PCA 3D ADVENTURE STUDIO

Creating Text and Sierra-Style Adventures

A complete game, from the first command to the final saved position.

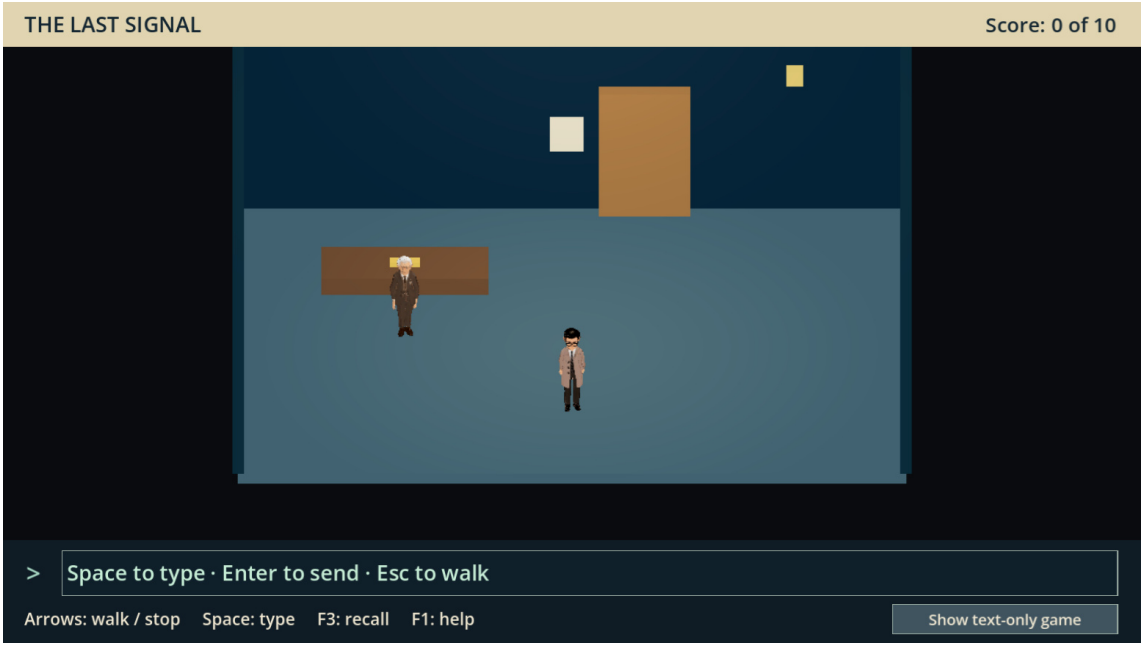

Figure 1. The Last Signal running in PCA. The graphical and text-only versions share the same puzzle rules.

Detailed editor instructions • Typed commands • Action lists
Inventory puzzles • Press-to-walk controls • Complete project script

For PCA 3D Adventure Studio 0.86.3.4

Contents

1. The kind of adventure we are making . . . 4
2. Read this before opening the editor . . . 5
3. Plan the game on one page . . . 6
4. Create a clean project and keep IDs steady . . . 7
5. Build only enough room to test the rules . . . 8
6. Camera, navigation and the player . . . 9
7. Give the parser a useful vocabulary . . . 10
8. Name the things players can talk about . . . 11
9. Open the Text Parser Designer . . . 12
10. Be careful with ignored words . . . 13
11. Use Command Preview as a matching test . . . 14
12. Make a built-in parser command field . . . 15
13. Connect a verb to an action list . . . 16
14. Native two-object commands . . . 17
15. What the native parser will not write for you . . . 18
16. Set up the story variables . . . 19
17. Write the opening transcript . . . 20
18. Add the Adventure Controller . . . 21
19. Expose the functions to action lists . . . 22
20. Build the text-only screen . . . 23
21. Make the transcript readable and scrollable . . . 24
22. Give Enter one owner . . . 25
23. The first five minutes of play . . . 26
24. A room object is not a carried item . . . 27
25. Unlocking and opening are separate actions . . . 28
26. NORTH is a rule, not a magic room name . . . 29
27. Repair the radio and consume the fuse . . . 30
28. Award points once . . . 31
29. Conversations without a dialogue tree . . . 32
30. Save, restore and restart . . . 33
31. Prove that restoring actually restores . . . 34
32. A complete winning playthrough . . . 35
33. Turn the same game into a graphical adventure . . . 36
34. Make a reply box you can dismiss . . . 37
35. Enable Sierra-style movement . . . 38
36. Walking and typing must take turns . . . 39
37. Add familiar keyboard shortcuts . . . 40

38. Make buttons and typed commands share rules . . . 41
39. Call the same command from a visual graph . . . 42
40. Add a new noun without guessing . . . 43
41. Add a new verb and its failure replies . . . 44
42. Add containers, pronouns and more rooms carefully . . . 45
43. Give it the look of an older adventure . . . 46
44. Add motion, sound and typewriter text . . . 47
45. Write replies that help the player think . . . 48
46. Test the game as an unfamiliar player . . . 49
47. Fix input and GUI problems first . . . 50
48. Fix story and save-state problems . . . 51
49. Finish a release players can understand . . . 52
Appendix A. Install the complete controller . . . 53
Appendix B. Command reference . . . 66
Appendix C. Where to change what . . . 67
References and verification notes . . . 68

1. The kind of adventure we are making

You enter a room. You read what is there. You try a command. Something changes. That small conversation between the player and the game is the heart of a text adventure.

This guide builds **The Last Signal**, a short mystery at a railway station. You will find a key, enter a locked cabin, repair a transmitter and send a warning. The same rules will work in two presentations: a text-only screen and a graphical room with typed commands.



Figure 2. The graphical version running inside PCA. The simple room makes the rules easy to see before you spend time on finished artwork.

Presentation	What the player does
Text-only adventure	Reads a scrolling transcript and types commands such as LOOK, TAKE KEY and NORTH.
AGI / early SCI-style adventure	Moves a visible character, types commands and dismisses a reply box before continuing.

> Here, “AGI / early SCI-style” describes the way the game plays. We are making a PCA game with similar controls and puzzle habits. This is not a way to compile an original AGI or SCI project, run Sierra scripts, or import a Sierra game.

2. Read this before opening the editor

The guide has two stages. First, you will learn PCA’s built-in parser and test a few commands. Then you will add one project script that turns the small example into a complete adventure with a transcript, explicit puzzle rules, recall and save commands.

Stage	Input route	Use it for
Built-in parser	A Text Box with Run Text Parser Command When Enter Is Pressed enabled.	Matching authored verbs, names and synonyms; exact custom commands; native verb actions.
Adventure Controller	A Text Box with that option disabled; the script receives its submitted text.	The complete game in this guide, including persistent replies and controlled grammar.

These are two useful routes. They must not both submit the same command. A single Enter press should have one owner. Later, the controller will become the owner of the two finished command fields.

What you need

Use PCA 3D Adventure Studio 0.86.3.4 or a version with the same controls. The screenshots were captured from the running editor and its Play view. The instructions apply across desktop platforms; window decorations may look different.

A mouse and keyboard make the setup easiest. You can build the text-only version without importing any art. For the graphical example, a floor, walls and a placeholder character are enough. The screenshots also use characters already included with the Clockmaker content.

> All code belongs in your game’s Script Designer. There are no engine changes and no practice files to install. The complete controller is printed in Appendix A, in order, without omitted sections.

3. Plan the game on one page

A small adventure should have a clear job. Here, a storm has cut the station’s normal line of communication. The player must get the cabin radio working before the next train reaches a damaged section of track.

Place	Objects and character	Exit
Station Yard (lab)	Brass key, note, cabin door, Morgan the operator.	NORTH enters the cabin after the door is unlocked and opened.
Signal Cabin (cabin)	Spare fuse and radio.	SOUTH returns to the yard.

The puzzle chain

1. Read the note. It tells you where the key is and what the radio needs.
2. Take the brass key from the yard. Use it on the cabin door.
3. Open the door, then go north.
4. Take the spare fuse. Use it on the radio.
5. Turn on the radio. The warning goes through.

There are no timed deaths in this first game. The story mentions urgency, but the player can read at a comfortable pace. Add a real timer only after the basic route works.

Decide the rules before the wording

The key must be carried before it can unlock anything. The fuse is removed from inventory when installed. Opening the door is separate from unlocking it. Repeating a successful command must not award the same points twice. These rules are easier to design now than to patch after the whole game has been written.

4. Create a clean project and keep IDs steady

1. Create a new blank game. Give it the title The Last Signal and save the project.
2. In Project Settings, use a 1280 × 720 game resolution for the layouts in this guide.
3. Keep a player character and place that character in the starting room. Choose that character as the starting player.
4. Create two scenes named Station Yard and Signal Cabin. For the printed controller, their IDs must be lab and cabin.
5. Turn off the opening main menu and startup splashes while testing. Add them back after the adventure works.

Record	Name shown to you	ID used by the example
Scene	Station Yard	lab
Scene	Signal Cabin	cabin
Player character	Eli	tester
Script	Adventure Controller	adventure
Inventory item	Brass Key	brass_key
Inventory item	Spare Fuse	fuse

Names are for people. IDs are the references used by scripts and action lists. You can call the yard “Old Junction” later without changing its ID. If you change an ID, update every reference that uses it.

> This example uses the character’s placement to establish the starting room. Do not look for an old project-level start-scene setting. Put the chosen starting player in the yard, then verify the result with Play / Restart.

5. Build only enough room to test the rules

For a text-only game, the scenes can be plain. They still give PCA a place to keep the player, room objects, inventory use and save state. The full-screen transcript will cover the scene.

For a graphical game, start with the same two rooms and add a few boxes. Use the Scene Designer’s primitive controls to make a floor, a back wall, a bench and a door. Give each object a simple material color. We are testing playability here, not carving a museum-quality door.

Object	Size X, Y, Z	Position X, Y, Z	Color
Yard floor	12, 0.2, 9	0, -0.1, -0.4	#57717b
Back wall	12, 3.8, 0.25	0, 1.9, -4.8	#233e50
Bench	3, 0.35, 1.05	-3, 0.65, -1.6	#795b43
Cabin door	1.65, 2.6, 0.3	1.3, 1.3, -4.55	#a87c4e
Note	0.6, 0.7, 0.06	-0.1, 1.7, -4.5	#eee0bb

Name the visible door Cabin Door and give it ID cabin_door. The note can use ID yard_note. Add a small key-shaped placeholder on the bench with ID lab_key. Give the cabin its own key placeholder, cabin_key, for the optional DROP command.

Add fuse placeholders with IDs lab_fuse and cabin_fuse. Only the one in the item’s current room will be visible. The controller controls those appearances after startup. These are room models; the carried items will be separate Inventory Designer records.

6. Camera, navigation and the player

Choose a fixed room view while learning. A moving camera can make a working parser game harder to test because the player may lose sight of the object they just named.

Setting	Example value
Room camera ID / name	wide / Room view
Projection	Orthographic
Position	X 0, Y 9, Z 13
Look-at target	X 0, Y 0, Z -0.9
Orthographic size	11.5
Default camera	Room view (wide)
Walkable floor height	0
Navigation bounds	X -5.3 to 5.3; Z -3.7 to 3.2
Player entry	X 0, Y 0, Z 2

Set the room’s default camera explicitly. Create an entry spawn named entry in each room. NORTH and SOUTH will use these spawns, so the player arrives at a predictable position instead of retaining coordinates from the previous room.

Create Morgan as a non-player character with ID morgan and place an instance in the yard. Add operator and station operator as synonyms. The player must be a separate playable character. Then use a placeholder actor first. If you use an imported model, assign its idle and walk states in the Character Designer. Check the model’s scale, facing and walk animation with ordinary movement before introducing typed commands.

A visible floor and a walkable navigation area are separate things. Extending the artwork does not extend navigation. Check the room’s navigation bounds and obstacles if the character stops at an invisible edge. Typed room changes also need authored rules; walking to the top of a room does not automatically mean NORTH.

7. Give the parser a useful vocabulary

Open the Verb Designer. Create or review the records below. The built-in parser can match a verb’s ID, display name and synonyms. The finished controller also has its own small vocabulary table; we will keep the two lists in step.

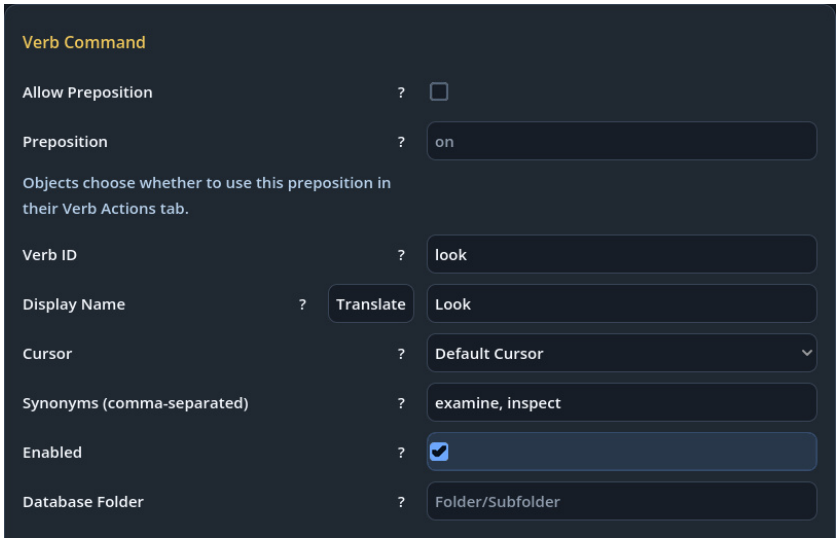

Figure 3. A verb’s Synonyms field gives players more than one sensible way to ask for the same action.

Verb ID	Display name	Suggested synonyms
look	Look	examine, inspect
take	Take	get, pick up
use	Use	apply
talk	Talk	speak
read	Read	
open	Open	
walk	Walk	go

Use short display names when they are also parser phrases. A display name such as “Look at” is fine, but remember that it is another phrase the native matcher can recognize.

> Synonyms help with words, not with every possible sentence. Adding GET to Take does not teach the engine grammar for “could you get that thing over there?” The game still needs rules for the commands it accepts.

8. Name the things players can talk about

Each noun needs a record and a clear name. Use the same noun in the room description and in the parser vocabulary. If the text calls something a transmitter, accept TRANSMITTER as well as RADIO.

Visible name	Record / ID	Synonyms
Brass Key	Room object lab_key; inventory brass_key	key, brass key; add pocket key to the inventory record
Spare Fuse	Room object cabin_fuse; inventory fuse	fuse, spare fuse
Cabin Door	Room object cabin_door	door, cabin door
Station Note	Room object yard_note	note, notice, paper
Radio	Room object radio	transmitter, wireless
Morgan	Character morgan	operator, station operator

Open an object’s inspector and enter its synonyms. Add a useful Look response in its Verb Actions. A background decoration does not have to be clickable, but if the room text mentions it prominently, consider letting the player examine it.

The native matcher can see aliases for room entities and inventory definitions. A carried key and its room model may both be called “key.” Give the inventory item a distinctive alias such as pocket key, then test USE POCKET KEY ON DOOR in Command Preview.

> A successful name match is not proof that the player owns the item. Ownership, visibility, distance and puzzle conditions still belong in the response rules. The complete controller checks those conditions explicitly.

9. Open the Text Parser Designer

1. Open the Text Parser Designer from the designer toolbar.
2. Enable Text Parser.
3. Review Words to Ignore. Begin with a small list rather than removing every word that sounds unimportant.
4. Add HELP as a Custom Command if you want an exact phrase to run a script.
5. Choose Apply Parser Settings after changing the configuration.

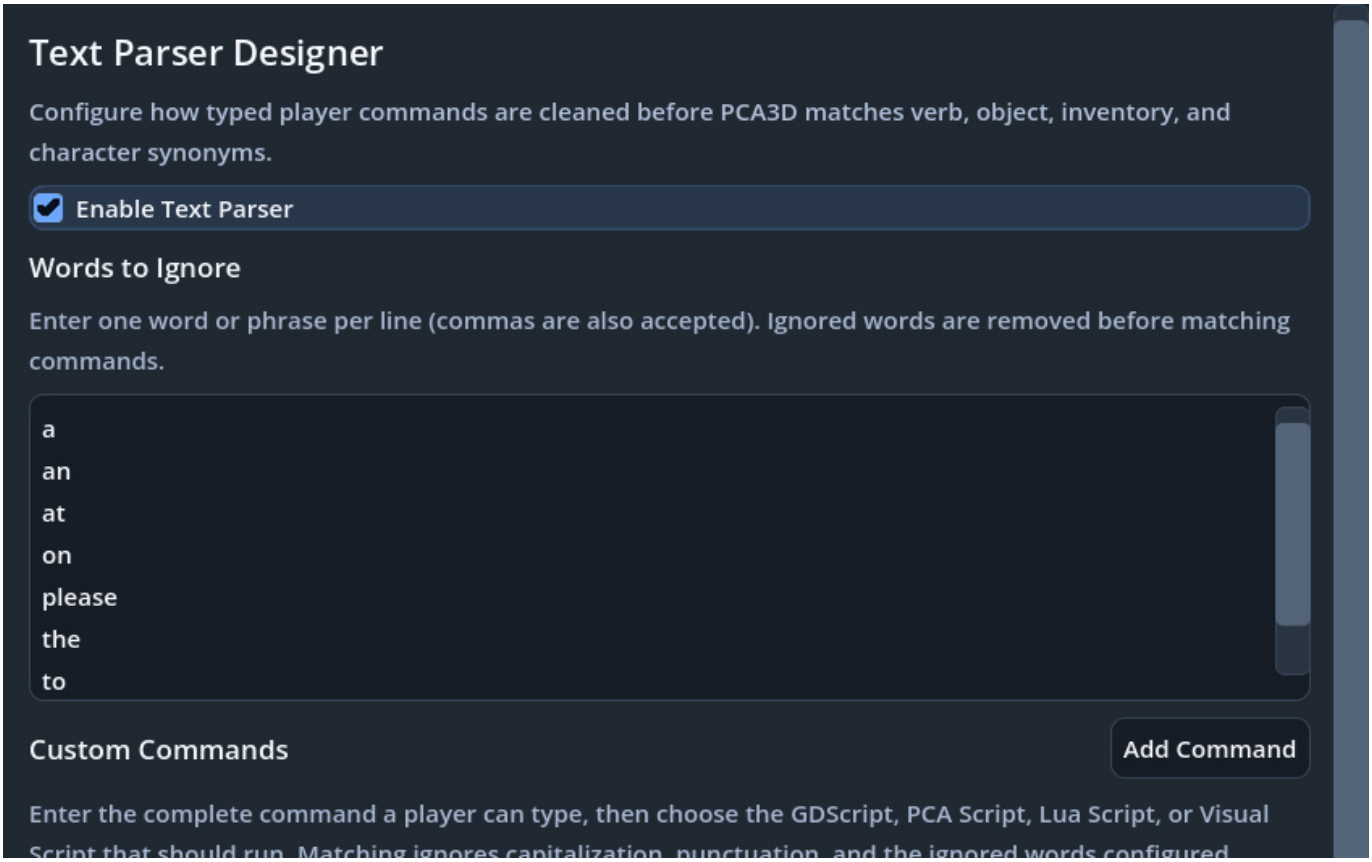

Figure 4. The native parser has one place for ignored words, exact custom commands and matching previews.

The native route first cleans the player’s text. Exact custom commands are checked before ordinary verb-and-target matching. That order makes a phrase such as OPEN SESAME useful: you can give it a special response instead of letting OPEN run a normal object action.

> A Custom Command row is a complete phrase. It is not a pattern field. Writing USE * ON * does not create two parameters, and writing ASK [person] ABOUT [topic] does not create a conversation grammar.

10. Be careful with ignored words

Ignored words make polite or padded commands easier to match. PLEASE LOOK AT THE KEY can become LOOK KEY. PCA also normalizes capitalization and common punctuation, so players do not need to copy your letter case.

Original text	After simple cleanup	Why it matters
Please look at the key!	look key	Articles and polite filler do not need separate responses.
use key on door	use key door, if ON is ignored	A custom exact row is compared after cleanup; the original command is still passed to its script.
ask Morgan about radio	Keep ABOUT for a topic parser	Removing a meaningful separator can destroy the sentence structure you need.

For the native prototype, a, an, the, at and please are reasonable starting words. The built-in two-object parser examines the configured preposition before the ignored-word pass used for ordinary matching. The finished controller in this guide has its own cleaner and keeps ON and ABOUT.

Avoid accidental duplicates

If two custom rows normalize to the same words, the earlier matching row wins. “Look at the key” and “look key” can therefore become the same command. Keep one row or make both point to the same response.

Do not ignore a word simply because it is common English. In a puzzle, IN, ON, OFF, ABOUT and WITH may change the meaning. Test complete phrases after every vocabulary change.

11. Use Command Preview as a matching test

Type PLEASE EXAMINE THE BRASS KEY in Command Preview and choose Parse Preview. The result should identify the Look verb and the key object. This is a quick way to spot a missing synonym or an unexpected target.

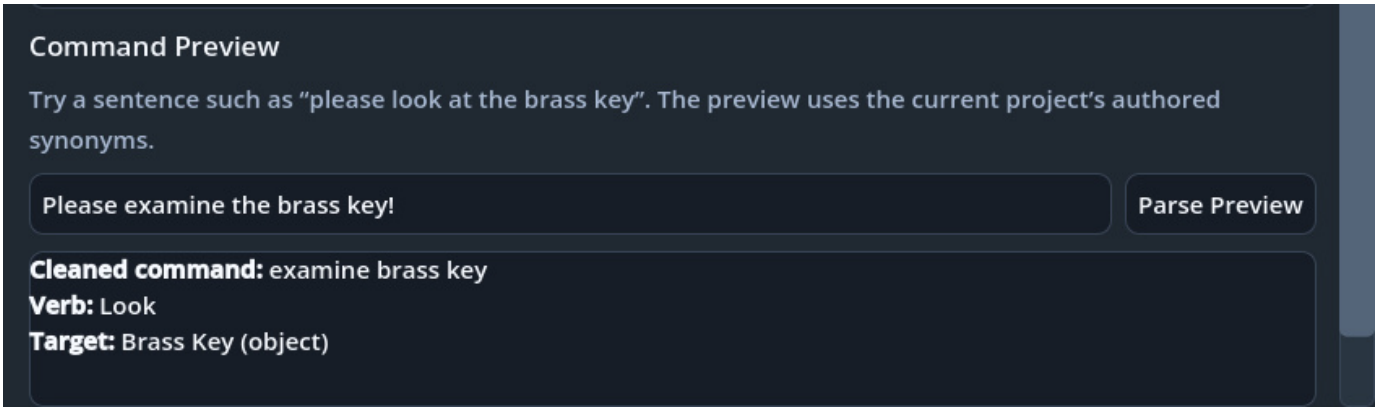

Figure 5. Command Preview shows what the parser recognized. It does not run the puzzle.

1. Try LOOK KEY, EXAMINE KEY and INSPECT KEY.
2. Try the same commands with mixed case and punctuation.
3. Try an unknown noun. Check that it does not accidentally match another record.
4. Try a two-object command and inspect the source and target references.
5. Run the command in Play as well. Preview cannot prove that inventory checks, actions or room transitions work.

The native matcher prefers the longest matching alias. Equal-length matches do not produce an automatic "Which key?" question. Avoid ambiguous names, or add explicit disambiguation in your project's command handler.

12. Make a built-in parser command field

Create a temporary GUI called Built-in Parser Test. Add a single-line Text Box, make the GUI visible while testing and select the Text Box in the GUI Designer.

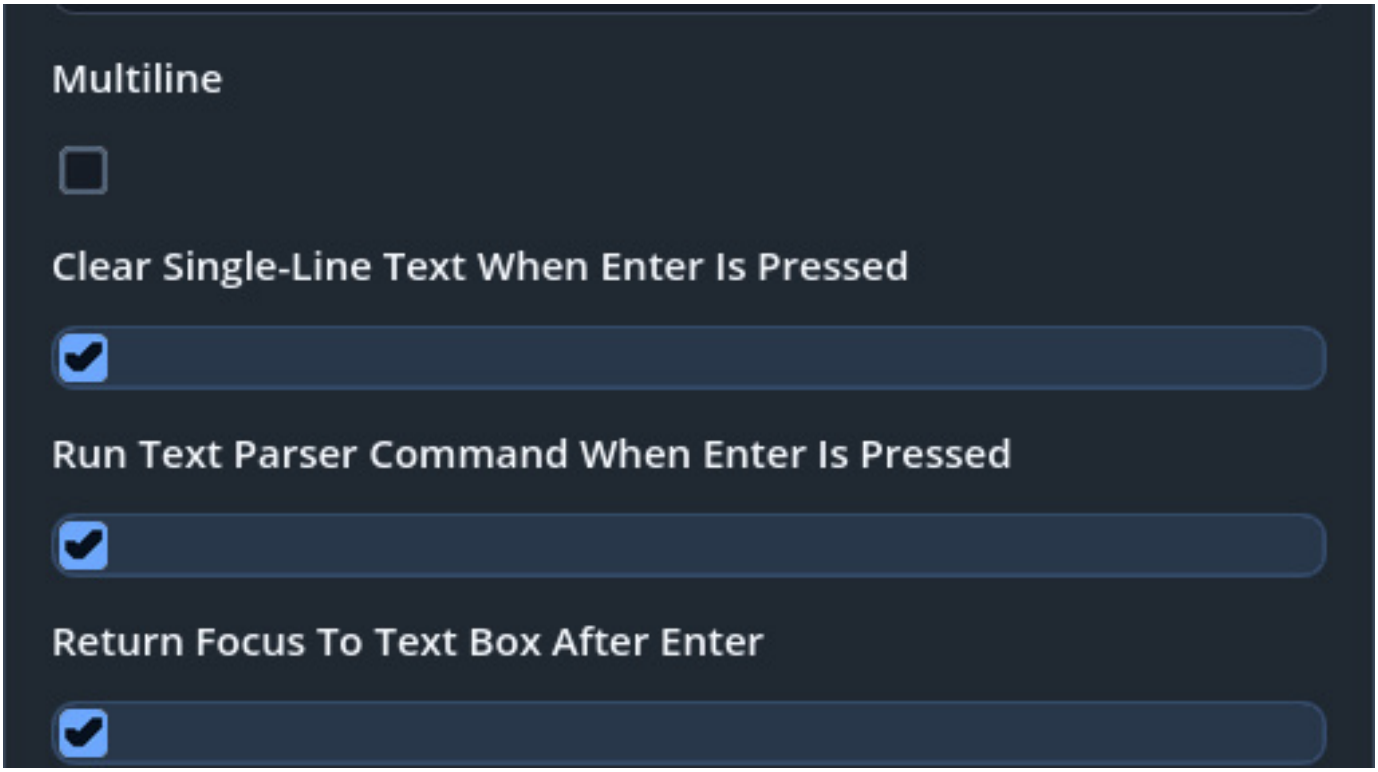

Figure 6. These Enter options connect a single-line Text Box to PCA’s native parser.

Type Settings option	Native prototype value
Multiline	Off
Clear Single-Line Text When Enter Is Pressed	On
Run Text Parser Command When Enter Is Pressed	On
Return Focus To Text Box After Enter	On while testing repeated commands

In Play, click the field, type a command and press Enter. For ordinary object commands, PCA uses the same authored verb responses that your clickable interface can use. A Look response does not need a second copy merely because the player typed it.

> The temporary native field is not one of the finished controller fields. Hide or remove the test GUI after this stage. Leaving two visible command receivers can send input to the wrong field or make a shortcut open the test GUI.

13. Connect a verb to an action list

Select the note object in the Scene Designer. Enable its hotspot if needed, open Verb Actions and find the response for Look. Add a Show Message action containing the note’s text.

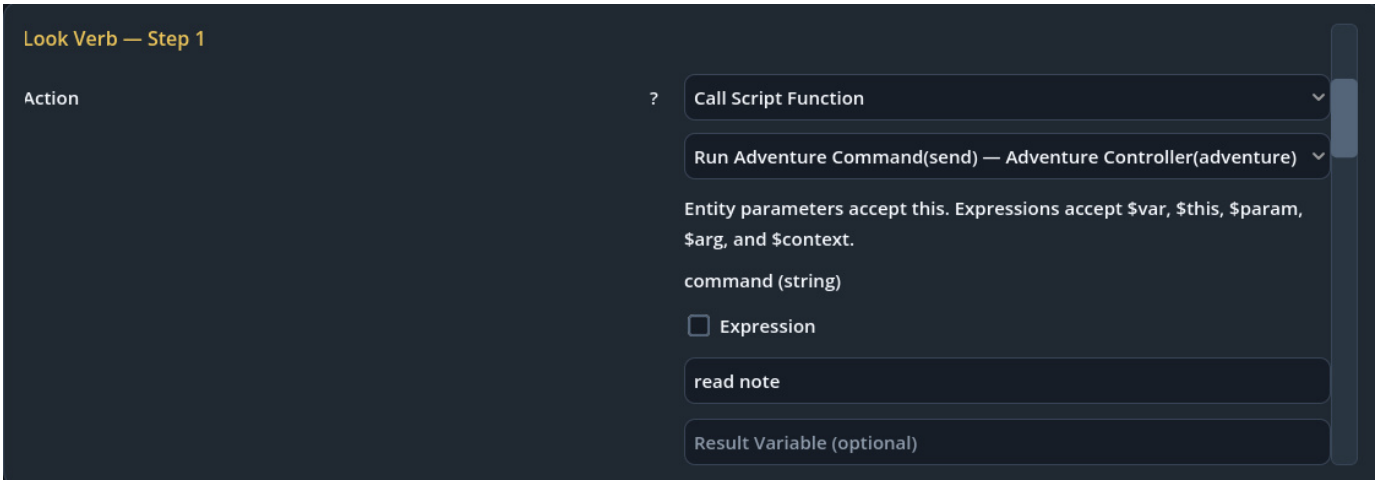

Figure 7. Verb Actions for the note. This shows the finished controller call; begin with a Show Message action for the first test.

Response	Actions in order
Look at note	Show Message: “Key on the bench. Spare fuse inside. Send the warning.”
Take key: simple first test	Add Inventory: Brass Key; Hide / Unhide Object: room key, Visible off; show “Taken.”
Open door: initial test	Show Message: “It is locked.”

Run LOOK NOTE through the native field. Then click the note with the Look verb selected. Both should reach the same object response. That is the useful connection between the parser, the Verb Designer and action lists.

The simple Take sequence is only a first test. It needs a guard before it becomes a finished puzzle: if the item has already been taken, say so and stop. If you put inventory and score changes in an unconditional list, repeated commands can duplicate rewards.

> Later pages route these responses to the shared Adventure Controller instead. Use one finished implementation for each puzzle. Remove the temporary unconditional Take actions before connecting the controller.

14. Native two-object commands

USE KEY ON DOOR contains two nouns and a separator. PCA can handle a configured preposition, but the verb and its source item must both be set up for it.

1. In the Verb Designer, select Use. Enable Allow Preposition and set Preposition to on.
2. Open the source item or object. In its Verb Actions, enable the preposition form for Use.
3. Add a pair rule whose target is the cabin door, then add the actions for that pair.
4. Give the inventory key the alias pocket key to distinguish it from the room key model.
5. Take the key first, then test USE POCKET KEY ON DOOR. Also test the command before taking it.

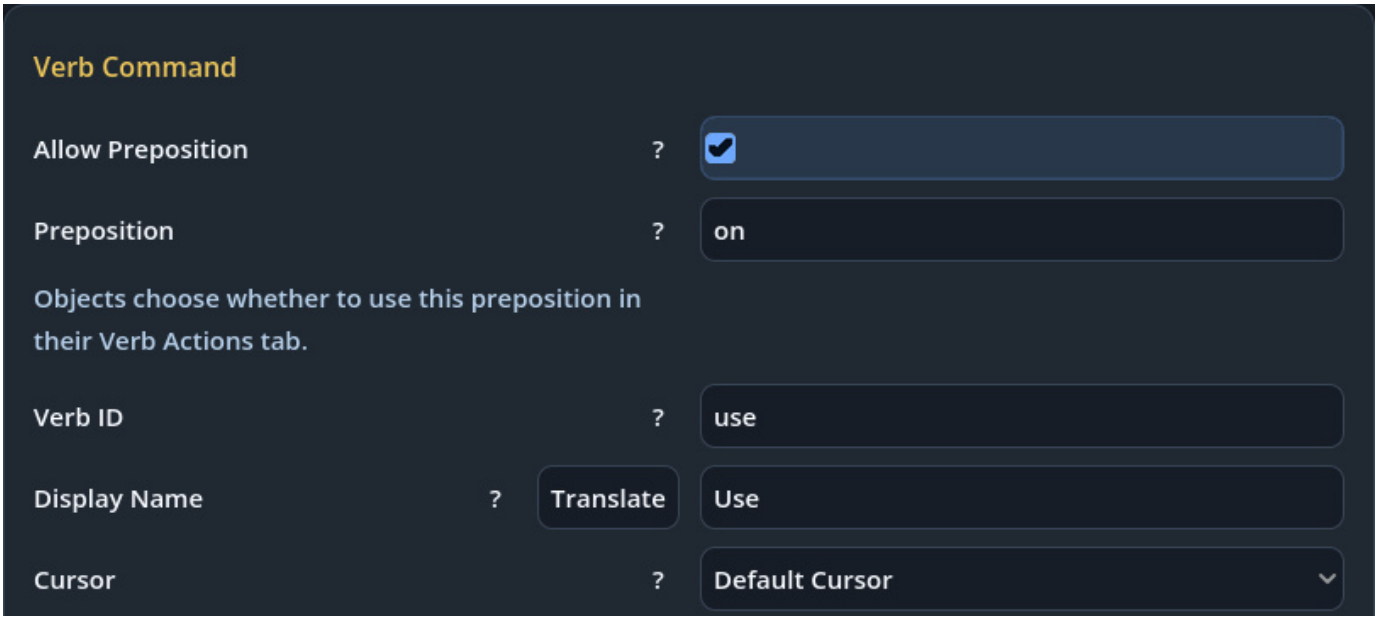

Figure 8. A preposition is configured on the verb; the source record also opts into using it.

The typed pair must name complete nouns on both sides of ON. The runtime also checks that the source is usable and the target exists as an interactive object. A matching rule can then unlock the door, play a sound and update a variable.

> The full controller uses its own explicit USE ... ON ... rule. You do not need both implementations active. Learn the native route here; use the controller route for the complete printed game below.

15. What the native parser will not write for you

The native parser is a matcher for your authored adventure data. It does not invent puzzle responses, conversation topics or story knowledge.

You want…	What to author
LOOK, EXAMINE and INSPECT to mean the same thing	Verb synonyms.
A secret phrase such as OPEN SESAME	An exact Custom Command row pointing to a script.
A key to work only while carried	An ownership condition in the response.
ASK MORGAN ABOUT RADIO	A topic rule or a custom phrase.
IT to mean the last object	A project-level last-object variable and a rule that resolves the pronoun.
A permanent scrolling transcript	A GUI reader plus a project script that stores and displays replies.
Commands separated by AND	A deliberate multi-command grammar, including a rule for stopping after failure.

For a first game, leave pronouns, chained commands and typo correction out. Give the player a short HELP page and useful responses to incomplete commands. A small vocabulary with good feedback feels better than a huge vocabulary that guesses unpredictably.

The next stage adds a modest command handler. It accepts a published set of commands, gives consistent errors and shares all puzzle state between the text and graphical presentations.

16. Set up the story variables

Open the Variable Designer and add the variables below. Use the IDs exactly as printed if you intend to copy the controller. A variable is the game’s memory: it records what happened, rather than making the game guess from what is currently visible.

ID	Type	Starting value / purpose
ta_text_mode	Boolean	true: start with the text-only presentation.
ta_score	Number	0
ta_key_room	String	lab
ta_fuse_room	String	cabin
ta_unlocked	Boolean	false
ta_door_open	Boolean	false
ta_fuse_fitted	Boolean	false
ta_found_key	Boolean	false: guards the first key point.
ta_found_fuse	Boolean	false: guards the first fuse point.
ta_signal_sent	Boolean	false: guards the ending points.
ta_reply	String	Empty: current graphical reply.
ta_last	String	Empty: last submitted command.
ta_lines	Array	Opening text and all later transcript entries.

The key and fuse location strings use ordinary words: lab, cabin, carried and, for the installed fuse, fitted. These strings describe puzzle state. They are not inventory IDs.

> Use real Boolean values for true and false. A string containing the word “false” is text, not a false checkbox value.

17. Write the opening transcript

Set ta_lines to an array of strings. Keep the first description short enough to invite a command. You can use this opening:

```
[
  "THE LAST SIGNAL",
  "Station Yard\nA key lies by Morgan. A note hangs by the door. The cabin is north.",
  "Type HELP for commands."
]
```

This is a JSON array of three strings. The \n sequence inserts a line break inside the second entry. The screenshots use a slightly longer opening paragraph; either version gives the player the same starting facts.

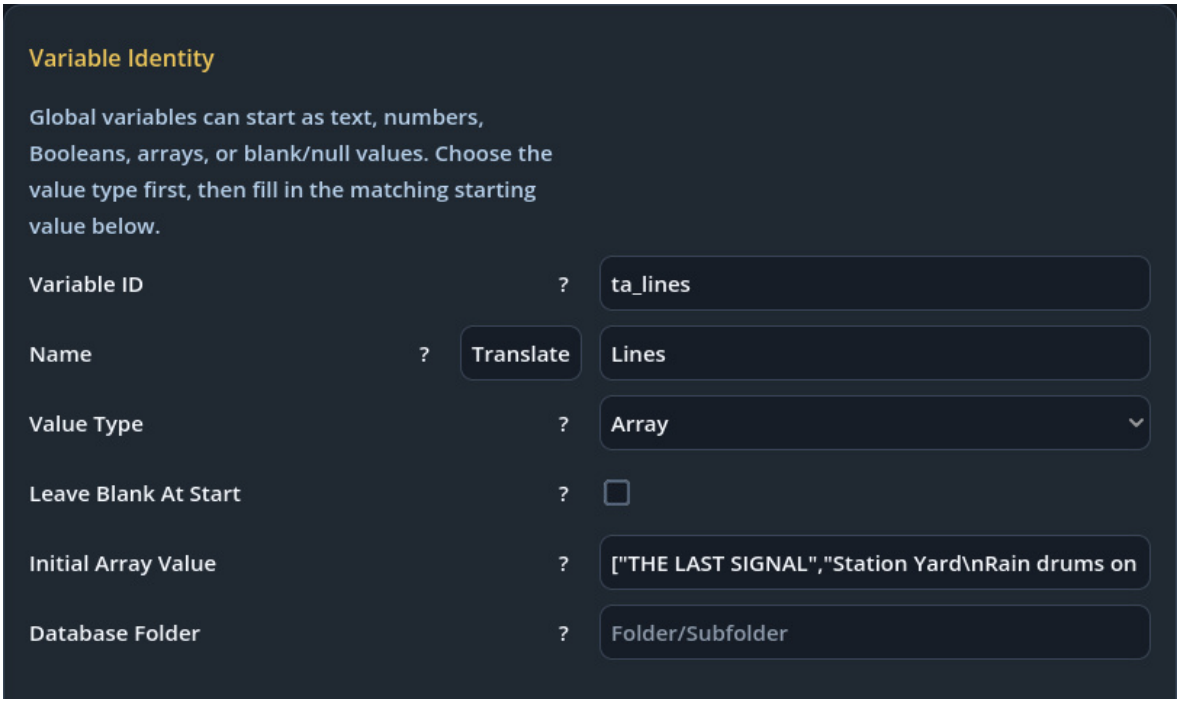

Figure 9. The transcript is a saved Array variable. The reader displays its entries in order.

Each command adds a line beginning with >, then a reply. The controller keeps the latest 100 entries so the saved transcript does not grow forever. An entry may contain several printed lines.

> Room descriptions in the controller are separate from the Scene Designer’s Description field. The built-in parser uses the scene description; the full controller uses describe_room(). Update both while the prototype and controller are being compared.

18. Add the Adventure Controller

1. Open the GDScript Script Designer and add a new script.
2. Set its ID to adventure and its name to Adventure Controller.
3. Enable the script and mark it Global. It must remain available between commands and room changes.
4. Paste the whole of Appendix A into this one script, preserving indentation.
5. Subscribe the script to OnSaveLoaded in its event settings. Its on_save_loaded method will refresh the display after restore.
6. Apply or save the script, then check for compilation errors.

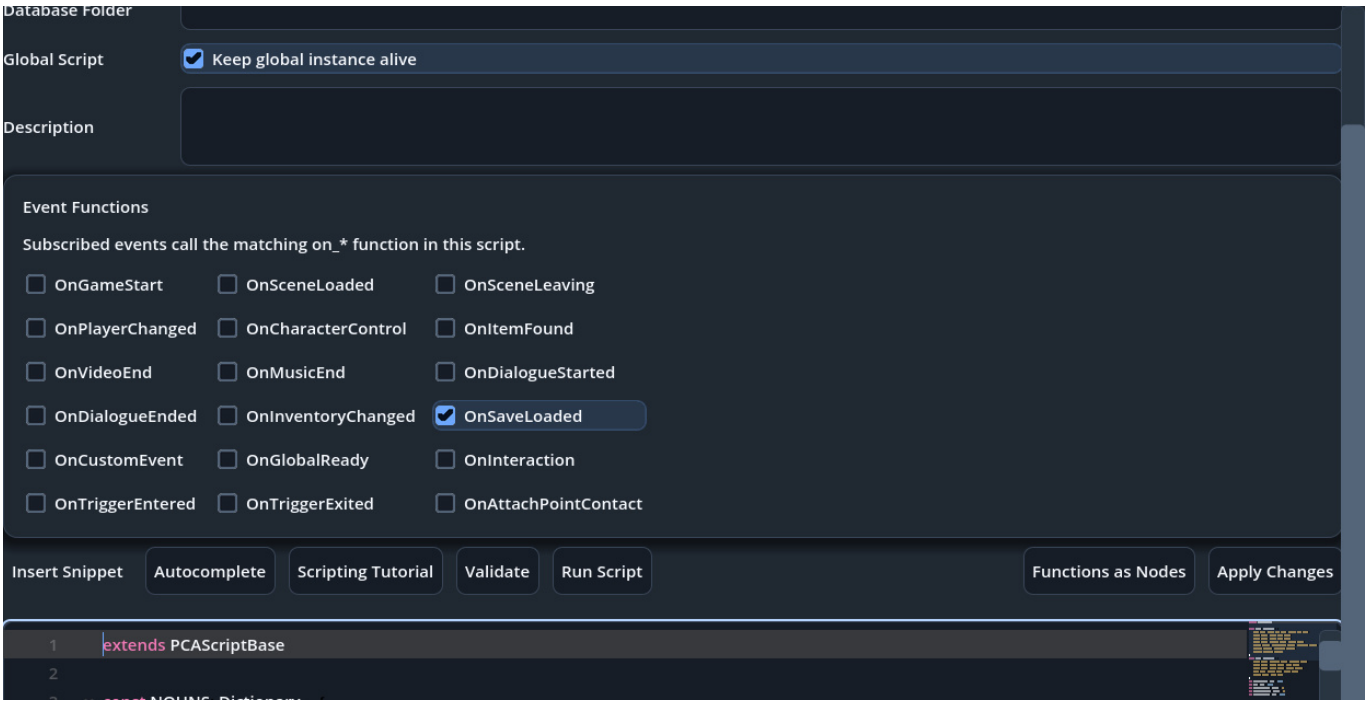

Figure 10. The controller lives in the project’s Script Designer. It does not replace or patch an engine script.

The first line extends PCAScriptBase, which gives the script its Runtime reference and adventure helpers. on_global_ready starts the small display and input routines. The send function is the common entry point for typed commands, GUI buttons and action lists.

> Leave other event subscriptions off unless you deliberately add handlers for them. In particular, do not subscribe this script to every event just to make it “global.” Global lifetime and event subscriptions are different settings.

19. Expose the functions to action lists

Open **Functions as Nodes** for the Adventure Controller. Add definitions for the three functions below. Their code is already in the script; the definitions tell PCA how to show them in visual editors.

Function name	Friendly label	Parameters
send	Run Adventure Command	command: String; default look
presentation	Switch Adventure Presentation	None
dismiss_reply	Close Adventure Reply	None

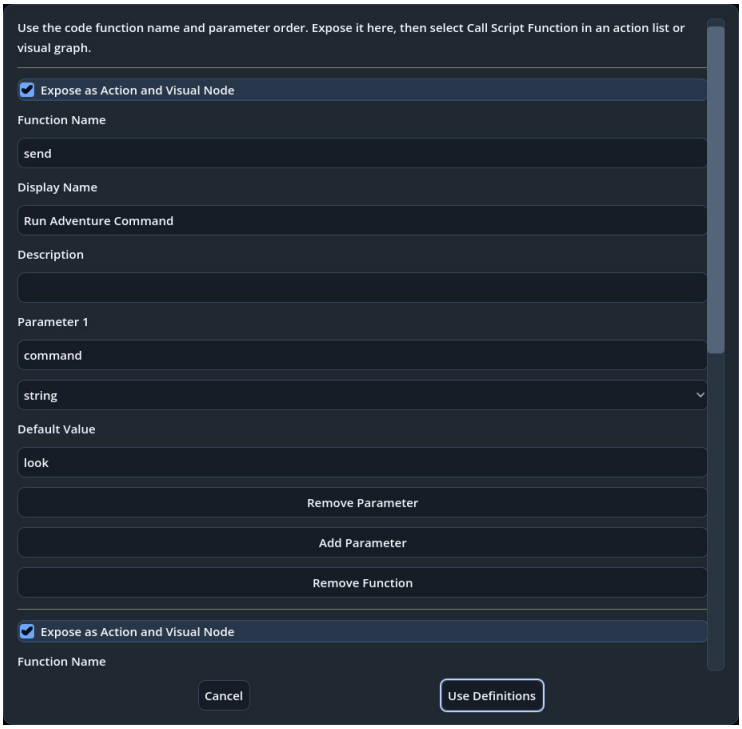

Figure 11. Function definitions provide the names and editable parameters used by action-list and visual-script calls.

Apply the definitions. In an action list, add **Call Script Function**, choose Run Adventure Command from Adventure Controller and enter a command such as read note.

“Use Definitions” applies the declared function list to the editor. It is not a Play button and does not execute the selected function. After applying, look for the function in a Call Script Function selector or the visual-node function tools.

> A definition must point to an actual function with a compatible parameter list. Changing the friendly label does not rename the GDScript function. Keep send(command: String) as written.

20. Build the text-only screen

Create a GUI named Text Adventure Screen with ID text_screen. Make it visible at startup. Use a full-screen layout with the window frame disabled and a Cut opening/closing transition.

Element ID	Type	X, Y	Width × height	Content
paper	Panel	0, 0	1280 × 720	Dark background.
text_title	Label	36, 18	700 × 48	THE LAST SIGNAL
text_score	Label	890, 18	350 × 48	Score: $$ta_score of 10
story_transcript	Text Box	36, 86	1208 × 486	Multiline on.
text_command	Text Box	84, 596	1160 × 64	Multiline off.
text_help_button	Button	720, 671	220 × 34	Help; call send with help.
text_switch	Button	970, 671	274 × 34	Show graphical game

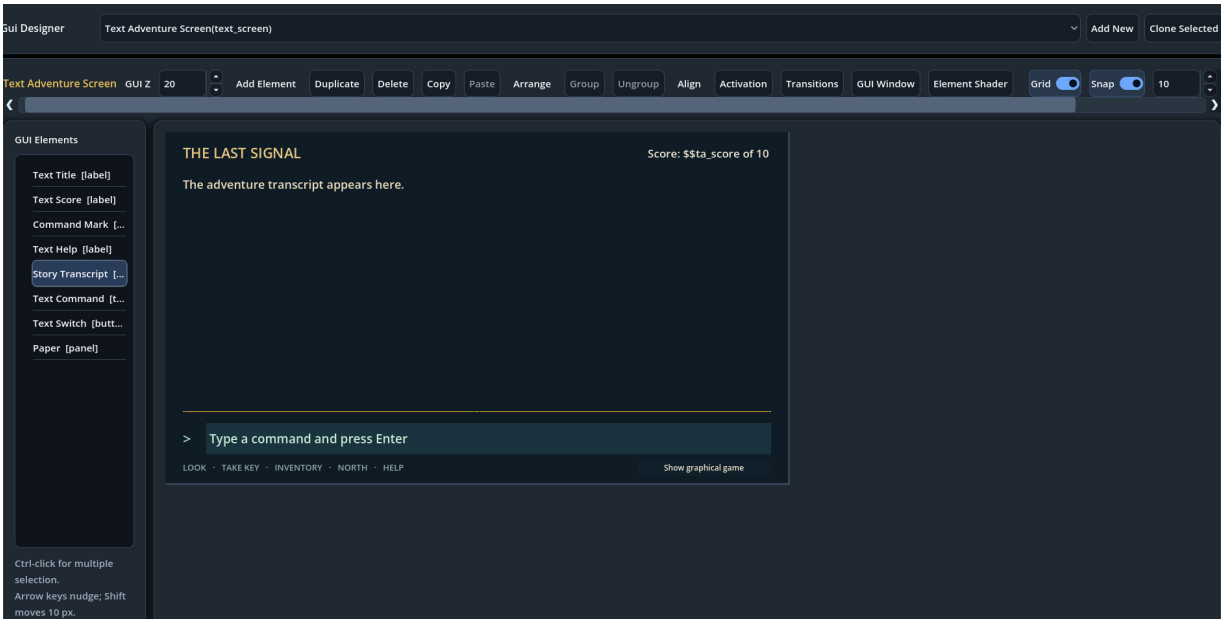

Figure 12. The finished text screen in the GUI Designer. The large reader and the command field have separate jobs.

Add a small > label before the command field and a short help line at the bottom. Use a high-contrast text color and a comfortable font size; 24 for the transcript and 26 for the command field are suitable at this resolution.

> Keep the IDs story_transcript and text_command. The controller locates those controls by ID. Giving them a nicer display name is fine; changing the ID requires changing the script too.

21. Make the transcript readable and scrollable

Select story_transcript. In Type Settings, enable Multiline. Leave it as a large reader rather than trying to fit the whole adventure into a Label. A multiline Text Box provides wrapping, selection and scrolling.

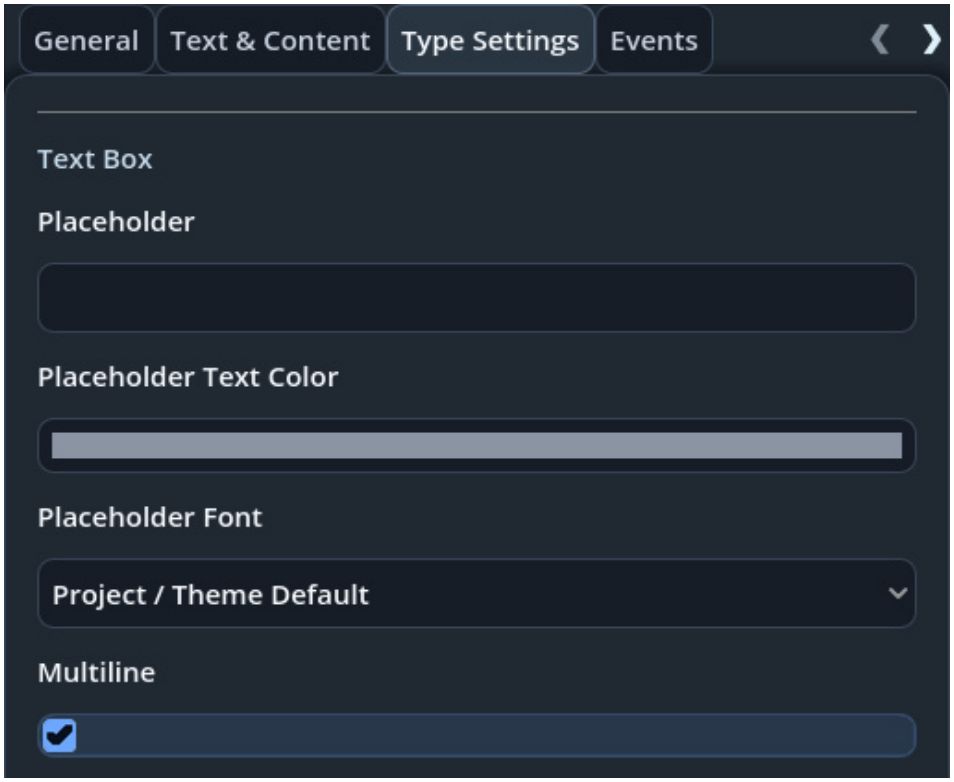

Figure 13. Use a multiline Text Box for the transcript. The controller makes it read-only while the game runs.

The controller joins ta_lines with blank lines between entries. It updates the reader only when the text changes and scrolls to the latest entry. The player can scroll back to inspect earlier replies.

The reader is made read-only by reader.editable = false. The script also sets its read-only text color, because a theme can otherwise dim it as if it were disabled. Read-only and unreadable should not be the same setting.

Keep all important information in text, even in the graphical version. For example, the note should say what the player needs rather than relying on the tiny color of a fuse to explain the puzzle.

> The script stores the transcript in ta_lines. It does not save whatever a player happens to type into the reader, and the reader is not an editable notebook.

22. Give Enter one owner

Select text_command and use the following Type Settings. The finished controller receives text_submitted directly and clears the field itself.

Option	Finished controller value
Multiline	Off
Clear Single-Line Text When Enter Is Pressed	Off
Run Text Parser Command When Enter Is Pressed	Off
Return Focus To Text Box After Enter	Off

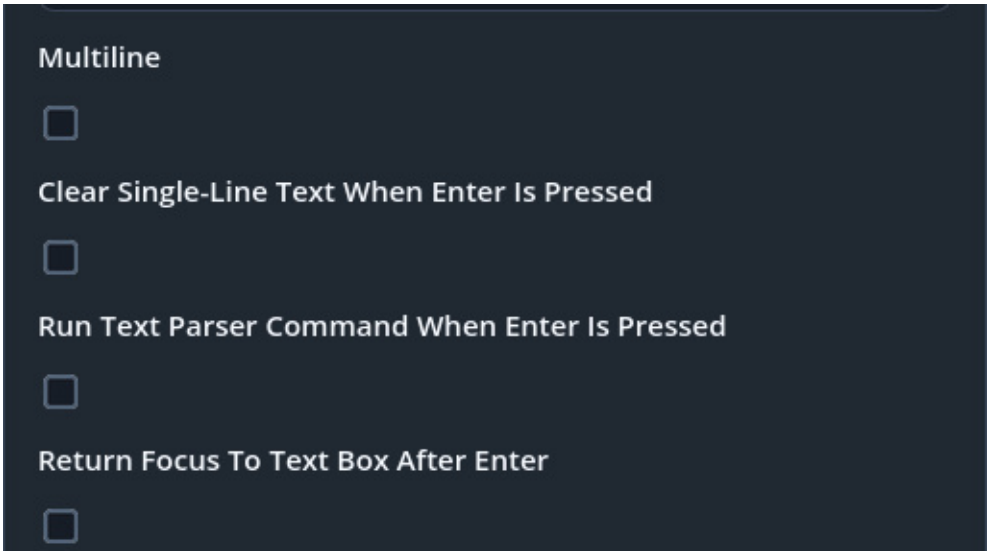

Figure 14. For the controller route, all three built-in Enter behaviors are off.

The script connects the field’s text_submitted signal once, calls send, clears the command and restores focus in text-only mode. You do not need an additional Changed event, Enter action or parser action on the field.

Run the game and type LOOK. You should see both > LOOK and a room description in the transcript. Type an unknown command and check that the game gives a helpful reply rather than doing nothing.

> If each command appears twice, first check this page. A native parser submission plus a script submission is the usual cause. If nothing happens, check the field ID, the script’s Global setting and compilation errors.

23. The first five minutes of play

1. Choose Play / Restart. The title, opening paragraph and command field should appear.
2. Type HELP and press Enter. Read the accepted command examples.
3. Type LOOK, then READ NOTE.
4. Type USE KEY ON DOOR before taking the key. The reply should say that you are not carrying it.
5. Type TAKE KEY, then INVENTORY. The key should be listed.
6. Type GET KEY again. It should say you already have it, with no extra point.

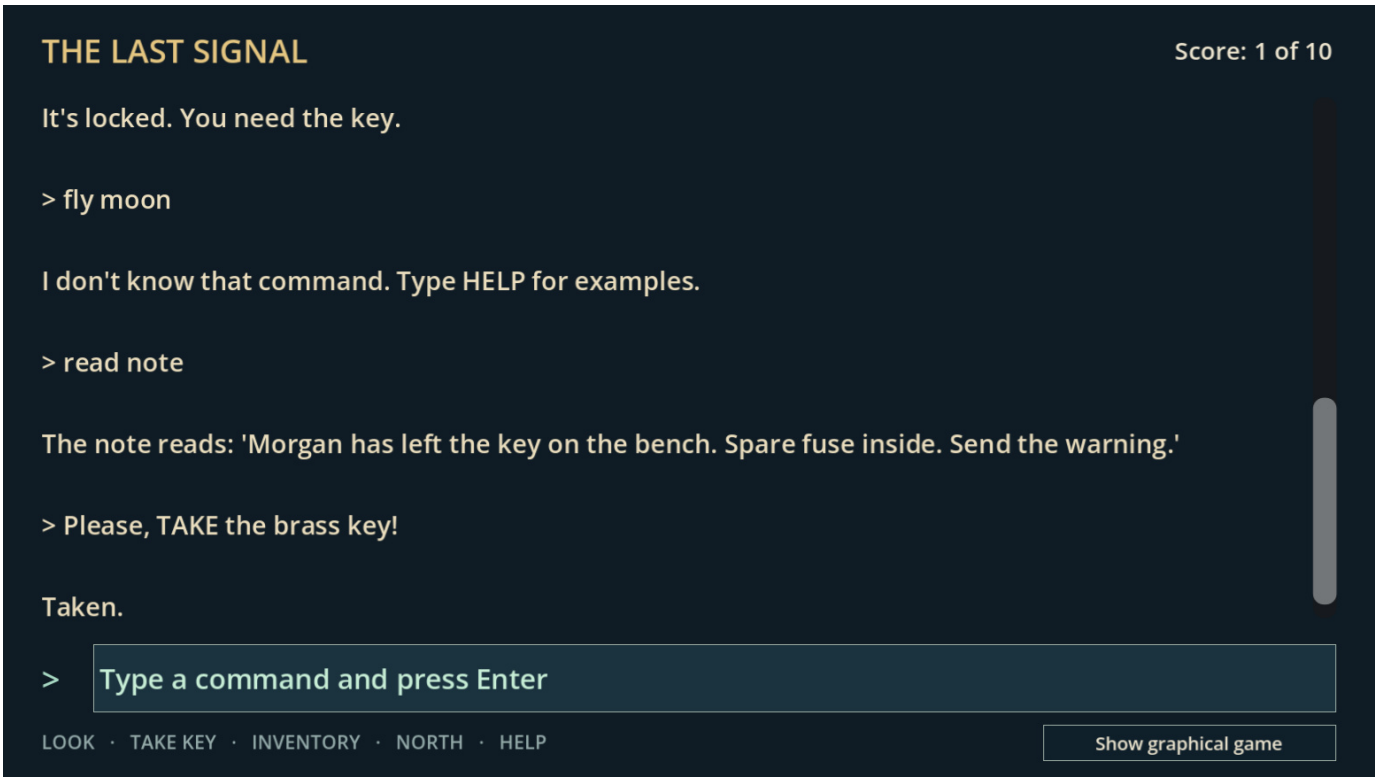

Figure 15. The command is echoed into the transcript before its response. The score changes only on the first pickup.

This short test checks several systems at once: input, noun recognition, room state, real PCA inventory and score guards. If one fails, fix it before adding another room or a more elaborate GUI.

24. A room object is not a carried item

The key on the bench is a room object. The key in your bag is an Inventory Designer record. The controller joins those two representations when TAKE KEY succeeds.

State	Room model	PCA inventory	ta_key_room
Before taking	Visible in yard	No Brass Key	lab
After taking	Hidden	Brass Key present	carried
After dropping in cabin	Visible in cabin	No Brass Key	cabin

Create the Brass Key inventory record with ID brass_key and the Spare Fuse record with ID fuse. Start with an empty inventory. The controller uses Runtime.add_inventory and Runtime.remove_inventory, so ordinary inventory displays can show the same items.

TAKE checks whether the player already has the item. DROP removes it from real inventory and records the room where it was left. The matching placeholder becomes visible in that room. A later TAKE retrieves it.

> The example gives each room a fixed drop placeholder. It does not drop a 3D model at the player’s exact feet. That would be a separate placement feature. For a text game, recording the correct room is usually enough.

25. Unlocking and opening are separate actions

Try OPEN DOOR while the door is locked. The reply should point you toward the key. Then use USE KEY ON DOOR. That changes ta_unlocked and awards two points. It does not enter the cabin automatically.

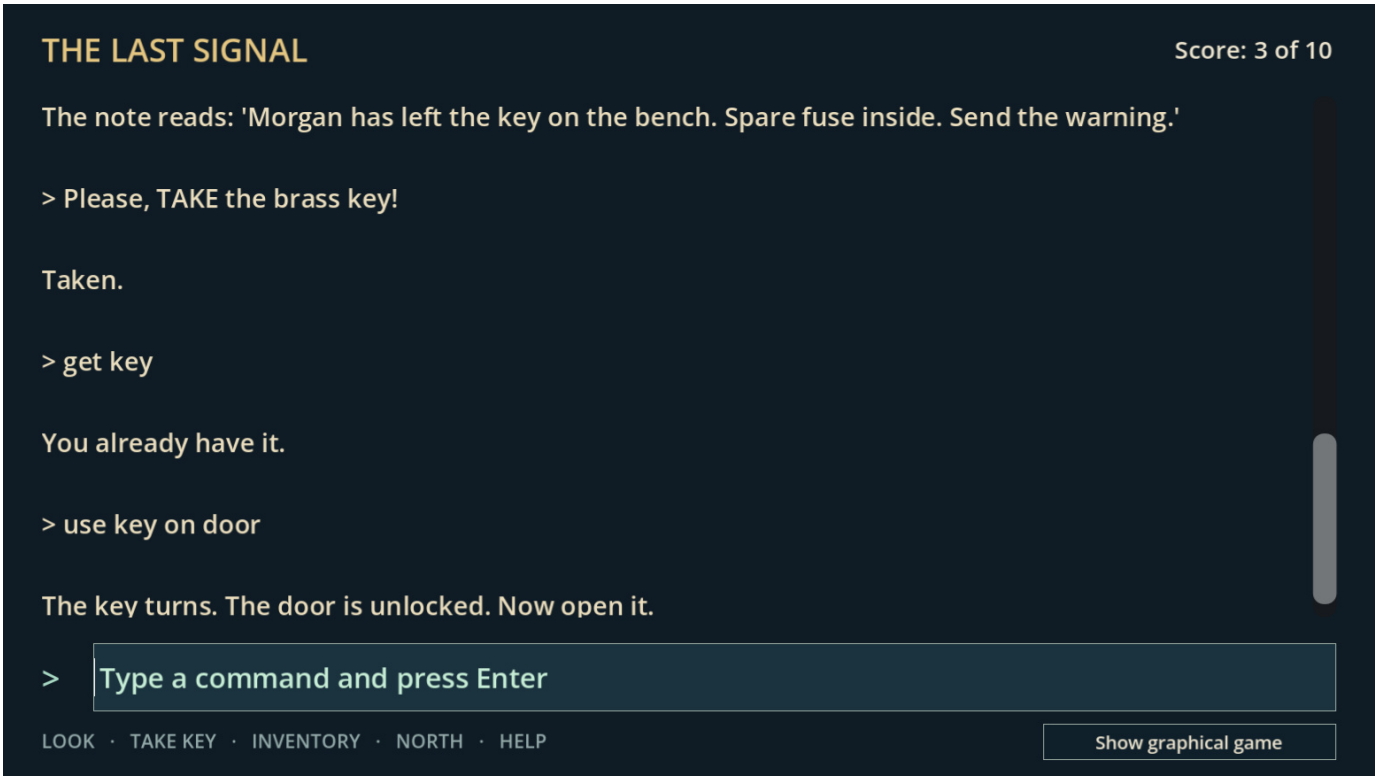

Figure 16. The reply confirms the state change and tells the player the next useful action.

1. Type USE KEY ON DOOR while carrying the key. Expect the unlock reply.
2. Type it again. Expect “already unlocked,” with no extra points.
3. Type NORTH while the door is still closed. You should be stopped.
4. Type OPEN DOOR. The door becomes open.
5. Type NORTH. The player enters the cabin.

The two flags matter. ta_unlocked answers “Does the lock allow opening?” ta_door_open answers “Is the route open right now?” Reusing one flag for both questions makes later CLOSE DOOR behavior awkward.

> The room model disappears when this example’s door opens. A finished game can play an opening animation instead, but that animation should follow the same state change.

26. NORTH is a rule, not a magic room name

A text adventure can let the player type NORTH instead of walking to an exit. The controller’s travel function interprets N, NORTH and GO NORTH as the same route.

Current room	Command	Condition	Result
Yard	N / NORTH / GO NORTH	Door open	Change to cabin, spawn entry.
Yard	NORTH	Door closed	Reply; remain in yard.
Cabin	S / SOUTH / GO SOUTH	None	Change to lab, spawn entry.
Either room	An unavailable direction	No route authored	Reply; remain in current room.

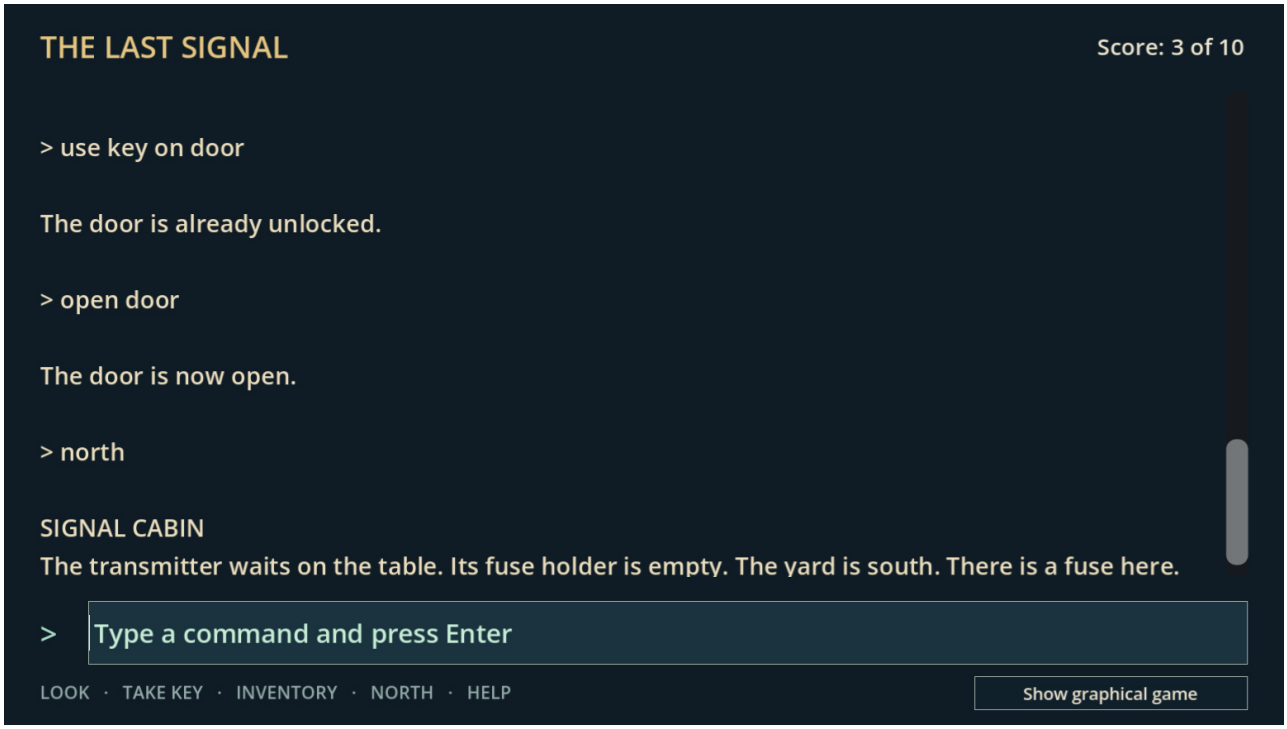

Figure 17. A room change produces a new description in the same transcript.

After entering the cabin, try TALK TO MORGAN. The controller rejects it because Morgan is in the yard. Knowing a noun does not mean the noun is present.

> In the graphical version, arrow keys move within a room and NORTH changes rooms. They are separate controls. If you later add a doorway trigger for automatic travel, make it call the same guarded route instead of bypassing the lock.

27. Repair the radio and consume the fuse

Inside the cabin, type LOOK RADIO. The response names the missing fuse. TAKE FUSE adds the inventory item and earns one point. USE FUSE ON RADIO removes it from inventory and marks the fuse as fitted.

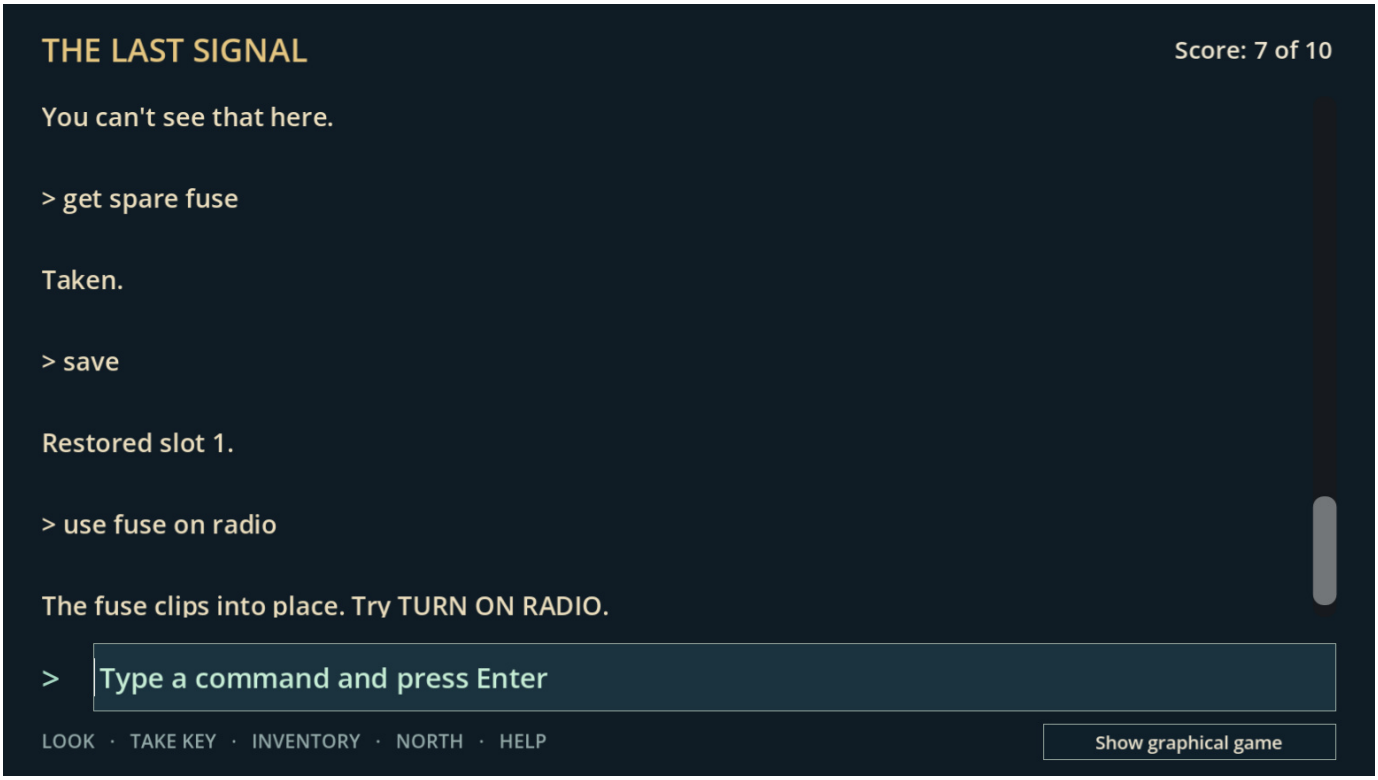

Figure 18. A successful item-use command changes inventory and puzzle state together.

Command	Before fitting	After fitting
LOOK RADIO	Empty fuse holder.	Fuse fitted; ready to switch on.
USE FUSE ON RADIO	Requires a carried fuse.	Already fitted; no extra reward.
TURN ON RADIO	Nothing happens.	Warning sent; ending reached.

The fuse location becomes fitted rather than lab, cabin or carried. That prevents an installed fuse from reappearing as a loose room model. Repeating USE FUSE ON RADIO gives a reply without consuming another item.

> Keep the inventory removal, location change and success flag in one successful branch. If only the picture disappears, the player may still carry the item. If only inventory changes, a room model may remain available to take again.

28. Award points once

Scoring is optional, but it is a useful way to make this example feel familiar. The important rule is that a completed task can award points only once, even if the player repeats commands or drops an item and picks it up again.

Achievement	Guard variable	Points
First key pickup	ta_found_key	1
Unlock cabin door	ta_unlocked	2
First fuse pickup	ta_found_fuse	1
Fit fuse	ta_fuse_fitted	3
Send warning	ta_signal_sent	3
Total		10

```
func award(flag: String, points: int) -> void:
    if bool(Runtime.variables.get(flag, false)):
        return
    Runtime.variables[flag] = true
    Runtime.variables["ta_score"] = int(
        Runtime.variables.get("ta_score", 0)) + points
```

This is the same function printed in Appendix A. It sets the guard before adding points. The HUD label uses Score: $$ta_score of 10. The controller converts the starting score to an integer so the opening display reads 0 rather than 0.0.

For a larger game, give each optional discovery its own guard. Do not use a single “puzzle complete” flag for several independent rewards unless that is the behavior you want.

29. Conversations without a dialogue tree

A parser game does not have to offer a list of dialogue choices. TALK TO MORGAN can give a general introduction; ASK MORGAN ABOUT RADIO can give a more specific clue.

Command	Purpose
LOOK MORGAN	Describe the person.
TALK TO MORGAN	Give a general response and point toward the note.
ASK MORGAN ABOUT RADIO	Reveal the repair clue.

The printed controller accepts that one named topic phrase. It does not automatically understand every topic, and its exact ASK phrase does not use all the ordinary noun aliases. Add explicit topic variants if you want ASK OPERATOR ABOUT RADIO too.

When you add topics, decide who is present, which subject is known and whether the answer changes after a puzzle. A good answer can acknowledge success: after the warning, Morgan could thank the player instead of repeating the original instructions.

Mix typed topics with PCA dialogue

A typed command can also call a Start Dialogue action or a script function that starts an authored conversation. Use that when you want spoken lines, portraits, lip sync or a choice tree. Keep the command’s room and availability checks before the dialogue starts.

> A response in this guide is written into the transcript or reply card. It is not automatically voiced, animated or lip synced. Those presentation features require their normal dialogue setup.

30. Save, restore and restart

The first version uses one quick-save slot. SAVE writes slot 1 under the name The Last Signal; RESTORE loads that slot. The same commands work from either presentation.

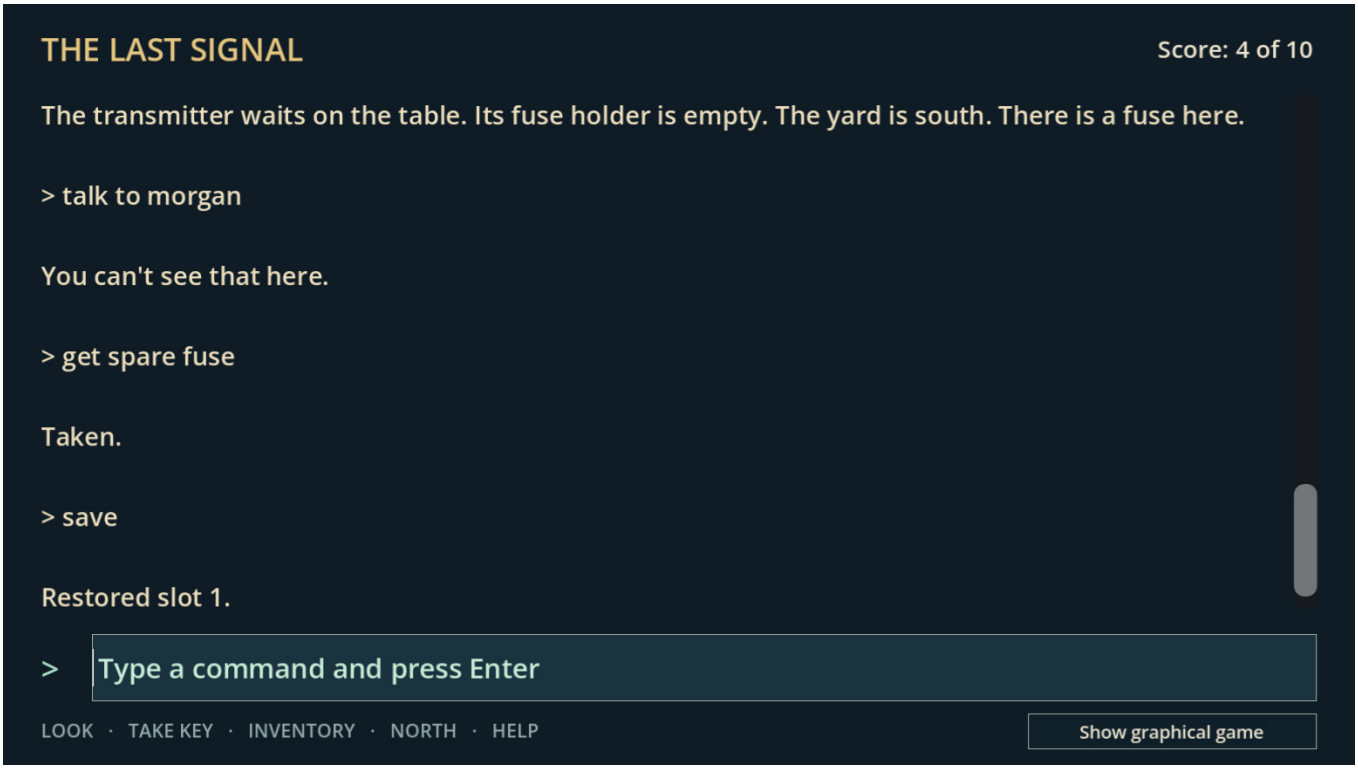

Figure 19. Restoring returns the room, inventory, puzzle flags, score and saved transcript together.

Command	Behavior
SAVE / SAVE GAME	Writes slot 1. An earlier save in that slot is replaced.
RESTORE / LOAD / LOAD GAME	Loads slot 1, if a file exists.
RESTART	Asks for a second command.
RESTART YES	Starts a new game after RESTART has been requested. Saved files remain.

The game stores story progress in PCA variables and real inventory, not in temporary local GDScript fields. That is why the repaired radio and score survive restore.

> RESTORE is deferred to the runtime. Loading rebuilds script instances, so the controller must not keep waiting inside the script instance being replaced. The new instance handles OnSaveLoaded and refreshes its screen. Keep the deferred call from Appendix A.

31. Prove that restoring actually restores

1. Reach the cabin and take the fuse. Check that the score is 4.
2. Type SAVE. Expect the slot 1 confirmation.
3. Type DROP FUSE. Confirm that inventory no longer contains the fuse.
4. Type RESTORE. The fuse should be carried again and the score should still be 4.
5. Fit the fuse and turn on the radio. The score should reach 10.
6. Restore the earlier save again. The radio should be unfitted and the score should return to 4.

Also test a fresh game with no save file. RESTORE should explain that there is no save in slot 1. Do not confuse an old test save from another revision with the new project’s starting state.

The sample keeps one quick slot to make the code easy to follow. Before release, consider opening a proper Save Game / Load Game interface so players can keep several checkpoints. PCA’s Save Game Widget and system menu actions can provide that interface.

> PCA saves can include mutable project data as well as story state. When you change a script during development, test with Play / Restart and new saves. An older save can contain the earlier project configuration.

32. A complete winning playthrough

Type this	What you should see	Score
LOOK	Yard description.	0
READ NOTE	Key and radio clues.	0
TAKE KEY	Taken.	1
USE KEY ON DOOR	Door unlocked.	3
OPEN DOOR	Door open.	3
NORTH	Cabin description.	3
LOOK RADIO	Empty fuse holder.	3
TAKE FUSE	Taken.	4
USE FUSE ON RADIO	Fuse fitted.	7
TURN ON RADIO	Warning sent; ending.	10

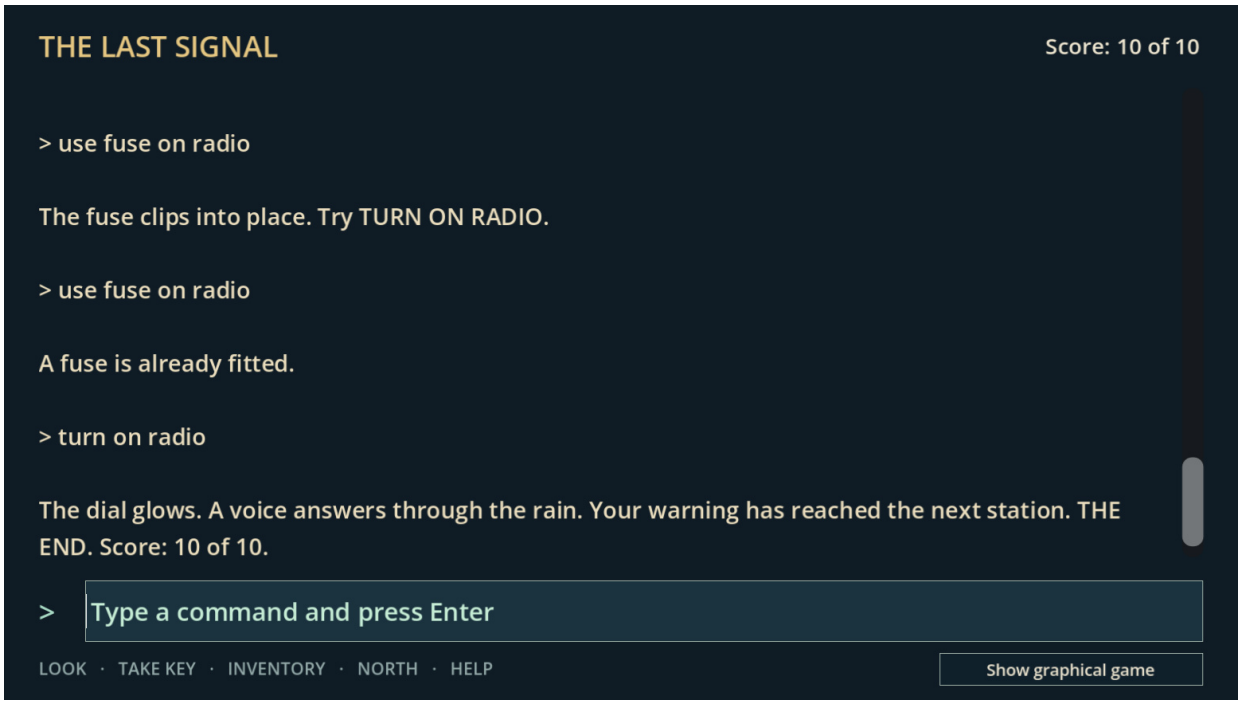

Figure 20. The text version reaches the ending through the same puzzle state used by the graphical game.

After the ending, repeat TURN ON RADIO. The score must stay at 10. Return to the yard, drop the key and take it again. The score must still stay at 10. Those repeated actions are useful tests, not just player mischief.

33. Turn the same game into a graphical adventure

Create a GUI with ID graph_hud. It provides the status strip and command line while the room and player remain visible. Its command field must have ID graph_command.

Element	X, Y	Width × height	Content / setting
Top Panel	0, 0	1280 × 50	Light background.
Title Label	24, 4	600 × 42	THE LAST SIGNAL
Score Label	862, 4	390 × 42	Score: $$ta_score of 10; right aligned.
Bottom Panel	0, 604	1280 × 116	Dark background.
graph_command	66, 616	1188 × 51	Single-line Text Box; native Enter options off.
Help Label	24, 677	820 × 32	Arrows: walk / stop; Space: type.
Switch Button	1000, 676	254 × 34	Show text-only game.

Set graph_hud to start hidden. The controller shows it when ta_text_mode is false. Set the project’s Main HUD system GUI to None so the template’s old verb buttons do not appear under your new command bar.

On both presentation-switch buttons, use an action list with Call Script Function → Switch Adventure Presentation. No parameter is needed. The function changes presentation without resetting the puzzle.

> The text screen covers the room; it does not delete it. The controller disables player movement in text-only mode and enables it in graphical mode.

34. Make a reply box you can dismiss

Create a GUI with ID reply_card. Make it hidden at startup and Modal. Use Cut transitions. Give it a higher Z order than the graphical HUD.

Element ID	Type	X, Y	Size	Content
reply_panel	Panel	170, 170	940 × 340	Light background, dark border.
reply_text	Label	202, 194	876 × 230	$$ta_reply; 27-point dark text; top/left aligned.
reply_ok	Button	820, 445	240 × 46	Enter / OK



Figure 21. The graphical reply stays open until the player dismisses it. Walking is blocked while this modal GUI is open.

Set the button’s Click event to an action list calling Close Adventure Reply. The controller also handles Enter and Escape while a reply is open. Dismissing it hides the card and releases GUI focus so movement can resume.

> A timed Show Message action and this reply card are different presentations. Use the card for the classic “read, press Enter, continue” rhythm. Do not add a second Show Message action to the same successful controller response.

35. Enable Sierra-style movement

1. Open Global Settings, then Adventure & Cursor.
2. Find Keyboard Player Movement and enable keyboard movement.
3. Choose Arrow Keys for Movement Keys.
4. Choose Sierra AGI: Press to Walk / Stop for Movement Style.
5. For the native parser prototype, enable Space Focuses Assigned Text Parser. For the finished controller, turn that option off: the controller handles Space itself.

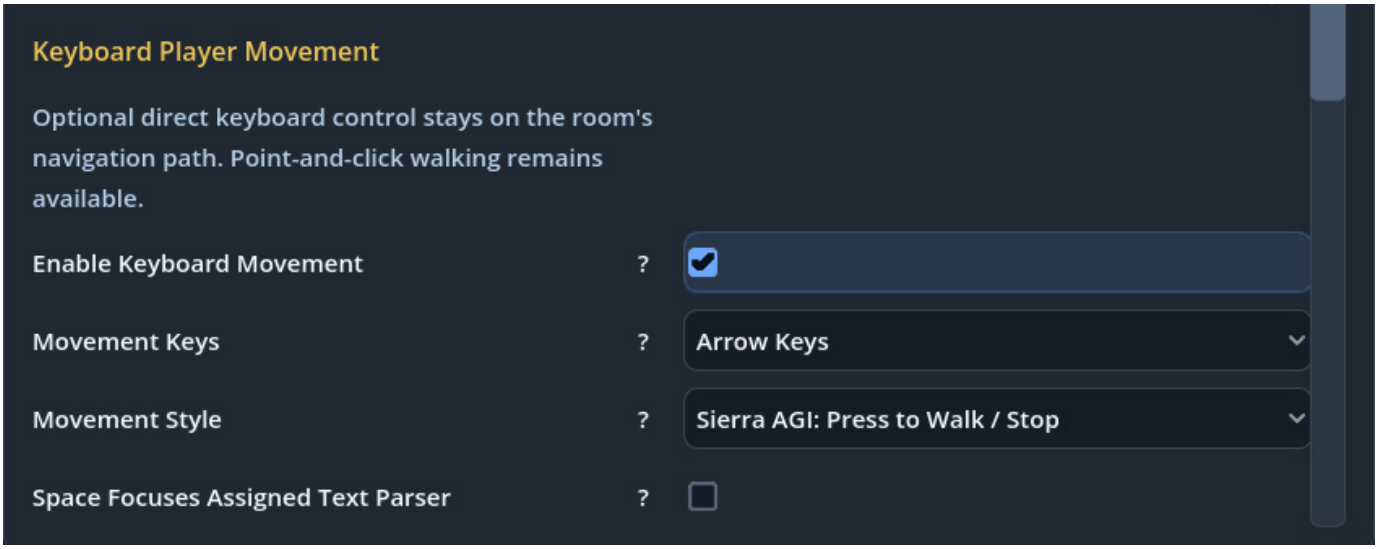

Figure 22. PCA already includes press-to-walk / press-again-to-stop controls.

In this mode, pressing a direction starts continuous walking. Press the same direction again to stop, or choose a different direction to turn. An obstacle or the end of navigation stops the walk.

This reproduces a familiar part of early Sierra play without changing PCA’s engine. It still uses your room’s navigation and collisions. The printed example uses the four arrow directions; it does not claim to reproduce every diagonal or joystick detail of an original interpreter.

> Point-and-click walking can remain available. If you want a strictly keyboard-only game, review mouse and interaction settings separately. Selecting Sierra movement alone does not remove every mouse action.

36. Walking and typing must take turns

When the command field owns the keyboard, arrows should edit the line rather than move the character. When the field releases focus, arrows can control the player. This handoff is the part most likely to make a parser game feel broken if it is overlooked.

Situation	What should happen
Walking, then Space	The command field takes focus. Type a command.
A printable letter while not typing	The controller opens the graphical command field and inserts that first character.
Typing, then Enter	The field submits once and releases focus in graphical mode.
Reply open, then Enter	The reply closes; that Enter does not become another command.
Typing, then Escape	The field releases focus so movement can resume.
Text-only mode	The command field regains focus after each response.



Figure 23. After closing the reply, the scene is visible again and movement can resume.

Use Arrow Keys rather than WASD when automatic type-to-open is enabled. Otherwise a letter such as W is both the beginning of a command and a movement key. You can support both schemes, but you must decide which behavior takes priority.

37. Add familiar keyboard shortcuts

Key	Behavior in this controller
Arrow key	Start walking in that direction; same direction again stops.
Space	Focus the command field when not already typing.
Enter	Submit a command, or dismiss an open reply.
Escape	Release the command field, or dismiss a reply.
F1	HELP, while not typing or reading a reply.
F3	Recall the last submitted command into the field. Press Enter to run it again.
F5	SAVE to slot 1, while not typing or reading a reply.
F7	RESTORE slot 1, while not typing or reading a reply.
Tab	INVENTORY, while not typing or reading a reply.

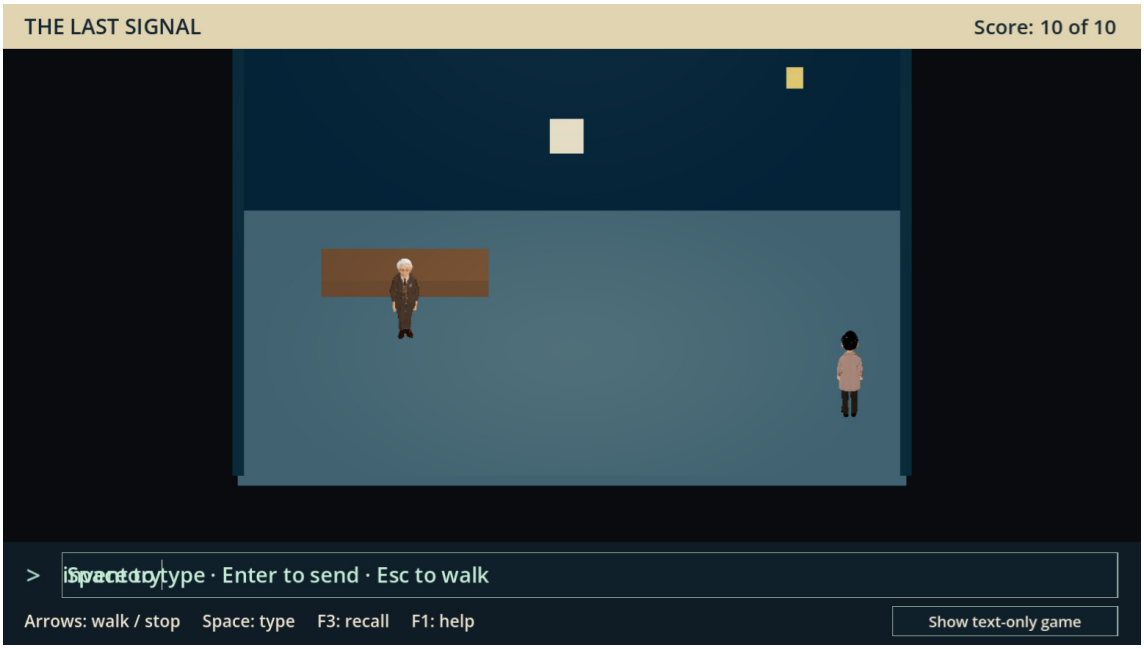

Figure 24. F3 puts the previous command back into the input field so it can be edited or submitted.

These are project shortcuts added by the controller, except for the native arrow movement. Check Global Settings keybindings for conflicting actions. Remove or reassign a Tab-based character switch, for example, if Tab is going to mean Inventory.

The original games did not all use exactly the same controls on every platform. This guide uses familiar conventions rather than claiming one universal Sierra keyboard layout. Space is reserved for command focus here, while F3 provides recall.

38. Make buttons and typed commands share rules

A text command, a clicked object and a visual-script node should all be able to reach the same puzzle rule. Otherwise you can fix a bug in one route and leave it in the other two.

1. Select the note’s Look response in Verb Actions.
2. Remove the temporary Show Message test if it is still present.
3. Add Call Script Function → Run Adventure Command from Adventure Controller.
4. Set command to read note.
5. Test the typed command and a Look click. Both should show the same reply.

Object response	Command parameter
Room key → Take	take key
Room key → Look	look key
Note → Look / Read	read note
Door → Open	open door
Radio → Look	look radio
Radio → Use	turn on radio

For an inventory key-on-door rule, call send with use key on door. For a fuse-on-radio rule, call use fuse on radio. The controller still verifies ownership and room state, even when the call came from a visual interface.

> Do not leave the old Add Inventory, hide-model and score actions alongside the controller call. The shared function already performs the successful puzzle changes.

39. Call the same command from a visual graph

Open the Visual Script Designer and add a Call Script Function node. Choose Run Adventure Command, then set its command parameter. Connect the node to the flow where you want the command to run.

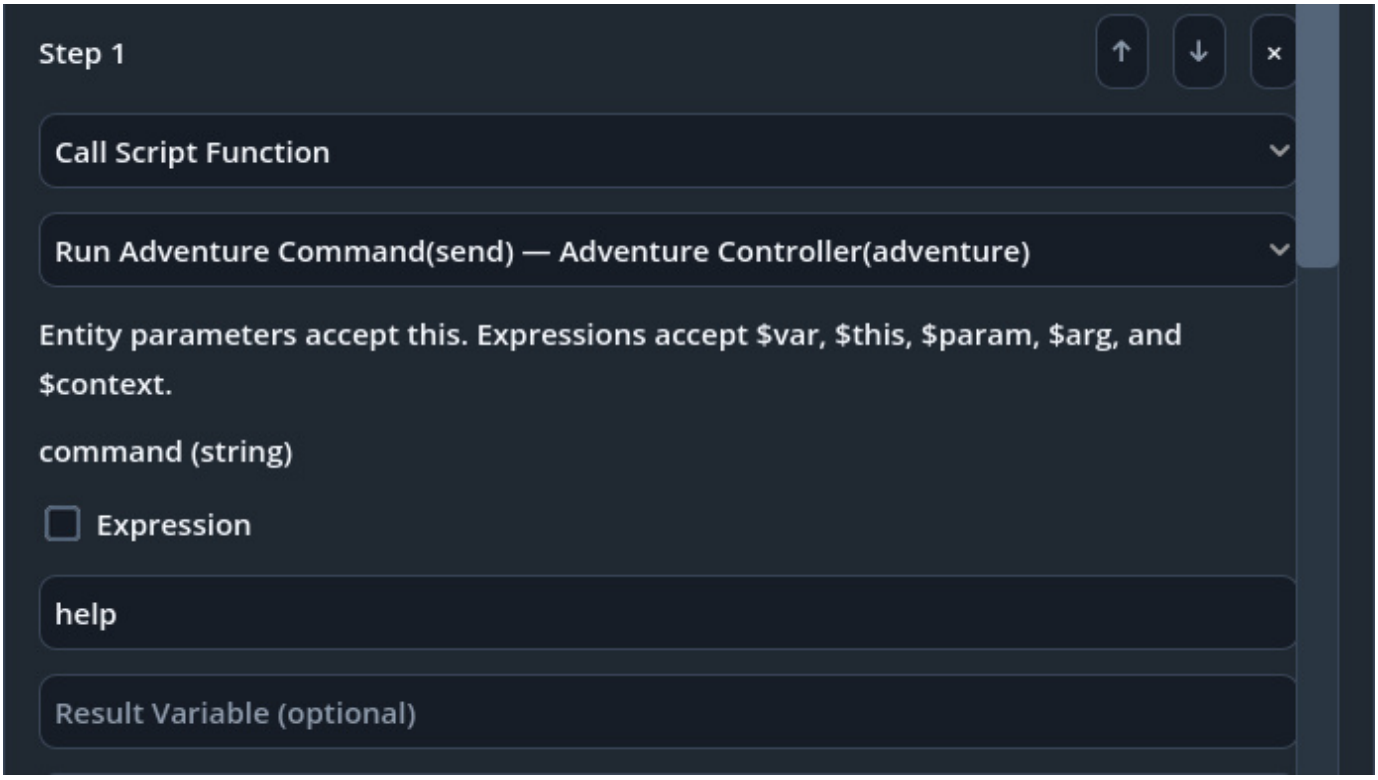

Figure 25. The same exposed function is available through GUI event action lists. The command is an editable parameter.

For example, a Help button can call help. A shortcut icon can call inventory. A room exit trigger can call north, provided the trigger is intended to use the same open-door check.

Use the function for a player-like command. If you are authoring a cinematic that must force entry regardless of the lock, use a separate, clearly named scene-change action. Do not quietly weaken the normal NORTH rule to accommodate a cutscene.

Calling it from another script

```
await CallScript("adventure", "send", ["read note"])
```

This uses the same function and passes one String argument. Other supported scripting languages can call through PCA's normal script/function bridge; use their own syntax rather than pasting GDScript into a Lua or BASIC editor.

40. Add a new noun without guessing

Suppose you add a bell to the yard. First decide what the player can learn or change. “A bell is here” is not a rule. “Ringing the bell calls the operator” is a rule.

1. Create the bell room object. Give it a stable ID and a readable name.
2. Add bell and signal bell to the native record’s synonyms.
3. In Appendix A’s NOUNS table, map those phrases to bell.
4. Extend present() so bell is present in the yard.
5. Add a Look description and a response for the action you want.
6. If the action changes the story, add a saved variable and test both its before and after states.

```
"bell":"bell", "signal bell":"bell"
```

That line belongs inside the existing NOUNS dictionary, with the correct comma between entries. It is not a complete script. Add one noun at a time and test it with a sensible command, a repeated command and a command from the wrong room.

> With the controller active, editing only the native Synonyms field is not enough. The controller uses its own NOUNS and VERBS tables. The native parser and the controller have separate vocabulary sources by design.

41. Add a new verb and its failure replies

A new verb needs more than a word in a dropdown. If you add RING, decide which objects can be rung and how the game should respond when the player tries the verb on something else.

1. Create a native Ring verb if you also want it in clickable interfaces.
2. Add ring → ring to the controller’s VERBS dictionary.
3. In act(), add the successful ring + bell branch.
4. Leave a useful fallback for RING DOOR, RING RADIO and a missing target.
5. Test it in both presentations and through any action-list call.

For longer verb phrases, normalize the phrase before splitting the command into a verb and target. The controller already does this for PICK UP: it converts PICK UP KEY into TAKE KEY. That is why it can recognize a two-word verb without treating UP as part of the object name.

Player input	Helpful response
RING	What do you want to ring?
RING BELL	The bell sounds; the appropriate state changes.
RING RADIO	The radio has a switch, not a bell.
RING UNKNOWN OBJECT	A clear missing-target reply.

The exact wording in this extension is a design example. The shipped controller accepts the commands printed in its vocabulary and command reference; it does not already contain the bell extension.

42. Add containers, pronouns and more rooms carefully

Once the little game works, you can extend it. Keep each new grammar feature explicit so a sentence cannot accidentally trigger a different puzzle.

Feature	State and parsing you need
LOOK IN BOX	A container-open flag, contents list and a grammar branch that preserves IN.
PUT FUSE IN BOX	A two-object rule with IN; checks for carried source and accessible container.
USE IT ON DOOR	A saved or session last-noun value; a rule to resolve IT before validating ownership.
Which key?	A candidate list, a pending question and a way to accept the answer as the next input.
More rooms	An exit map with explicit destination/spawn and optional conditions.
Several commands in one line	A queue and a rule that stops processing when a reply, room change or failure requires attention.

Do not reuse the ignored-word list to erase IN and then try to infer a container action afterward. Preserve meaningful separators until the grammar has used them.

The example handles one command at a time. It rejects or fails to match extra phrases instead of promising general natural-language understanding. Keep that limit in the game’s HELP text so players know how to talk to it.

43. Give it the look of an older adventure

The controls and puzzle structure do most of the work. To push the presentation toward AGI or early SCI, use a fixed view, compact status line, clear command prompt, modest palette and a readable bitmap-style font.

Choice	AGI-leaning presentation	Early SCI-leaning presentation
Room artwork	Bold shapes and a small palette.	More detailed painted or pixel-art rooms.
Character	Simple sprite with clear facing directions.	More detailed animated sprite.
Interface	Status strip, command line, simple reply box.	Similar typed-command structure, with richer windows and menus.
Text	Short, direct observations.	More room for description and character voice.

Treat these as art directions, not strict technical specifications. The sample’s 3D blockout and bundled models prove the gameplay first. Replacing them with your own 2D or 2.5D artwork is a separate presentation step.

For pixel art, use crisp source images and suitable nearest-neighbor filtering. Do not enlarge a tiny image with smoothing and then try to recover its edges with a screen effect. A pixel shader can be a finishing touch, but it does not create well-drawn pixel art.

> If you change the project’s logical resolution, rebuild the GUI coordinates to suit it. The 1280 × 720 tables in this guide will not fit unchanged into a small 4:3 canvas.

44. Add motion, sound and typewriter text

Keep feedback simple and deliberate. A key sound on unlock, a click when the fuse fits and a radio crackle at the ending can make a small puzzle feel finished without covering every response in noise.

Add sound or animation calls inside the same successful branches that change puzzle state. Put the unlock sound after the “already unlocked” guard so repeated USE commands do not play it as if the puzzle were solved again.

Effect	Where it belongs
Door opening animation	After the door-open state changes; keep the model’s final pose consistent with the saved flag.
Player talk / gesture	A deliberate spoken or animated response, not every parser error.
Radio light	Driven by the fitted/signal state; refresh it after restore.
Typing sound	Optional, quiet and adjustable; avoid one loud sound per visible letter.

PCA’s Show Message and dialogue actions have their own typewriter settings. The transcript and reply Label in this controller are direct text displays; checking a message action elsewhere will not animate them automatically. Add a deliberate reveal routine if you want typewriter text in this custom screen.

> If you add letter-by-letter reveal, define two Enter states: the first press reveals the rest, and the next dismisses the message. Do not let a single press both reveal and close a reply before it can be read.

45. Write replies that help the player think

Parser games are conversations. A failed command is a chance to tell the player what the game understood. It should not feel like a spelling exam with scenery.

Situation	Weak reply	More useful reply
Wrong verb	No.	The radio has a switch. Try TURN ON RADIO.
Missing inventory	You can’t.	You aren’t carrying the key.
Wrong room	Not possible.	The radio is inside the cabin.
Already solved	Nothing happens.	A fuse is already fitted.
Unknown input	Error.	I don’t know that command. Type HELP for examples.

Name important things in the prose. If you describe “a small brass key,” accept KEY and BRASS KEY. If the only accepted phrase is the internal ID yard_key_03, the player is solving your database rather than your puzzle.

Use short paragraphs and clear nouns. Put the exit direction near the end of a room description. Let LOOK repeat the important information. Give successful commands a visible consequence and an honest next step.

A little personality is welcome. Save the joke for a harmless attempt; do not bury the only clue in three paragraphs of sarcasm. The player should be able to tell the difference between “wrong idea” and “right idea, wrong wording.”

46. Test the game as an unfamiliar player

Test	Expected result
Mixed case and polite filler	Please, TAKE the brass key! takes one key.
Repeated pickup	GET KEY again adds no item and no score.
Use before pickup	USE KEY ON DOOR explains missing ownership.
Wrong-room noun	TALK TO MORGAN in the cabin is rejected.
Direction before opening	NORTH cannot pass a closed door.
Drop and recover	The item is visible in the correct room and can be taken again.
Fuse consumption	The fitted fuse disappears from inventory.
Repeated win	TURN ON RADIO cannot raise score above 10.
Restore after changes	Room, inventory, flags and transcript return to saved state.
Presentation switch	The current puzzle continues; it does not restart.
Enter in reply	Closes the reply once.
Typing a direction word	The actor does not walk because letters were typed.

Have someone try the game without your walkthrough. Write down the words they use. Add reasonable synonyms or clearer prose when those attempts make sense. Keep the test session separate from the authoring session so you do not explain every puzzle before the person has a chance to play.

47. Fix input and GUI problems first

Symptom	Check
Enter does nothing	The script is enabled and Global; both field IDs match; compilation succeeded; the input is single-line.
Every command runs twice	Turn off Run Text Parser Command When Enter Is Pressed on controller fields; remove duplicate Enter events.
Only the native test field works	Hide the temporary parser GUI and check that the controller fields use their exact IDs.
Arrow keys edit the line	The command field still has focus. Press Escape, or dismiss the reply.
Arrow keys never move the actor	Check graphical mode, navigation, keyboard movement, player input and any open modal GUI.
A first letter is missing or doubled	Ensure only the controller handles type-to-open; remove another auto-focus handler.
Text reader looks dim	Use the read-only font color override in Appendix A.
Reply text is cut off	Make the Label taller, reduce font size or write a shorter reply; test your longest translation.
Space opens the wrong field	Turn off native Space Focuses Assigned Text Parser for the controller, and hide extra parser receivers.

Work through the actual input route: key press → focused field → send → execute_command → reply. Inspect one step at a time. Changing colors or adding more buttons will not repair a missing submission connection.

48. Fix story and save-state problems

Symptom	Check
Key can be used before pickup	The successful branch must check real inventory, not only a known noun.
Item returns after changing rooms	Check ta_key_room / ta_fuse_room and the room placeholder IDs.
Score grows on repetition	Award through a unique guard flag; remove duplicate action-list rewards.
Door looks open but NORTH fails	The route checks ta_door_open, not the model’s appearance.
A saved game freezes during load	Keep RESTORE deferred. Loading rebuilds script instances; do not await it inside the instance being replaced.
Old code reappears after restore	Start a new game and make a fresh save after changing project scripts.
Native preview disagrees with controller	They use separate vocabulary and cleanup routes. Update and test the intended route.
Controller rejects a new noun	Add it to NOUNS, present(), descriptions and the relevant action rule.
A new room is unreachable	Add an explicit route and a valid destination spawn.

Story state should be the authority. The visible door, loose fuse and status strip should follow that state. If you make the picture the only record of progress, room changes and restore will eventually expose the mismatch.

49. Finish a release players can understand

1. Run the complete winning route with Play / Restart, not only Test Scene.
2. Test a fresh game, a mid-puzzle save and a completed-game save.
3. Test the exported game as well as the editor preview. Check file access, keyboard focus and the first screen.
4. Make HELP explain accepted commands, movement, recall and save-slot behavior.
5. Hide the temporary native parser test GUI and remove debugging controls.
6. Add your opening menu, options, credits and a clear way to quit.
7. Check required notices for any fonts, sounds, art or bundled models used in your own release.

For a pure text release, you can leave ta_text_mode true and remove the presentation-switch buttons. Keep player movement disabled. For a graphical release, start with false and retain the text view only if you want it as an alternate presentation.

Before adding more rooms, make a small list of what the player can know, carry and change. Keep the vocabulary consistent with that list. A short adventure with clear rules is a better foundation than twenty rooms full of commands that almost work.

> The controller is deliberately small: two rooms, a limited grammar, fixed drop positions and one quick-save slot. It is a complete working example of those features, not a replacement for every part of a full interactive-fiction parser.

Appendix A. Install the complete controller

The following pages contain the complete Adventure Controller in order. Paste every code block into one GDScript record with ID adventure. Do not paste page headings, line-range captions or the explanatory paragraphs.

Indentation is part of GDScript. Preserve the four spaces shown inside each function or branch. If copying from a PDF changes quotation marks, indentation or a backslash, correct it in the Script Designer before running the game.

Before Play	Required setup
Script	Enabled and Global; event subscription OnSaveLoaded.
Scenes	lab and cabin; both have entry spawns.
Inventory	brass_key and fuse, initially not carried.
Variables	All ta_ variables from lessons 16-17.
GUI IDs	text_screen, graph_hud, reply_card.
Control IDs	story_transcript, text_command, graph_command.
Reply Label	Text is $$ta_reply.
Graphical input	Arrow Keys + Sierra AGI movement; native Space focus off.

RESTORE uses a deferred runtime call because loading replaces script instances. Keep the OnSaveLoaded handler at the end of the script. The variables hold persistent story data; pending_reply, busy and display signatures are temporary control state.

A1. Controller source: lines 1-35

Vocabulary, temporary fields and the global startup hook.

```
extends PCAScriptBase

const NOUNS: Dictionary = {
    "key":"key", "brass key":"key", "pocket key":"key",
    "fuse":"fuse", "spare fuse":"fuse",
    "door":"door", "cabin door":"door",
    "note":"note", "notice":"note", "paper":"note",
    "radio":"radio", "transmitter":"radio", "wireless":"radio",
    "morgan":"morgan", "operator":"morgan",
    "station operator":"morgan"
}
const VERBS: Dictionary = {
    "look":"look", "examine":"look", "inspect":"look",
    "read":"read", "take":"take", "get":"take",
    "drop":"drop", "open":"open", "close":"close",
    "use":"use", "apply":"use", "talk":"talk",
    "speak":"talk", "ask":"ask"
}
var pending_reply: bool = false
var busy: bool = false
var world_signature: String = ""
var screen_signature: String = ""
var confirm_restart: bool = false

func on_global_ready(_event: Dictionary = {}) -> void:
    Runtime.variables["ta_score"] = int(Runtime.variables.ta_score)
    set_process(true)
    set_process_input(true)

func control(element_id: String) -> Control:
    return Runtime.authored_gui_manager.element_nodes.get(
        element_id) as Control

func _process(_delta: float) -> void:
    for element_id: String in ["text_command", "graph_command"]:
```

Continue directly with the next source block.

A2. Controller source: lines 36-70

Finding live GUI controls and maintaining the transcript reader.

```
        var field: LineEdit = control(element_id) as LineEdit
        if is_instance_valid(field):
            if not field.text_submitted.is_connected(send):
                field.text_submitted.connect(send)
    var reader: TextEdit = control("story_transcript") as TextEdit
    if is_instance_valid(reader):
        reader.editable = false
        reader.add_theme_color_override("font_readonly_color",
            Color("ecdfbf"))
        var body: String = "\n\n".join(
            Runtime.variables.get("ta_lines", []))
        if reader.text != body:
            reader.text = body
            reader.set_deferred("scroll_vertical",
                reader.get_line_count())
    sync_view()
    sync_world()

func text_mode() -> bool:
    return bool(Runtime.variables.get("ta_text_mode", true))

func sync_view() -> void:
    var signature: String = str(text_mode())
    if signature == screen_signature:
        return
    screen_signature = signature
    Runtime.set_player_input_enabled(not text_mode())
    Runtime.hide_gui("reply_card")
    pending_reply = false
    Runtime.show_gui("text_screen" if text_mode() else "graph_hud")
    Runtime.hide_gui("graph_hud" if text_mode() else "text_screen")
    if text_mode():
        call_deferred("focus_command")

func sync_world() -> void:
```

Continue directly with the next source block.

A3. Controller source: lines 71-105

Synchronizing the loose-item models with story state.

```
    if not Runtime.ready_for_input:
        return
    var signature: String = str([
        Runtime.scene_id, Runtime.entity_nodes.size(),
        Runtime.variables.get("ta_key_room"),
        Runtime.variables.get("ta_fuse_room"),
        Runtime.variables.get("ta_door_open")])
    if signature == world_signature:
        return
    world_signature = signature
    for item: String in ["key", "fuse"]:
        for room: String in ["lab", "cabin"]:
            Runtime.set_entity_visible(room + "_" + item,
                str(Runtime.variables.get("ta_" + item + "_room"))
                == room)
    Runtime.set_entity_visible("cabin_door",
        not bool(Runtime.variables.get("ta_door_open", false)))

func focus_command() -> void:
    var element_id: String = (
        "text_command" if text_mode() else "graph_command")
    var field: LineEdit = control(element_id) as LineEdit
    if is_instance_valid(field) and field.is_visible_in_tree():
        field.grab_focus()
        field.edit()

func _input(event: InputEvent) -> void:
    if not Runtime.active_system_gui_id.is_empty():
        return
    if Runtime.system_menu != null and Runtime.system_menu.is_open():
        return
    var key: InputEventKey = event as InputEventKey
    if key == null or not key.pressed or key.echo or busy:
        return
    var code: int = int(key.keycode)
```

Continue directly with the next source block.

A4. Controller source: lines 106-140

Focus, reply dismissal and keyboard shortcuts.

```
    if pending_reply and code in [KEY_ENTER, KEY_ESCAPE]:
        dismiss_reply()
        get_viewport().set_input_as_handled()
        return
    var focused: Control = get_viewport().gui_get_focus_owner()
    var typing: bool = focused is LineEdit or focused is TextEdit
    if code == KEY_ESCAPE and typing:
        focused.release_focus()
        get_viewport().set_input_as_handled()
    elif code == KEY_SPACE and not typing and not pending_reply:
        focus_command()
        get_viewport().set_input_as_handled()
    elif code == KEY_F3 and not pending_reply:
        focus_command()
        var field: LineEdit = control(
            "text_command" if text_mode() else "graph_command")
        field.text = str(Runtime.variables.get("ta_last", ""))
        field.caret_column = field.text.length()
        get_viewport().set_input_as_handled()
    elif not typing and not pending_reply:
        var shortcuts: Dictionary = {
            KEY_F1:"help", KEY_F5:"save", KEY_F7:"restore",
            KEY_TAB:"inventory"
        }
        if shortcuts.has(code):
            get_viewport().set_input_as_handled()
            send(str(shortcuts[code]))
        elif key.unicode >= 32 and not text_mode():
            if key.ctrl_pressed or key.alt_pressed or key.meta_pressed:
                return
            focus_command()
            var field: LineEdit = control("graph_command")
            field.insert_text_at_caret(String.chr(key.unicode))
            get_viewport().set_input_as_handled()
```

Continue directly with the next source block.

A5. Controller source: lines 141-175

Command submission and text cleanup.

```
func dismiss_reply() -> void:
    pending_reply = false
    Runtime.hide_gui("reply_card")
    var focused: Control = get_viewport().gui_get_focus_owner()
    if focused != null:
        focused.release_focus()
    if text_mode():
        focus_command()

func presentation() -> void:
    Runtime.variables["ta_text_mode"] = not text_mode()
    screen_signature = ""
    sync_view()
    Runtime.state_changed()

func run() -> void:
    await send(str(context.get("command", "help")))

func send(command: String) -> void:
    if busy or pending_reply:
        return
    var cleaned: String = clean(command)
    if cleaned.is_empty():
        return
    busy = true
    Runtime.variables["ta_last"] = command
    for element_id: String in ["text_command", "graph_command"]:
        var field: LineEdit = control(element_id) as LineEdit
        if is_instance_valid(field):
            field.clear()
            if not text_mode():
                field.release_focus()
    append_line("> " + command.strip_edges())
    await execute_command(cleaned)
    Runtime.state_changed()
```

Continue directly with the next source block.

A6. Controller source: lines 176-210

Transcript output, one-time scoring and object availability.

```
    busy = false
    if text_mode():
        focus_command()

func clean(raw: String) -> String:
    var value: String = raw.to_lower().strip_edges()
    for mark: String in [".", ",", "!", "?", ":", ";"]:
        value = value.replace(mark, " ")
    var words: PackedStringArray = []
    for word: String in value.split(" ", false):
        if word not in ["a", "an", "the", "please"]:
            words.append(word)
    return " ".join(words)

func append_line(text: String) -> void:
    var lines: Array = Runtime.variables.get("ta_lines", [])
    lines.append(text)
    while lines.size() > 100:
        lines.pop_front()
    Runtime.variables["ta_lines"] = lines

func reply(text: String) -> void:
    append_line(text)
    Runtime.variables["ta_reply"] = text
    if not text_mode():
        Runtime.show_gui("reply_card")
        pending_reply = true

func award(flag: String, points: int) -> void:
    if bool(Runtime.variables.get(flag, false)):
        return
    Runtime.variables[flag] = true
    Runtime.variables["ta_score"] = int(
        Runtime.variables.get("ta_score", 0)) + points
```

Continue directly with the next source block.

A7. Controller source: lines 211-245

Room descriptions and the first command branches.

```
func present(noun: String) -> bool:
    if noun in ["key", "fuse"]:
        var place: String = str(Runtime.variables.get(
            "ta_" + noun + "_room", ""))
        return place == Runtime.scene_id or place == "carried"
    if noun in ["door", "note", "morgan"]:
        return Runtime.scene_id == "lab"
    return noun == "radio" and Runtime.scene_id == "cabin"

func describe_room() -> String:
    var loose: String = ""
    for item: String in ["key", "fuse"]:
        if Runtime.variables.get("ta_" + item + "_room") == Runtime.scene_id:
            loose += " There is a " + item + " here."
    if Runtime.scene_id == "lab":
        return ("STATION YARD\nRain drums on the old signal cabin. "
            + "A note is pinned beside its door. Morgan waits by "
            + "the bench. The cabin is north." + loose)
    return ("SIGNAL CABIN\nThe transmitter waits on the table. "
        + ("Its fuse is fitted. " if Runtime.variables.ta_fuse_fitted
        else "Its fuse holder is empty. ")
        + "The yard is south." + loose)

func execute_command(command: String) -> void:
    match command:
        "help":
            reply("Try LOOK, LOOK KEY, TAKE KEY, INVENTORY, "
                + "USE KEY ON DOOR, OPEN DOOR, NORTH, SOUTH, "
                + "TALK TO MARA or ASK MARA ABOUT RADIO. "
                + "SAVE uses slot 1. RESTORE loads it.")
            return
        "look", "l", "look around":
            reply(describe_room())
            return
        "inventory", "i":
```

Continue directly with the next source block.

A8. Controller source: lines 246-280

Save, restore, restart and radio commands.

```
            var names: PackedStringArray = []
            for item: String in Runtime.inventory:
                names.append(Runtime.project.get_inventory(item)
                    .get("name", item))
            reply("You are carrying: " + (
                ", ".join(names) if not names.is_empty()
                else "nothing" ) + ".")
            return
        "north", "n", "go north", "south", "s", "go south":
            await travel(command)
            return
        "save", "save game":
            var ok: bool = Runtime.save_game(1, "The Last Signal")
            reply("Saved in slot 1." if ok else "Save failed.")
            return
        "restore", "restore game", "load", "load game":
            if Runtime.save_slot_exists(1):
                Runtime.call_deferred("load_game", 1)
            else:
                reply("No save in slot 1.")
            return
        "restart":
            confirm_restart = true
            reply("Type RESTART YES to begin again. "
                + "Your saved game will remain.")
            return
        "restart yes":
            if confirm_restart:
                Runtime.call_deferred("start_new_game")
            else:
                reply("Type RESTART first.")
            return
        "turn on radio", "switch on radio":
            if Runtime.scene_id != "cabin":
                reply("The radio is inside the cabin.")
```

Continue directly with the next source block.

A9. Controller source: lines 281-315

Grammar routing from verbs and noun phrases.

```
            elif not bool(Runtime.variables.ta_fuse_fitted):
                reply("The switch clicks. Nothing happens. "
                    + "The fuse holder is empty.")
            else:
                award("ta_signal_sent", 3)
                reply("The dial glows. A voice answers through "
                    + "the rain. Your warning has reached the "
                    + "next station. THE END. Score: 10 of 10.")
            return
    if command.begins_with("ask "):
        if command == "ask morgan about radio" and present("morgan"):
            reply("Morgan says, 'The spare fuse is in the cabin. "
                + "Fit it, then turn the radio on.'")
        else:
            reply("Try ASK MARA ABOUT RADIO while in the yard.")
        return
    if command.begins_with("pick up "):
        command = "take " + command.substr(8)
    var words: PackedStringArray = command.split(" ", false)
    var verb: String = str(VERBS.get(words[0], ""))
    if verb.is_empty():
        reply("I don't know that command. Type HELP for examples.")
        return
    var target: String = " ".join(words.slice(1))
    if target.begins_with("at ") or target.begins_with("to "):
        target = target.substr(3)
    if verb == "use" and target.contains(" on "):
        var pair: PackedStringArray = target.split(" on ")
        if pair.size() != 2:
            reply("Use one item on one target at a time.")
            return
        use_pair(str(NOUNS.get(pair[0], "")),
            str(NOUNS.get(pair[1], "")))
        return
    var noun: String = str(NOUNS.get(target, ""))
```

Continue directly with the next source block.

A10. Controller source: lines 316-350

Object descriptions, pickups, drops and the door.

```
    if noun.is_empty():
        reply("Please name one thing: for example, LOOK KEY.")
    elif not present(noun):
        reply("You can't see that here.")
    else:
        act(verb, noun)

func act(verb: String, noun: String) -> void:
    if verb in ["look", "read"]:
        var descriptions: Dictionary = {
            "key":"A small brass key. It should fit the cabin door.",
            "fuse":"A spare fuse, just right for the transmitter.",
            "door":"A locked wooden door leading north.",
            "note":"The note reads: 'Morgan has left the key on "
                + "the bench. Spare fuse inside. Send the warning.'",
            "radio":"An old transmitter with an empty fuse holder.",
            "morgan":"Morgan, the station operator, looks worried."
        }
        if noun == "door" and bool(Runtime.variables.ta_unlocked):
            reply("The cabin door is unlocked.")
        elif noun == "radio" and bool(Runtime.variables.ta_fuse_fitted):
            reply("The fuse is fitted. The radio is ready to switch on.")
        else:
            reply(str(descriptions[noun]))
    elif verb in ["take", "drop"] and noun in ["key", "fuse"]:
        var item_id: String = "brass_key" if noun == "key" else "fuse"
        if verb == "take":
            if HasInventory(item_id):
                reply("You already have it.")
                return
            Runtime.add_inventory(item_id)
            Runtime.variables["ta_" + noun + "_room"] = "carried"
            award("ta_found_" + noun, 1)
            reply("Taken.")
        elif HasInventory(item_id):
```

Continue directly with the next source block.

A11. Controller source: lines 351-385

Item-on-target puzzle rules.

```
            Runtime.remove_inventory(item_id)
            Runtime.variables["ta_" + noun + "_room"] = Runtime.scene_id
            reply("Dropped. You can take it again here.")
        else:
            reply("You aren't carrying it.")
    elif noun == "door" and verb in ["open", "close"]:
        if not bool(Runtime.variables.ta_unlocked):
            reply("It's locked. You need the key.")
        else:
            Runtime.variables.ta_door_open = verb == "open"
            reply("The door is now " + (
                "open." if verb == "open" else "closed."))
    elif noun == "morgan" and verb == "talk":
        reply("Morgan says, 'Send the warning before the next "
            + "train arrives. Read the note beside the door.'")
    else:
        reply("That won't help. Try LOOK, or be more specific.")

func use_pair(source: String, target: String) -> void:
    if source.is_empty() or target.is_empty():
        reply("Name both things: USE KEY ON DOOR.")
    elif not present(target):
        reply("You can't see that target here.")
    elif source == "key" and target == "door":
        if not HasInventory("brass_key"):
            reply("You aren't carrying the key.")
        elif bool(Runtime.variables.ta_unlocked):
            reply("The door is already unlocked.")
        else:
            award("ta_unlocked", 2)
            reply("The key turns. The door is unlocked. Now open it.")
    elif source == "fuse" and target == "radio":
        if bool(Runtime.variables.ta_fuse_fitted):
            reply("A fuse is already fitted.")
        elif not HasInventory("fuse"):
```

Continue directly with the next source block.

A12. Controller source: lines 386-415

Room travel and the post-restore display refresh.

```
            reply("You aren't carrying the fuse.")
        else:
            Runtime.remove_inventory("fuse")
            Runtime.variables.ta_fuse_room = "fitted"
            award("ta_fuse_fitted", 3)
            reply("The fuse clips into place. Try TURN ON RADIO.")
    else:
        reply("Those two things don't work together.")

func travel(command: String) -> void:
    var north: bool = command in ["north", "n", "go north"]
    if north and Runtime.scene_id == "lab":
        if not bool(Runtime.variables.ta_door_open):
            reply("The cabin door is closed.")
            return
        await Runtime.change_scene("cabin", "entry")
    elif not north and Runtime.scene_id == "cabin":
        await Runtime.change_scene("lab", "entry")
    else:
        reply("There is no exit that way.")
        return
    reply(describe_room())

func on_save_loaded(_event: Dictionary = {}) -> void:
    pending_reply = false
    screen_signature = ""
    world_signature = ""
    sync_view()
    reply("Restored slot 1.")
    Runtime.state_changed()
```

End of the complete Adventure Controller. Apply the script and test with Play / Restart.

Appendix B. Command reference

Command	Accepted forms / notes
Look around	LOOK, L, LOOK AROUND
Examine one object	LOOK, EXAMINE or INSPECT + noun; optional AT.
Read	READ + noun; use READ NOTE for the clue.
Take	TAKE, GET or PICK UP + key / fuse.
Drop	DROP + carried key / fuse.
Inventory	INVENTORY or I.
Open / close	OPEN DOOR or CLOSE DOOR. The door must be unlocked.
Use an item	USE or APPLY + source ON target.
Talk	TALK or SPEAK + MORGAN / OPERATOR; optional TO.
Ask topic	ASK MORGAN ABOUT RADIO. This is one explicit topic phrase.
Switch on	TURN ON RADIO or SWITCH ON RADIO.
Travel	N, NORTH, GO NORTH; S, SOUTH, GO SOUTH.
Help	HELP.
Save	SAVE or SAVE GAME. Slot 1 is replaced.
Restore	RESTORE, RESTORE GAME, LOAD or LOAD GAME.
Restart	RESTART, then RESTART YES.

Ordinary noun aliases include key / brass key / pocket key; fuse / spare fuse; door / cabin door; note / notice / paper; radio / transmitter / wireless; Morgan / operator / station operator. The exact ASK phrase and radio-switch phrases have their own branches.

> The cleaner ignores a, an, the and please, plus common punctuation. It preserves ON and ABOUT. This reference describes the controller, not every phrase accepted by the native parser.

Appendix C. Where to change what

Change you want	Edit this
Opening story	Initial ta_lines Array variable.
A room’s repeated description	describe_room() in the controller.
Native LOOK description	Scene Designer → Room Description.
Accepted ordinary nouns	NOUNS, present() and act().
Accepted ordinary verbs	VERBS and act(); normalize any multiword phrase.
An exact special command	execute_command().
Key/fuse puzzle conditions	use_pair() and real PCA inventory.
Exit routes	travel(), plus scene/spawn records.
Scoring	award() calls and their saved guard variables.
Text appearance	Text Adventure Screen GUI and reader color override.
Graphical appearance	Graphical Parser HUD and Adventure Reply GUIs.
Keyboard shortcuts	_input(); check Global Settings conflicts too.
Clickable interactions	Verb Actions → Call Script Function → send(command).
Native parser vocabulary	Verb/object/inventory synonyms and Text Parser Designer.

Keep the game’s HELP response and command reference up to date as you change the grammar. A player cannot use a feature that only exists in your source comments.

References and verification notes

Editor instructions and code examples were checked against PCA 3D Adventure Studio 0.86.3.4. Screenshots show the actual editor and native Play view. Room geometry is a simple original blockout; the graphical figures also use characters from the bundled Clockmaker content.

The completed example was exercised with real command-field submission, wrong-room and missing-item attempts, repeated rewards, room travel, item consumption, save/restore, presentation switching, continuous movement, reply dismissal, automatic command focus, recall and quick save.

Sierra On-Line: King’s Quest IV manual

Original game manual, preserved by the Museum of Computer Adventure Game History. Consult the playing instructions and introductory walkthrough for the typed-command approach, observation, inventory puzzles and saving.

Sierra On-Line: King’s Quest IV UK manual

Original manual and reference instructions for command entry, menus and keyboard conventions. The project shortcuts in this guide are explicitly listed; they are not assumed engine defaults.

The older games are references for interaction style. This guide does not reproduce their artwork, dialogue or game data. It does not provide AGI or SCI binary compatibility.

> For a new PCA release, confirm UI labels and runtime APIs before reusing the controller unchanged. The guide identifies the version it was tested with so that later behavior can be compared sensibly.