyms.

Use the output log for a stubborn case

Show the editor's Output panel or game debug console when you need a closer check. Parser log lines can identify the submitted sentence, matched verb and target, or the custom phrase and script ID. Script compilation errors belong to the handler, even if the command row matched correctly.

Keep a short written list of sentences and expected results while testing. Include ordinary wording, synonyms, missing nouns and commands that should be refused. That list is much more useful than repeatedly typing the one sentence you already know works.

20. Finish with a complete test pass

Run these checks in a fresh preview after saving. Replace the sample names with your own. Use both a direct room test and the normal game startup so the HUD, starting inventory and player state are checked together.

Test	Expected result in the workshop example
look clock	The clock's Look response.
please examine the timepiece!	The same Look response through aliases and cleanup.
wind up the clock	Wind response; clock_wound becomes True.
check clock, before and after winding	The appropriate stopped or running reply.
help	The selected custom handler's instructions.
instructions	The visual help handler, if you added that row.
help clock	Does not match the custom Help row.
talk to elias	Elias's authored Talk actions.
look brass key	The carried item's Look response in the tested mode.
use the lantern on the clock	The source lantern's second-target response.
use the brass key on the clock	The carried key's pair response; test again without the key.
look	The room description.
walk 2 1	Movement to a reachable coordinate, where navigation permits it.
walk 100 100	A refusal for an unreachable destination in this room.
polish the clock	An unrecognised verb unless you deliberately added Polish.
look unicorn	A recognised verb with no matching target.
please the	A request for a verb and target because all words are ignored.

Check the control as well as the words

1. Submit with Enter and confirm the response.
2. Confirm that clearing and return-focus behaviour match your chosen checkboxes.
3. Type a second command without clicking the box again.
4. Click another GUI control, then return to the command line.
5. Test your actual interaction mode, including inventory behaviour.
6. Check the normal startup, room changes, menus and any cutscene that hides the HUD.
7. Test supported languages and device keyboards in the finished game.

Keep the authoring jobs separate

If you want to change…	Edit…
Another word for Look	Look's verb synonyms
Another name for the clock	The clock target's synonyms
Which filler words disappear	Text Parser → Words to Ignore
A complete Help phrase	Text Parser → Custom Commands
What winding the clock does	Clock → Verb Actions → Wind
What using the lantern on the clock does	Lantern → Second-Target Responses
How Enter submits typed text	GUI text box → Type Settings
Whether the clock is already repaired	The response's variable checks or handler logic

Once this small chain works, extend it one response at a time. Keep the vocabulary close to the language of the story, reuse the target's existing actions, and test the sentence in play after the preview looks right.

Version and screenshots

This guide follows **PCA 3D Adventure Studio 0.86.3.4**. Screenshots were captured from the running editor and game preview, using the bundled Clockmaker artwork in a small workshop example. Close-ups are cropped for readability. The instructions use shared editor fields and are written for Windows, macOS and Linux; operating-system file pickers can differ.