PCA 3D ADVENTURE STUDIO

Replace the Built-in Screens

HUDs, menus, splash screens and personal character interfaces.
A complete guide to the design, assignments and working behavior.

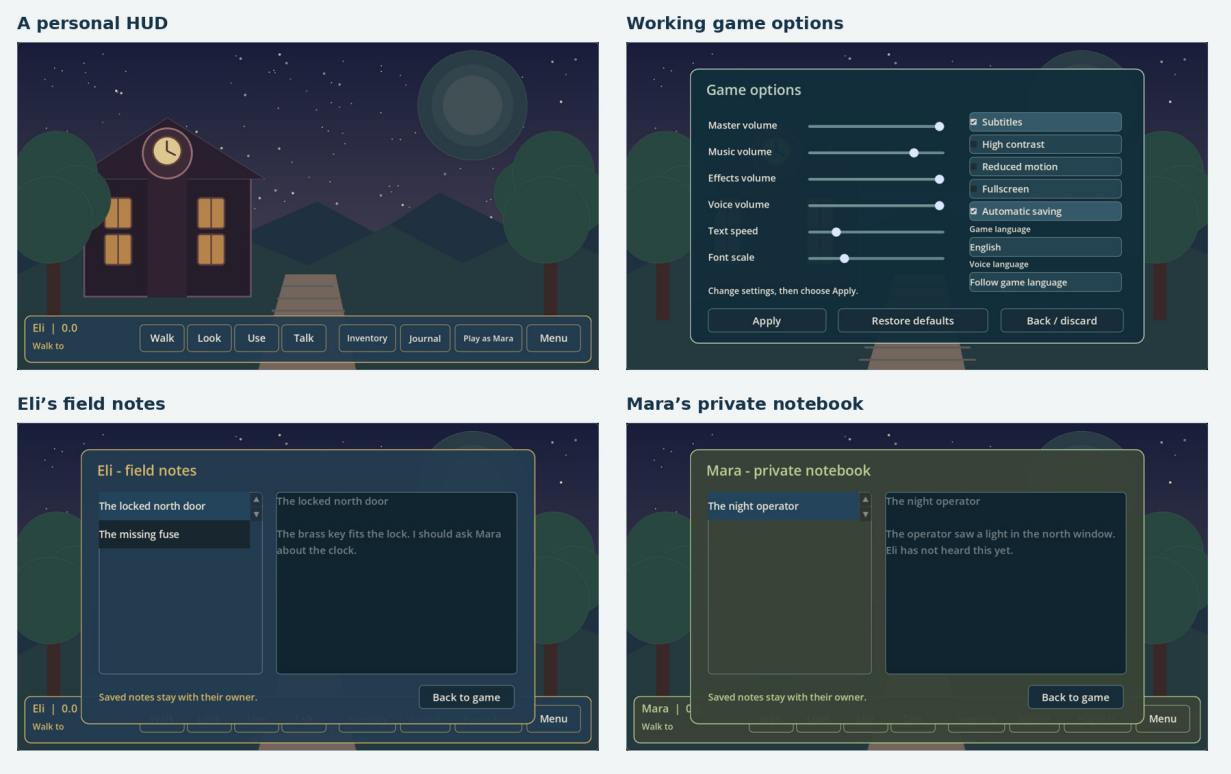

Figure 1. Personal HUDs and notebooks, plus a replacement game-options screen, running in PCA.

Character-specific HUDs · Separate inventories and saved notes
Main Menu · Pause · Save / Load · Options · Speech · Startup splashes

Reference build 0.86.3.4 | Windows, macOS and Linux

Before you begin

This guide takes you through replacing the built-in game screens in PCA 3D Adventure Studio. It explains what to draw, where to assign it and how to make its controls operate the real game systems.

The running example uses Eli and Mara. Each has a personal HUD, inventory, notebook and speech shell. They share the title menu, pause menu, options and save system. Use your own game and assets; no practice files or companion download are needed.

Every editor and Play image is an actual PCA capture. The notebook and options helpers are complete project-script examples. The engine code is not modified.

Read	For
Chapters 1-3	System assignments, replacing the HUD, playable characters and personal inventories.
Chapter 4	A shared quest journal versus genuinely separate saved character notes.
Chapters 5-6	Main/Pause/Save/Load routes and a working custom options menu.
Chapters 7-8	Speech and choice Groups, splash media, timed GUI splashes and the early boot image.
Chapters 9-11	Presentation, complete script helpers and the final test pass.

Make a project copy before replacing several screens. Keep a working plain layout until the new assignment and behavior have been tested.

Contents

Click a chapter title, or use the PDF bookmarks to jump to the individual steps and code listings.

1. Know which screen you are replacing 4

2. Replace the permanent HUD 11

3. Give each character their own screens 16

4. Give journals separate looks and contents 24

5. Replace the main menu and pause menu 32

6. Replace the game-options menu 40

7. Replace speech, dialogue and message screens 50

8. Replace splash screens and startup presentation 56

9. Finish the presentation without breaking the controls 64

10. Install the working script helpers 68

11. Run the complete replacement-screen test 83

1. Know which screen you are replacing

A GUI is a screen you draw in the GUI Designer. A system assignment tells the game when to use that GUI. You usually need both: a finished layout and the correct assignment.

For example, drawing an inventory window does not give every character a separate bag. The window controls the presentation; Inventory Mode controls ownership. The same distinction matters even more for journals.

You want to change	Start here
The permanent in-game controls	Create a GUI and assign Main HUD in Global Settings.
What one playable character sees	Character Designer → Player Settings → Personal GUI Surfaces.
The title, pause or options menu	Assign its system role, then wire its Buttons to real menu actions.
Spoken text and portraits	Assign the speech/dialogue role and add the special Group IDs.
Logos shown before the title menu	Project Settings → Startup Splashes; optionally add a timed Splash Screen GUI.
The image shown before PCA starts running the game	The exported Godot project’s boot splash, covered separately in Chapter 8.

This guide changes the game your players run. The PCA editor’s own toolbar, designers and project-opening screen are separate application interfaces.

Find Custom System Interfaces

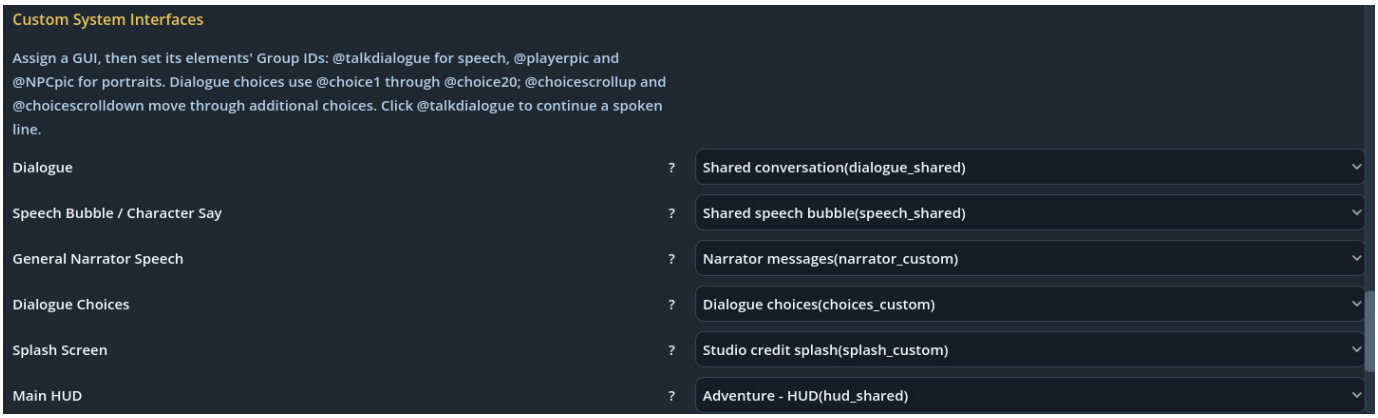

Figure 2. The upper assignments cover speech, choices, the timed GUI splash and the Main HUD.

1. Open Global Settings. In this build, the Custom System Interfaces section is in General; scroll down to it.
2. Find the role you want to replace. The dropdown shows the GUI’s name and its ID.
3. Select the new GUI. Do not type an element ID here: this setting needs the whole GUI record.
4. Click Apply Changes, then Save Project.
5. Stop an old Play session and start a fresh one so it uses the saved assignments.

Build and test one replacement at a time. First replace the HUD, then the bags and journals, then the menus and spoken-text screens. That makes it much easier to identify the setting responsible for a surprising result.

> The menu paths and field names in this book were checked in PCA 3D Adventure Studio 0.86.3.4. Some inspector sections share a tab rather than having a tab of their own.

The complete system-role list

Assignment	What it is used for
Dialogue	Conversation lines in a dialogue graph.
Speech Bubble / Character Say	The global presentation for character speech outside that graph.
General Narrator Speech	General messages shown with Show Message.
Dialogue Choices	The selectable responses; choice Groups connect these to the dialogue.
Splash Screen	One authored GUI shown for Splash Duration during normal startup.
Main HUD	The persistent gameplay controls.
Inventory Screen	The dedicated inventory opened through the inventory command.
Journal Screen	The dedicated journal opened through the journal command.
Pause Menu	The game’s in-play menu.
Main Menu	The title/start menu.
Save Screen / Load Screen	The system routes for storing and restoring positions.
Options Screen	The player’s settings screen.

You can point Save Screen and Load Screen at one combined GUI, or create two separate GUIs. The examples here use two screens so their headings, commands and Back routes remain clear.

> Personal character assignments cover HUD, inventory, journal and dialogue. Main Menu, Pause Menu, Save, Load and Options remain project-wide roles.

See the menu assignments together

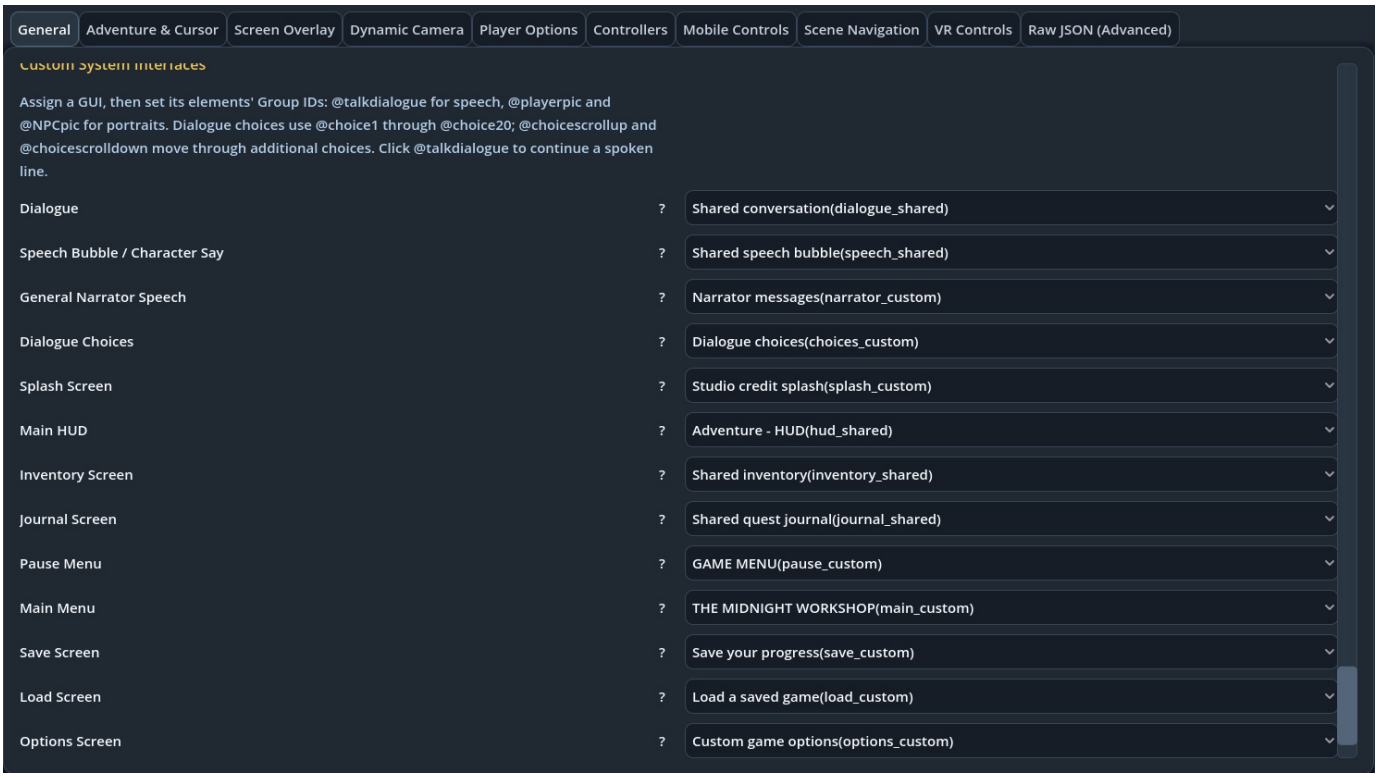

Figure 3. The lower part of Custom System Interfaces assigns inventory, journal and the game menus.

Use deliberate, readable names: “Game Menu(pause_custom)” is easier to recognize than a list of new_gui_12 entries. Once a script refers to an ID, changing the display name is easier than renaming the ID everywhere.

GUI ID in this book	Assigned role
main_custom / pause_custom	Main Menu / Pause Menu
save_custom / load_custom	Save Screen / Load Screen
options_custom	Options Screen
hud_shared / inventory_shared / journal_shared	Project-wide defaults for characters without personal replacements.

Build one clean replacement shell

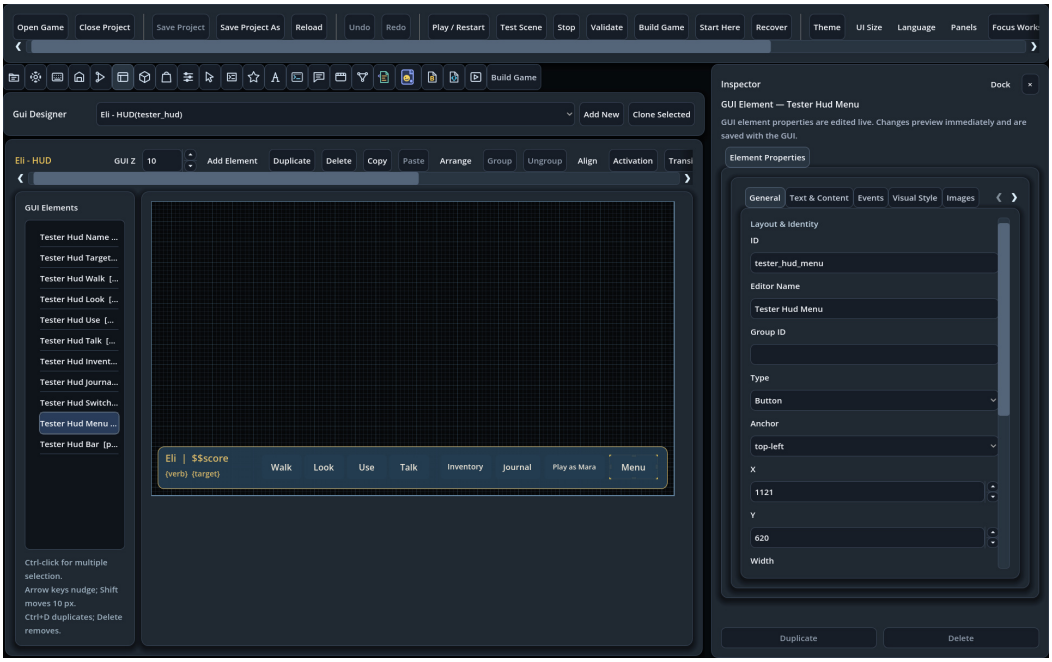

Figure 4. The example uses ordinary GUI elements; its system role is assigned separately.

1. Open the GUI Designer. Choose an existing screen and Clone Selected, or use Add New for a clean layout.
2. Give the new GUI a unique ID and a useful name. Keep the original until the replacement works.
3. Use the project’s design resolution. These examples use 1280 × 720.
4. Open GUI Window. Check the actual window position and size, not just the size of a background Panel.
5. For a screen built from your own Panels, turn off window title, border and background when they would duplicate your artwork.
6. Set sensible Z Index values: background first, Panels above it, then readable Labels and controls.

> A transparent full-screen picture can still sit over a control. If a Button stops responding after adding artwork, check which element is on top and which elements accept pointer input.

Choose how the GUI opens

Kind of GUI	Starting point
Main HUD	The system manages its visibility. Personal HUDs replace the project HUD.
Main / Pause / Save / Load / Options	Hidden initially; opened through the appropriate system route. Make the screen modal.
Personal inventory or journal	Hidden initially; use Toggle Inventory or Toggle Journal to open the active character’s assigned screen.
Dialogue, choices and narrator	Hidden initially; the speech/dialogue/message system shows them.
Ordinary extra screen	Use Show GUI / Hide GUI, or an intentional automatic activation rule.

Modal means that the GUI takes input while it is open. It does not promise that every sound, clock or animation in the world stops. Test movement and story actions explicitly if your pause screen must freeze more than player input.

Use trigger-only activation for the first version of your replacement menus. Add hover-edge or click activation later, once the system role and navigation work.

> Show GUI draws a named screen. A system route also manages the active menu, input lock, priority and HUD restoration. Use system routes for the game menus.

Understand Buttons, action lists and scripts

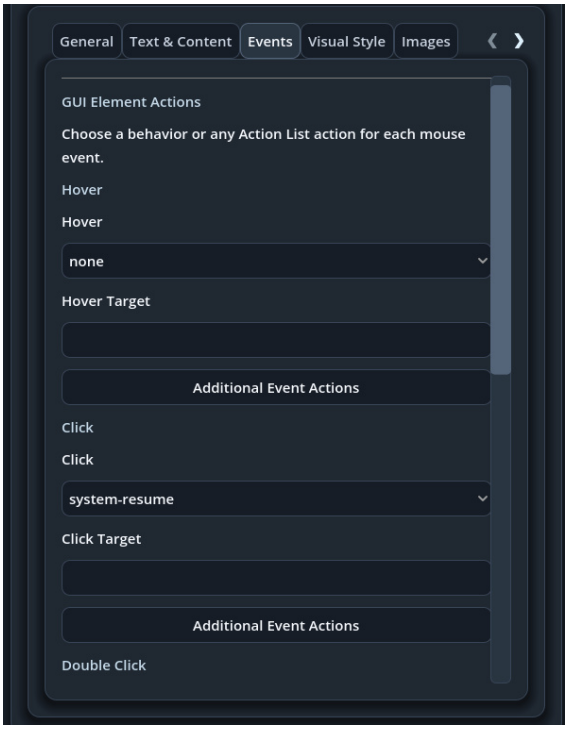

Figure 5. A Resume Button uses System Resume, which closes the system menu and restores gameplay.

Use this event choice	When
Set Verb	The Button selects Walk, Look, Use or another authored verb.
Toggle Inventory / Toggle Journal	It should open whichever personal screen belongs to the active player.
System Resume / New Game / Continue Game	It should perform that real menu operation.
An action list	Several shared actions should run in order.
Run Script / Run Visual Script	The operation needs a reusable helper, condition or custom data handling.

The shared Run Script/Run Visual Script event can run a GDScript record as well as a visual graph. Chapter 10 supplies complete helpers and shows exactly which Buttons call them.

> GUI Preview checks layout and hover/press feedback. Play checks actions, ownership, dialogue, saving and actual menu routes.

2. Replace the permanent HUD

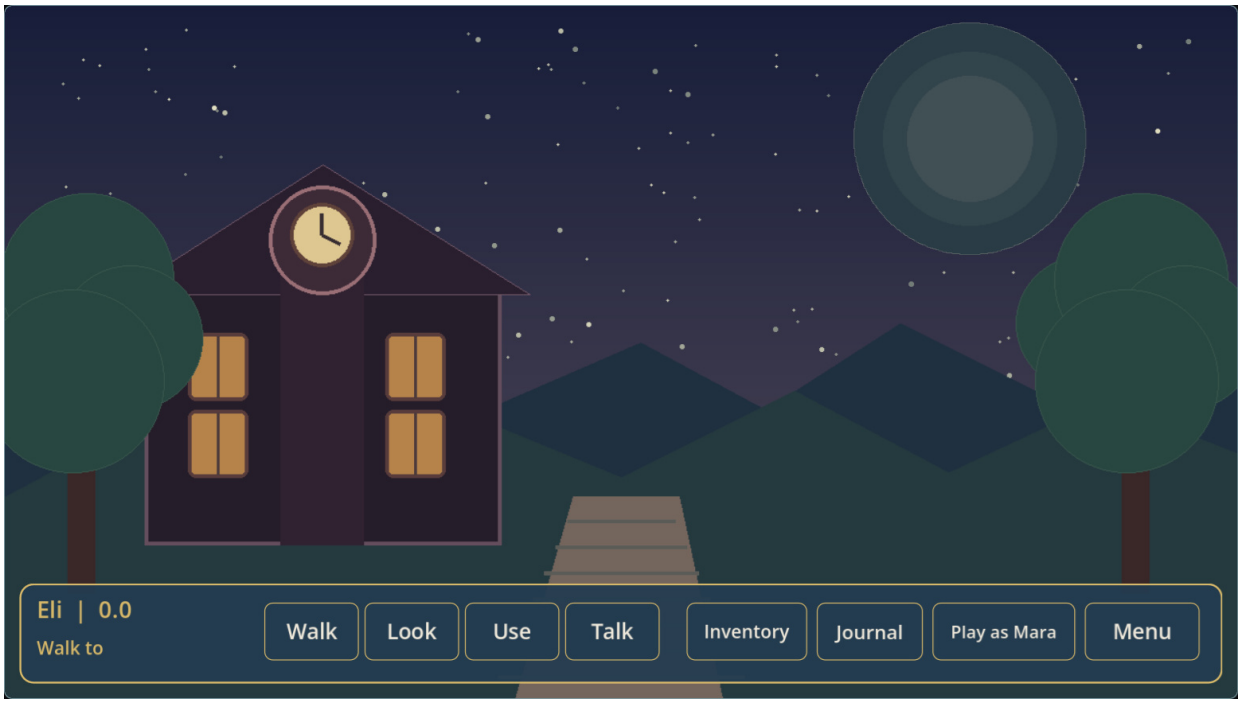

Figure 6. Eli’s HUD is a personal replacement. Its controls still call the normal gameplay systems.

Start with a small HUD that does four things well: selects actions, shows useful text, opens the player’s bag and journal, and reaches the game menu. You can change its artwork after those jobs work.

1. Create a GUI named Adventure HUD with ID hud_shared.
2. Add a background Panel near the bottom of the design canvas.
3. Place your verb Buttons, status Labels and screen openers on top.
4. Assign hud_shared to Main HUD in Global Settings and apply the setting.
5. Run Play. The default verb bar should be replaced, not left underneath the custom controls.

> A character’s Personal HUD GUI has priority over hud_shared. Clear that personal field temporarily if you are testing the project-wide HUD itself.

Lay out a practical HUD

Element	Position / size	Job
Background Panel	16, 600; 1248 × 104	One stable band for the controls.
Character / score Label	34, 610; 220 × 32	Name plus $$score in the sample.
Status Label	34, 650; 220 × 33	{verb} and {target}.
Four verb Buttons	Start 270, 620; 98 × 60 each	Walk, Look, Use and Talk; 104-pixel spacing.
Inventory / Journal	708 / 843, 620; 125 / 110 × 60	Open the active player’s assigned screen.
Character switch	963, 620; 148 × 60	Select the other playable character.
Menu	1121, 620; 119 × 60	Open the assigned Pause Menu.

These are starting measurements, not required engine values. If your project uses another resolution, preserve the spacing and hierarchy rather than pasting coordinates that no longer fit.

Set the Panel’s Z Index below the Buttons. Keep text clear of the rounded corners and test longer translated captions. A 98-pixel Button that fits “Use” may not fit every translation of “Pick up.”

> The sample background is a decorative PCA image used to make the interface easy to see. Your game can keep rendering its ordinary 2D, 2.5D or 3D room behind the same HUD.

Connect the HUD to real gameplay

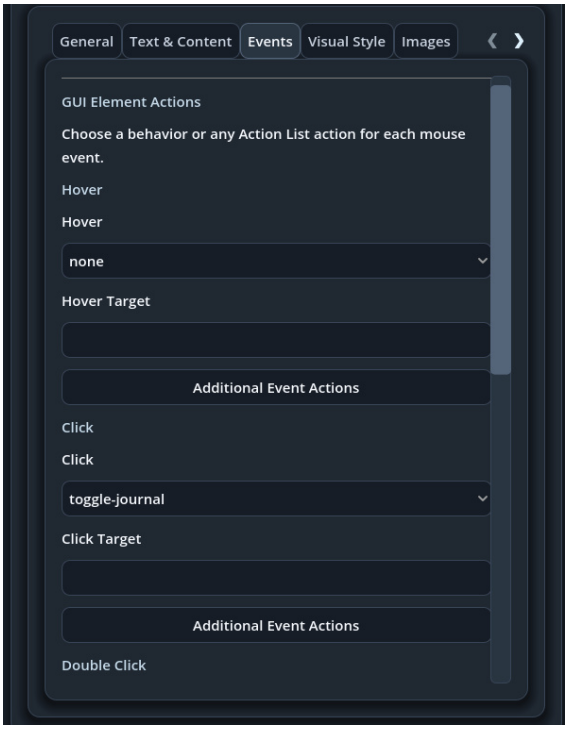

Figure 7. Toggle Journal follows the current player’s journal assignment.

1. Select the Walk Button. In Events, set its Click behavior to Set Verb and select the walk verb.
2. Repeat for Look, Use and Talk, using the matching verb IDs.
3. Set the Inventory opener to Toggle Inventory. Set the Journal opener to Toggle Journal.
4. Give the character switch Button the Switch Character behavior and choose the other character.
5. For Menu, call the open_pause helper from Chapter 10. It uses the system Pause Menu route.
6. Test a real room object. Its Look or Use result still belongs to that object’s interactions, not to the HUD Button.

A verb Button chooses the action. It does not create the action’s result on every object. Give the door an Open/Use result, the sign a Look result, and the character a Talk result as appropriate.

> Hard-coding Show GUI → tester_inventory on a shared HUD will always open Eli’s screen. Toggle Inventory is the useful choice when the active player can change.

Keep status text useful

Text	Example use
{verb}	The currently selected verb’s display name.
{target}	The current hover target’s display label.
$$score	A saved project variable named score.
$$activePlayerName	The runtime’s current player name variable.
$$ui_note	A short message stored by your own actions or scripts.

Use a Label for each job. One concise status line is easier to follow than a HUD where every Button repeats the same hover text. Leave room for an empty target when the pointer is over open ground.

If you need a complete sentence such as “Use brass key with north door,” build and test that sentence logic separately. A pair of simple wildcards is not a complete two-object grammar by itself.

1. Run Play with no item selected. Check the verb and empty target.
2. Hover a named hotspot. Check that the displayed name is the one you expect.
3. Change the verb and move the pointer away again.
4. Set ui_note from a room action and check the Label updates without rebuilding the HUD.

Use a shared HUD, personal HUDs or no HUD

Configuration	Expected result
A personal HUD is assigned	That active player uses their personal HUD instead of the project HUD.
Personal HUD is blank	The project’s Main HUD assignment is used.
Main HUD is Built-in Default, personal blank	The built-in gameplay HUD is used.
Main HUD is No HUD	The HUD is suppressed; this also overrides personal HUD presentation in this build.
A system menu opens	The gameplay HUD is hidden while that system screen is active, then restored through the system route.

Personal HUDs are exclusive. You should not have to hide hud_shared manually every time Eli takes control. The character switch updates the active HUD and hides the old personal HUD.

For an in-HUD inventory strip, place an Inventory Widget directly on the HUD. Keep the dedicated Inventory Screen assignment separate; assigning the HUD itself as that hideable screen can produce confusing open/close behavior.

> No HUD is useful for a cinematic section, but it also removes the player’s on-screen way back to inventory and menus. Keep an intentional keyboard, touch or scripted route where the game needs one.

3. Give each character their own screens

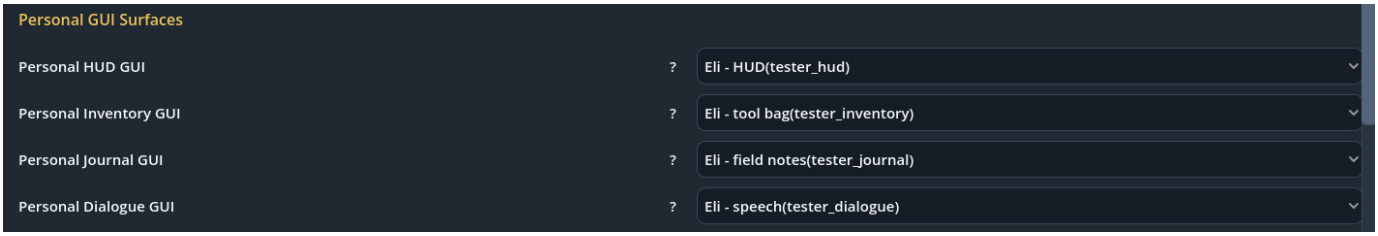

Figure 8. Personal GUI Surfaces is part of the character’s Player Settings.

The example has two playable characters. Eli uses a blue tool-bag interface; Mara uses a green satchel and notebook. Their screen IDs differ, and their inventory and note data are separate.

Character field	Eli (tester)	Mara (assistant)
Personal HUD GUI	tester_hud	assistant_hud
Personal Inventory GUI	tester_inventory	assistant_inventory
Personal Journal GUI	tester_journal	assistant_journal
Personal Dialogue GUI	tester_dialogue	assistant_dialogue

tester and assistant are the actual reusable character IDs in this example. “Eli” and “Mara” are their display names. Scripts use the IDs.

Make both characters available to play



Figure 9. Playable Character and Allow Character Switching serve different purposes.

1. Open Eli in the Character Designer and choose Player Settings.
2. Enable Playable Character. Enable Allow Character Switching.
3. Choose one default starting character, then check the Player Character selection in Project Settings.
4. Set a clear Switch Label and an order if you use next-character controls.
5. Repeat the playable/switching setup for Mara, without making both characters the default starter.
6. Place both character definitions in their intended scenes. A non-starting character left only in the database may not be discovered as a switchable runtime character.
7. Apply Changes and save before starting a fresh Play session.

A normal NPC can have portraits and dialogue. The personal HUD/inventory/journal assignments described here are chosen for the active playable character; they do not give every background NPC an independently visible HUD.

Assign the four personal screens

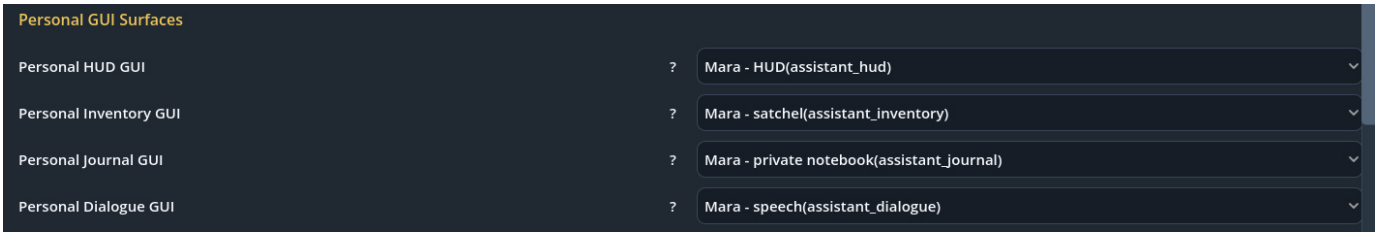

Figure 10. Mara has different GUI assignments while using the same gameplay systems.

1. Create or clone the four GUIs before assigning them to Mara.
2. In Mara’s Player Settings, select assistant_hud, assistant_inventory, assistant_journal and assistant_dialogue.
3. Check that each dropdown points at the entire GUI, not an internal Label or widget.
4. Click Apply Changes and Save Project.
5. Run Play, switch to Mara, and check her HUD appears while Eli’s disappears.
6. Use the HUD’s normal inventory and journal openers to confirm they resolve Mara’s assignments.

Leave a personal field blank when that character should use the project-wide version of that surface. You do not need to clone an unchanged menu simply to fill every dropdown.

> Personal Dialogue GUI follows the active player’s interface. If Eli speaks while Mara is being controlled, Mara’s personal dialogue presentation is still the selected shell.

Separate bags need separate inventory ownership

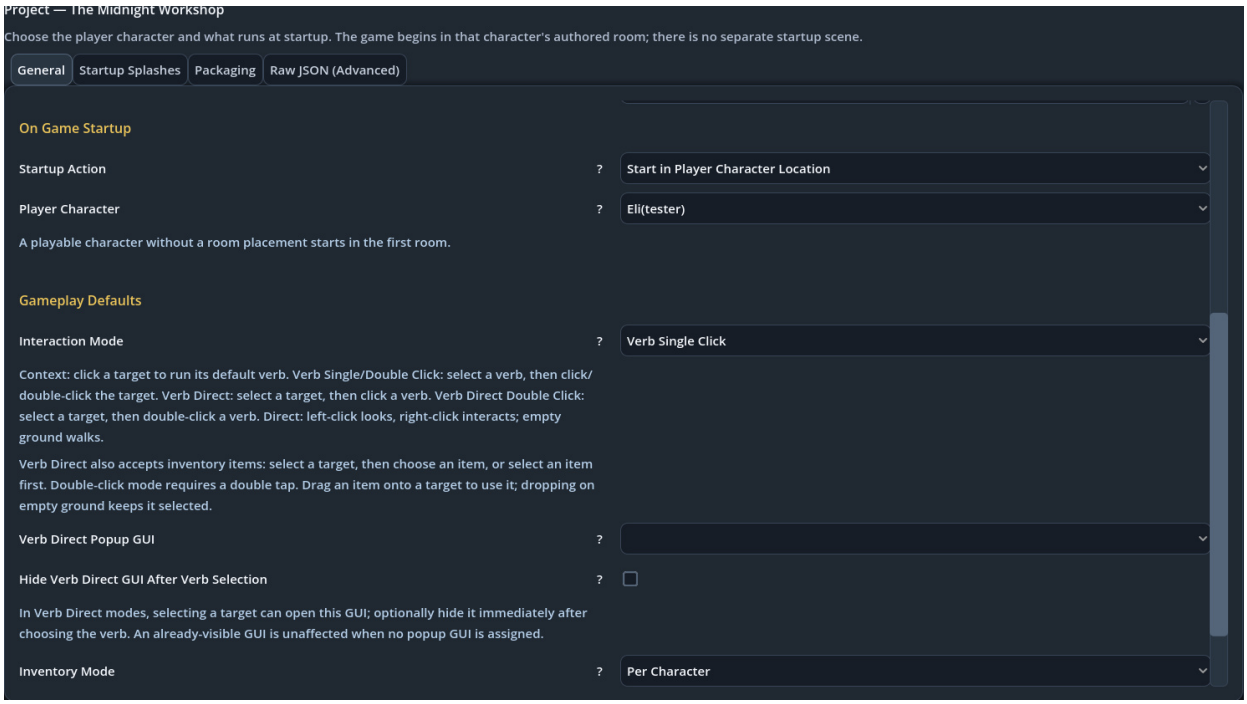

Figure 11. Inventory Mode belongs to Project Settings, rather than the GUI’s visual style.

Inventory Mode	What changes
Shared	Playable characters use one shared inventory. Separate-looking bags still show shared possessions.
Per-character	Each playable character has their own items, quantities, order and selection state.

For this tutorial choose Per-character. Put Eli’s brass_key, gear and lens in his Character Designer starting inventory. Put notebook and oil in Mara’s. Keep the project’s Initial Inventory empty for the clearest first test.

Project-level Initial Inventory is added to the starting character in per-character mode. Accidentally listing the same items in both places can make ownership tests harder to understand.

> Changing Inventory Mode changes the game’s data rules. Test it from a new game before trusting positions saved under another inventory arrangement.

Build the two inventory windows



Figure 12. Mara’s satchel shows her notebook and oil rather than Eli’s tools.

Part	Starting setup
Window / background	Full design canvas; Panel at 140, 58; 1000 × 604.
Title	A readable character-specific name at 176, 78.
Inventory Widget	180, 164; 920 × 370; four columns and two rows.
Slots	196 × 125; 8-pixel gaps; icons and names enabled.
Owner	Current Player. This makes the widget follow the actual active inventory.
Back Button	Toggle Inventory, so it closes the active inventory surface.

Clone the layout for Eli, then change the colors, title and artwork. Keep the owner rule as Current Player for both dedicated personal screens.

> Use a nonmodal bag when selecting an item must be followed by a click in the room. If you make it modal, provide an obvious close route and test item use after closing.

Current Player versus a fixed character

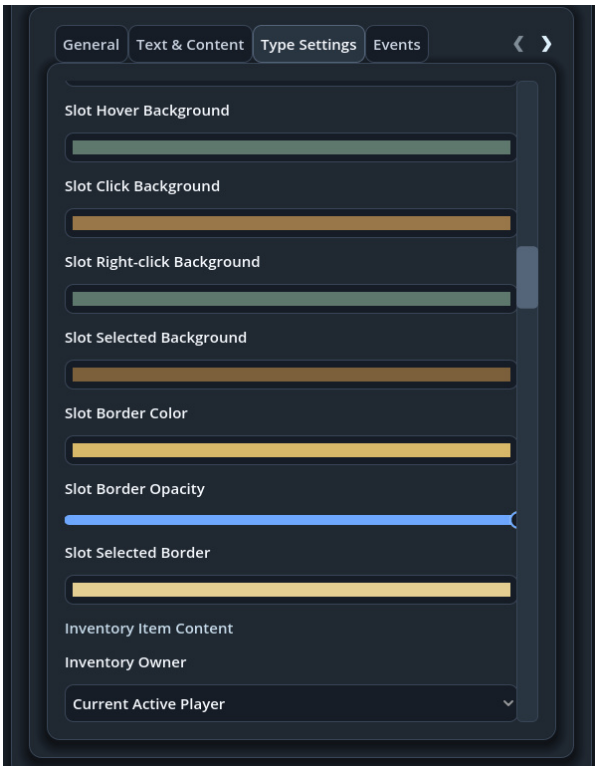

Figure 13. Inventory Owner is a widget setting. It is separate from the character’s Personal Inventory GUI.

Widget owner	Use it for
Current Player	The normal playable bag. It changes with the controlled character.
Specific Character	A deliberate view of that character’s items, such as a comparison screen.

A fixed-character widget linked to an inactive character is a read-only view in this build. It is not a way to use Mara’s items through Eli’s cursor. Give the player a proper transfer or switching action instead.

The character’s GUI assignment chooses the window. The widget’s owner chooses the inventory shown inside it. If Mara’s green satchel displays Eli’s key, check the owner field before changing the inventory data.

> Test an overflow item too. A bag that looks right with two items may clip its last row or navigation when the story gives the player ten.

Transfer items and switch safely

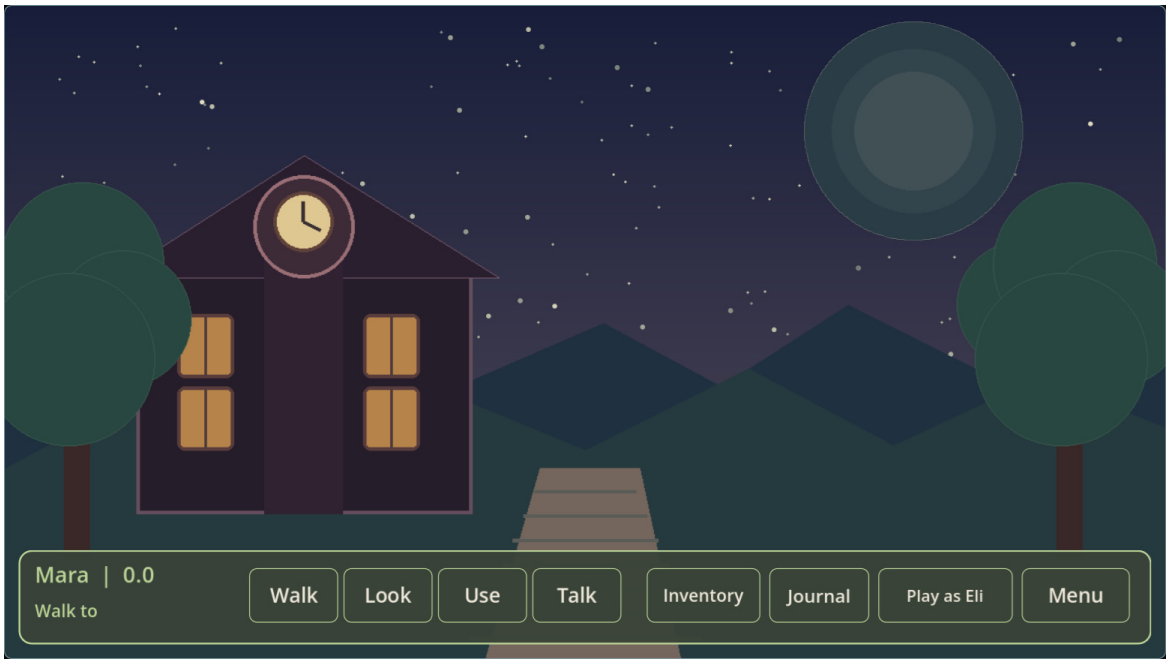

Figure 14. A character switch changes the HUD without layering both personal HUDs over the room.

1. Start as Eli. Confirm his three starting items appear.
2. Switch to Mara. Confirm she has notebook and oil.
3. Use a Transfer Inventory action when the story gives Eli’s key to Mara. Specify the source, recipient, item ID and count.
4. Check the source loses the item and the recipient gains it. Showing a new icon alone is not a transfer.
5. Close a modal menu before trying a character switch. Normal input locks can correctly prevent switching.
6. Switch back and forth. Check the displayed name, HUD, bag and notebook agree about who is active.
7. Save, change some state, then load. Check both characters’ inventories and the active character are restored.

The Switch Character GUI behavior and the shared SwitchCharacter action are sufficient for ordinary switches. Add a wrapper only when the story needs extra checks or actions.

Add a third character without rebuilding the game

Add or check	Example for a character with ID ivo
Character record	Playable and switch-enabled; placed in the appropriate scene.
Personal screens	ivo_hud, ivo_inventory, ivo_journal, ivo_dialogue.
Starting possessions	Item IDs in that character’s Starting Inventory.
Private notes variable	An array named journal_ivo.
Journal List Box	Array source journal_ivo; use the same journal_choice selection variable.
Journal reader ID	ivo_journal_body, a multiline Text Box.
Journal helper	Add ivo to the owner list in render(), so its reader becomes read-only too.
Switch controls	Add a Button or a next-character route that reaches ivo.

Keep the project-wide HUD and ordinary menus available as fallbacks. A character who needs only a different bag can share the same HUD and dialogue screen with everyone else.

Personalization is useful when it communicates character: a mechanic’s tool roll, an investigator’s casebook, a student’s scribbled notebook. Consistent action placement still helps the player move between them.

> Avoid changing the meaning of the same Button position from character to character without a clear visual cue. Different artwork need not mean relearning the controls.

4. Give journals separate looks and contents

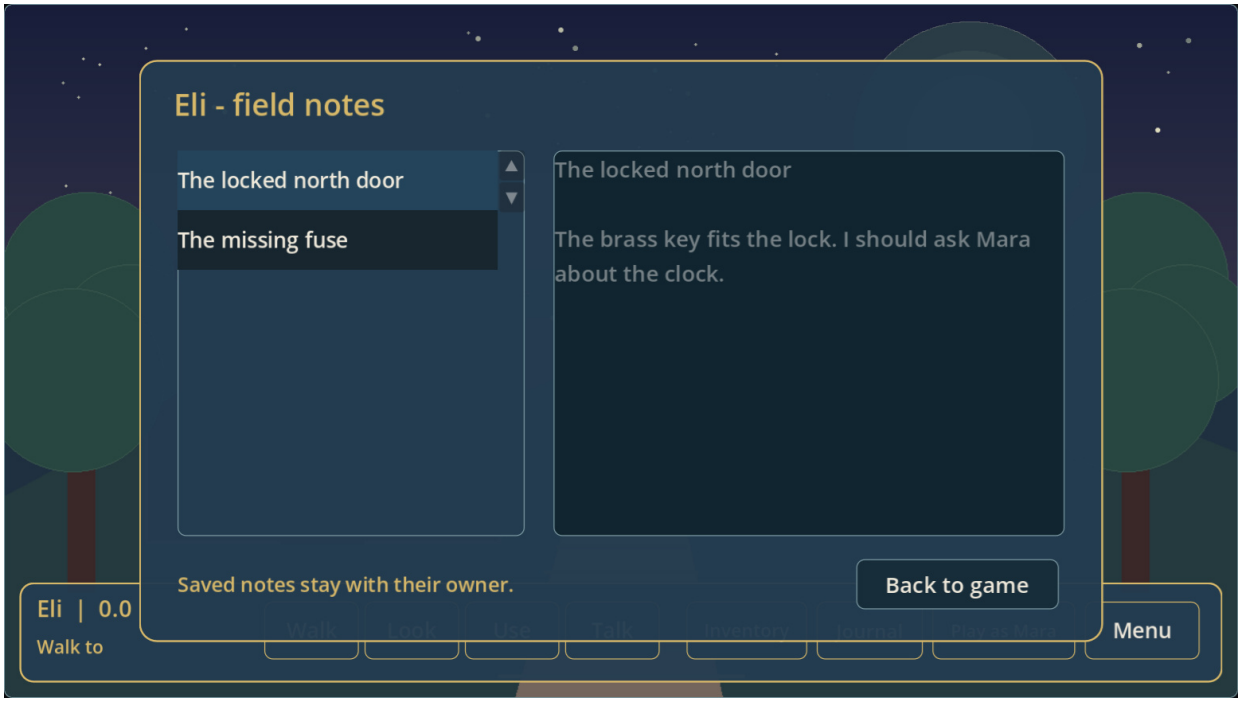

Figure 15. Eli’s field notes use his own saved array and a scrollable, read-only reader.

There are two useful kinds of journal. A quest journal reports the game’s objectives. A personal notebook records what a particular character knows. They can look similar, but they should not accidentally share the same source.

You want	Use
The same quests in a different-looking book	A personal Journal GUI containing the standard Journal widget.
Different notes for each character	A personal Journal GUI with an array-backed List Box and a reader.
A party-wide objective screen	The standard Journal widget in a shared GUI.

Assigning Personal Journal GUI does not make the standard quest manager character-specific. The private-notes example below deliberately keeps each owner’s data in a different saved variable.

Replace the shared quest-journal screen

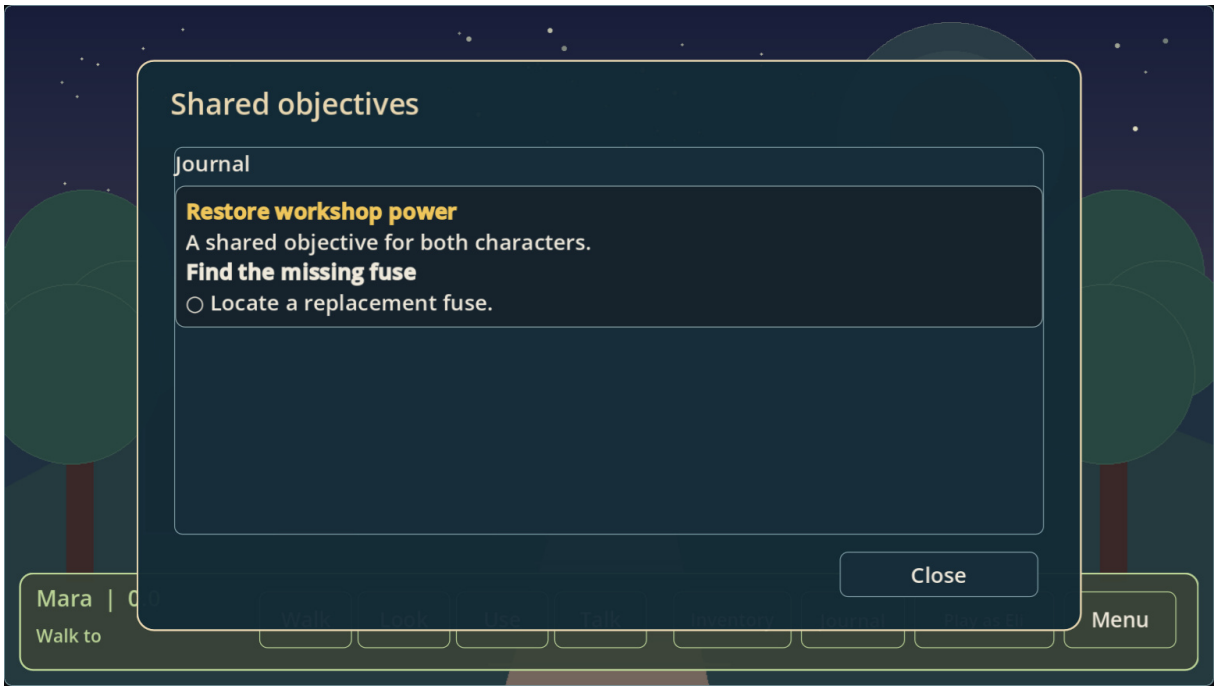

Figure 16. The standard Journal widget reads the shared quest definitions and their runtime states.

1. Create journal_shared, with a background Panel, a heading and a close Button.
2. Add a Journal element. Size it for several readable entries and scrolling.
3. Choose whether completed and failed quests should be shown.
4. Style the widget’s outer surface and its entry cards separately.
5. Set the close Button to Toggle Journal and assign journal_shared as the project’s Journal Screen.
6. Start or update a real quest, then run Play and open the journal.

For a character who needs only a different cover or palette, clone this GUI and assign that clone to Personal Journal GUI. Both books will still report the same shared quest progress.

> Do not use visual hiding of shared quest entries as your private-knowledge model. Store private facts separately, then display only the intended owner’s data.

Prepare the personal-notes variables

Variable ID	Type	Initial value / purpose
journal_tester	Array	Empty array. Eli’s entries.
journal_assistant	Array	Empty array. Mara’s entries.
journal_choice	String	Empty string. The selected entry ID in the currently open notebook.

Each entry is a dictionary with id, text and body. id is the stable key, text is the short title shown in the list, and body is the longer passage in the reader. The helper adds an entry once and updates it when the same ID is written again.

Entry field	Example
id	north_door
text	The locked north door
body	The brass key fits the lock. I should ask Mara about the clock.

An entry title can be translated or rewritten without changing its ID. Keep the ID stable when a puzzle progresses, so an update replaces the old note rather than adding a duplicate.

The example stores literal text. For a translated game, resolve translated titles and bodies before recording them, or extend the reader to localize stored text keys. Saved prose does not translate itself when the player changes language.

> The arrays are normal saved game variables. Text currently painted in a GUI control is only a view; it is not the notebook’s permanent storage.

Build the entry list

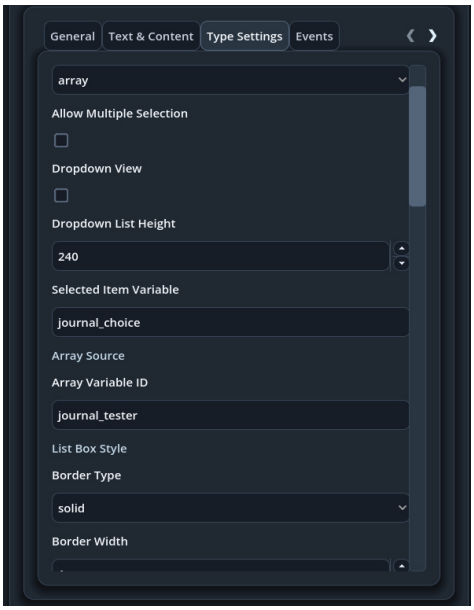

Figure 17. The personal notebook’s List Box reads a named Array Variable.

Setting	Eli’s notebook
GUI / element ID	tester_journal / tester_journal_entries
Position / size	180, 152; 360 × 400
Item Source	Array
Array Variable ID	journal_tester
Selected Item Variable	journal_choice
Multiple Selection	Off
Row height / font size	62 / 22 in the sample.
Empty text	No notes yet.

For Mara’s screen, use journal_assistant as the array source. Keep the same journal_choice selection variable: only one active character’s notebook is being operated at a time.

> A manual list typed into the designer is useful for a mock-up. It is not a replacement for a saved array that changes as the story progresses.

Make a reader that scrolls but cannot be edited

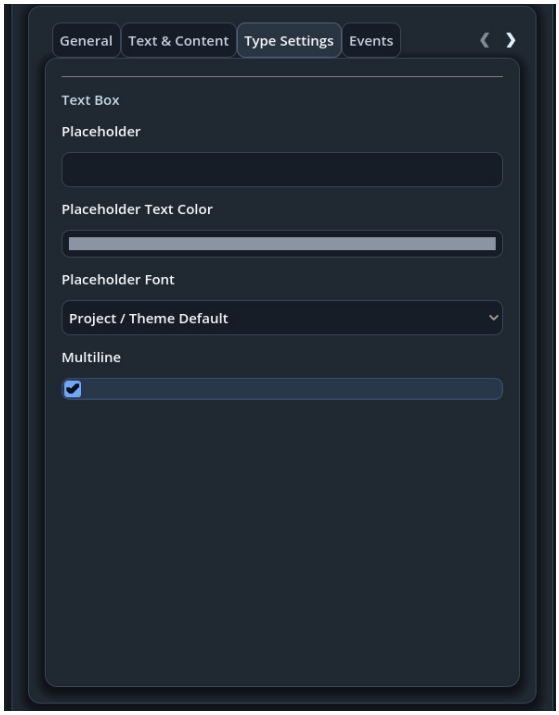

Figure 18. The reader starts as a multiline Text Box. The helper turns editing off during play.

1. Add a Text Box next to the entry list. Enable Multiline.
2. For Eli, set its element ID to tester_journal_body. For Mara, use assistant_journal_body.
3. Use position 570, 152 and size 530 × 400 as a starting point. Give the text comfortable padding and contrast.
4. Install journal_tools from Chapter 10. It sets the live TextEdit control to read-only and fills it when a title is selected.
5. Keep a Back Button set to Toggle Journal.
6. In Play, select an entry, scroll a long passage, then try typing. The player should be able to read and scroll without changing the note.

The ordinary Text Box is editable until the helper runs. Checking Multiline alone does not make a journal reader. The read-only step is part of the working example.

> The helper writes plain text. Do not expect BBCode tags such as [b] to become formatted text in this Text Box.

Record a note from a story action

Where the discovery happens	Suggested action
A door’s Look result	Run remember_door once the player notices the relevant clue.
A dialogue response	Run a wrapper that adds the newly learned information.
An action list	Use Run Script/Run Visual Script to call that wrapper at the correct step.
A visual script graph	Use Run Script or an exposed function node with the entry parameters.

1. Create the normal script record remember_door using the complete example in Chapter 10.
2. On the north door’s Look interaction, add a Run Script action selecting remember_door.
3. Run Play and look at the door as Eli.
4. Open his Journal. Select “The locked north door” and read the passage.
5. Look at the door a second time. There should still be one north_door entry, not two identical titles.
6. Change the wrapper’s body text for a later story state and call it again. The same entry ID should now show the revised note.

Do not attach a recording action to ordinary GUI hover. A player merely passing the pointer over a menu should not discover story information.

Let one character learn something privately

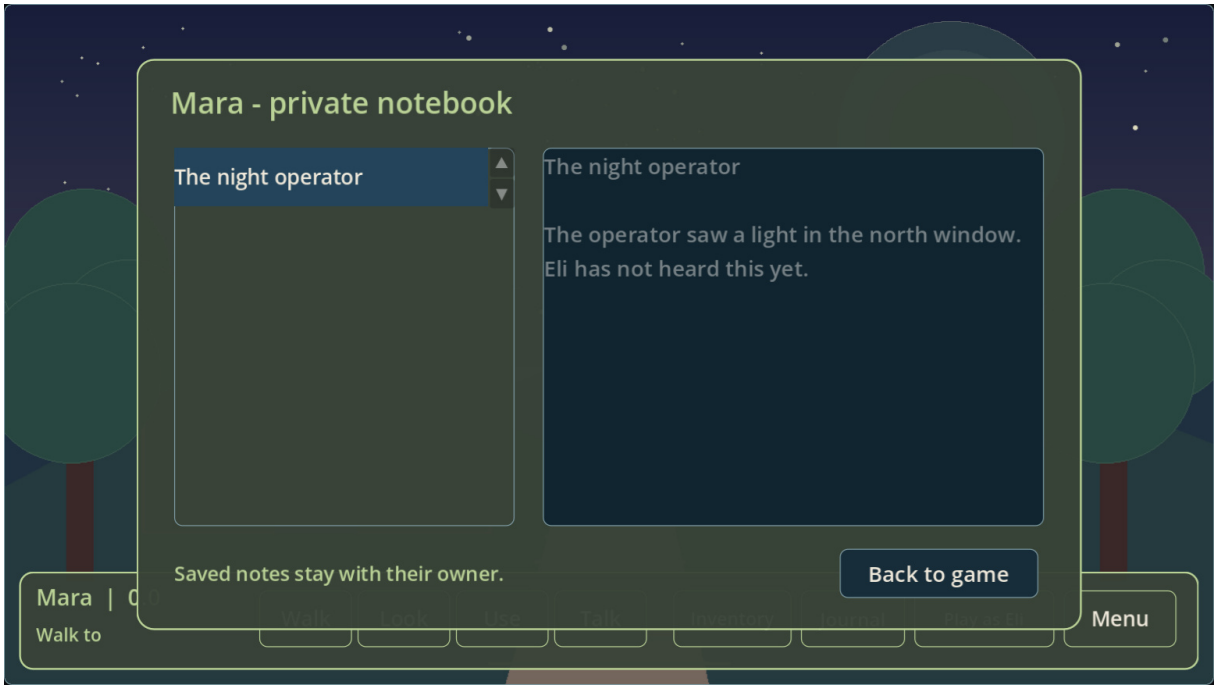

Figure 19. Mara knows about the night operator; Eli’s notebook still contains his own discoveries.

By default, add_entry writes to the active character. The optional fourth argument specifies another character ID. That is useful when a scene teaches Mara something while Eli is still the controlled character.

Call arguments	Result
entry ID, title, body	Add or update the active player’s notebook.
entry ID, title, body, "assistant"	Write directly to Mara’s notebook.
The same entry in both owners’ arrays	Both characters have deliberately learned the fact.

Information sharing is a story action. If Eli tells Mara the key fits the door, call add_entry for assistant at that point. Do not copy all of Eli’s notes simply because the player switched characters.

> For “give an item,” use Transfer Inventory. For “tell a fact,” update the recipient’s notebook or knowledge variables. These are different kinds of state.

Restore notebooks after a saved game

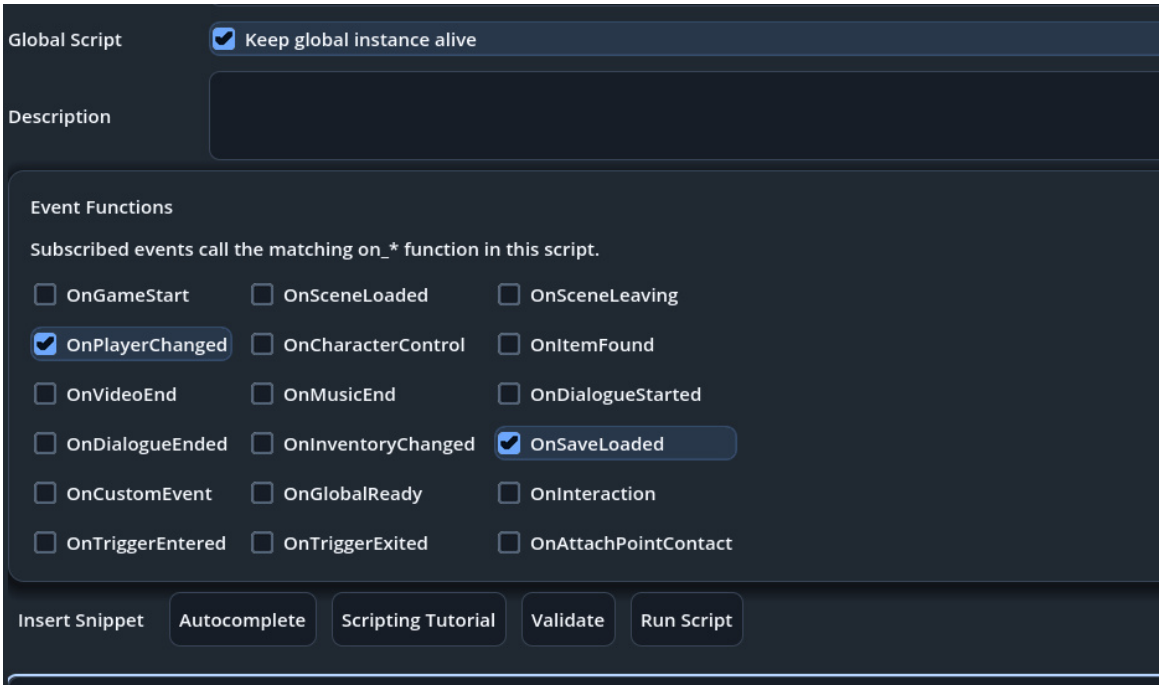

Figure 20. Enable the player-change and save-loaded event subscriptions on the journal helper.

1. Keep journal_tools as a Global Script. Its small controller observes the selected note while the game runs.
2. In its Event Functions, subscribe to OnPlayerChanged and OnSaveLoaded. Writing the functions in the source is not the same as subscribing the record to those events.
3. Give Eli two entries and Mara a different entry.
4. Save the game. Then alter the notes and switch characters.
5. Load the saved position. Check both arrays, the active HUD and the selected notebook’s reader.
6. Start a genuinely new game. Old discoveries should not remain because they were accidentally stored as authored initial values.

The controller rebuilds the reader from the saved array. It does not try to save the live Text Box as the source of truth.

> If you add another playable character, add their journal array, List Box source and reader ID, then extend the owner list in journal_tools.render().

5. Replace the main menu and pause menu

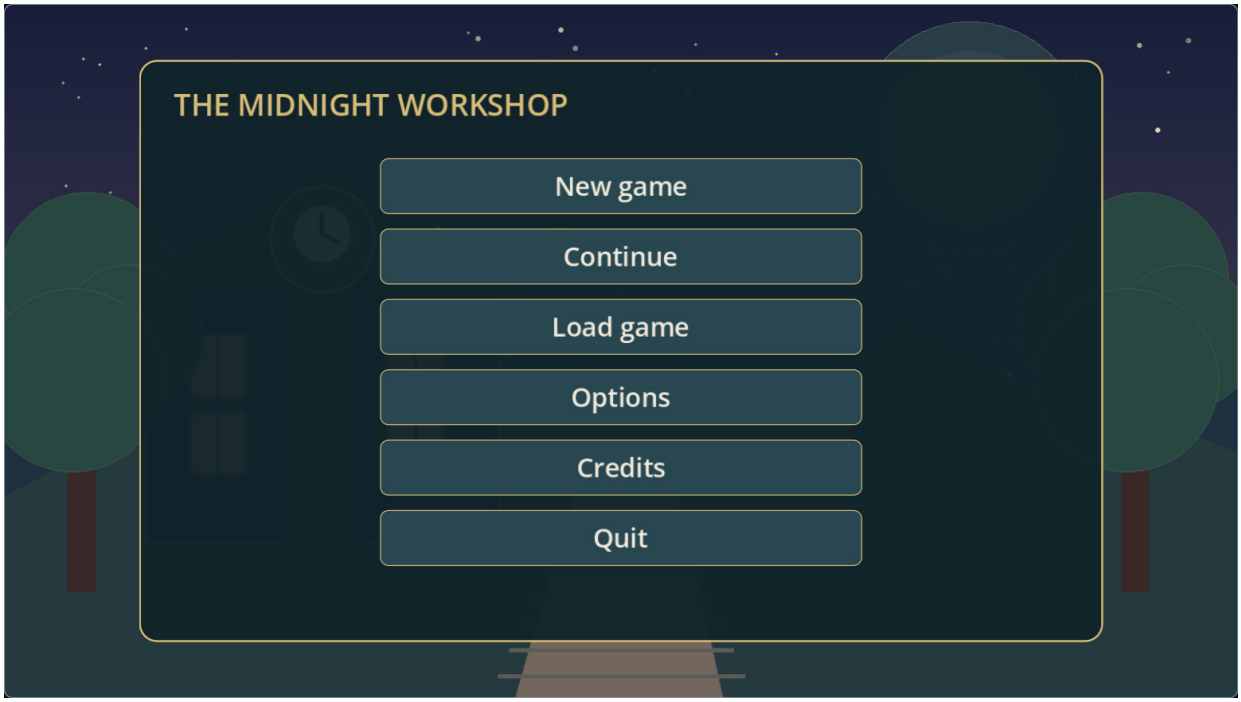

Figure 21. The title screen uses the Main Menu system role, including the real New Game and Continue operations.

A good menu is a set of routes, not just a collection of pictures. First make each route work with plain Buttons. Then replace the fonts, backgrounds, highlights and transitions.

1. Create main_custom with a background, title and six Buttons.
2. Set it to start hidden, use trigger-only activation, and make it modal.
3. Assign it as Main Menu in Global Settings.
4. Enable Show Main Menu on Start in Global Settings when the title screen should precede gameplay.
5. Use a full Play / Restart launch to check startup. Test Scene intentionally skips the ordinary startup presentation.

> Showing a GUI with a “New Game” caption does not reset a game. Its Click behavior must call the real new-game operation.

Wire the main-menu Buttons

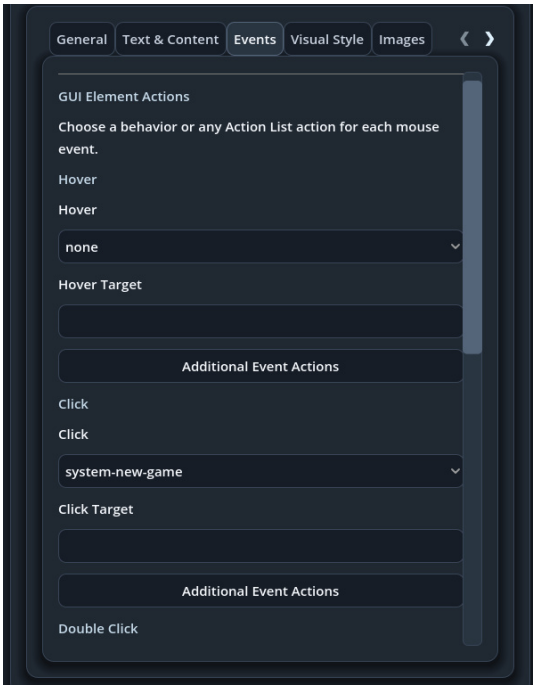

Figure 22. System New Game starts the configured project startup sequence rather than merely hiding the title screen.

Button	Click behavior / helper
New game	System New Game
Continue	System Continue Game; disabled if no save exists.
Load game	Run open_main_load, which remembers that Back should return to Main Menu.
Options	Run open_main_options.
Credits	Show GUI → credits_custom; its Back Button uses System Main Menu.
Quit	System Quit; test it in the standalone game.

The sample Continue Button has ID main_custom_button_1. menu_tools updates its enabled state when the main menu opens. Change that ID in the helper if you rename the Button.

> Continue uses the newest available saved position, which can include an autosave. Load lets the player choose a specific position.

Build the pause screen



Figure 23. The assigned Pause Menu replaces the built-in Game Menu.

Button	Route
Resume	System Resume
Save game	Run open_pause_save.
Load game	Run open_pause_load.
Options	Run open_pause_options.
Main menu	System Main Menu
Quit	System Quit

Keep the HUD’s Menu Button and the player’s pause key on the pause-menu route. Opening pause_custom directly with Show GUI does not provide the same menu lifecycle as show_pause_menu().

> Resume should close system interfaces and restore the previous HUD state. A plain Hide GUI Button is the wrong shortcut when it leaves the runtime thinking a system menu is still active.

Give every submenu an honest Back route

Opened from	Destination	Back should return to
Main Menu	Load / Options	Main Menu
Pause Menu	Save / Load / Options	Pause Menu
Credits opened from Main Menu	Credits	Main Menu
Gameplay	Inventory / Journal	Gameplay, through that surface’s close/toggle command.

The custom menu system is not a web browser with an automatic history stack. Chapter 10’s menu_tools remembers the intended parent for Save and Load. options_tools does the same for Options.

1. Call open_main_load from the title menu, not the pause-load wrapper.
2. Use menu_back on the Save/Load Back Buttons.
3. Use options_back on the Options Back Button.
4. Open each submenu from both entry points and test Back before adding decorative animation.
5. Test the pause key too. Decide whether that shortcut closes the current system screen directly or should follow your explicit Back route.

If a shortcut bypasses the wrapper, the helper cannot infer an arbitrary navigation history. Route important entry Buttons through the supplied wrappers and test other entry routes deliberately.

Create the replacement Save and Load screens

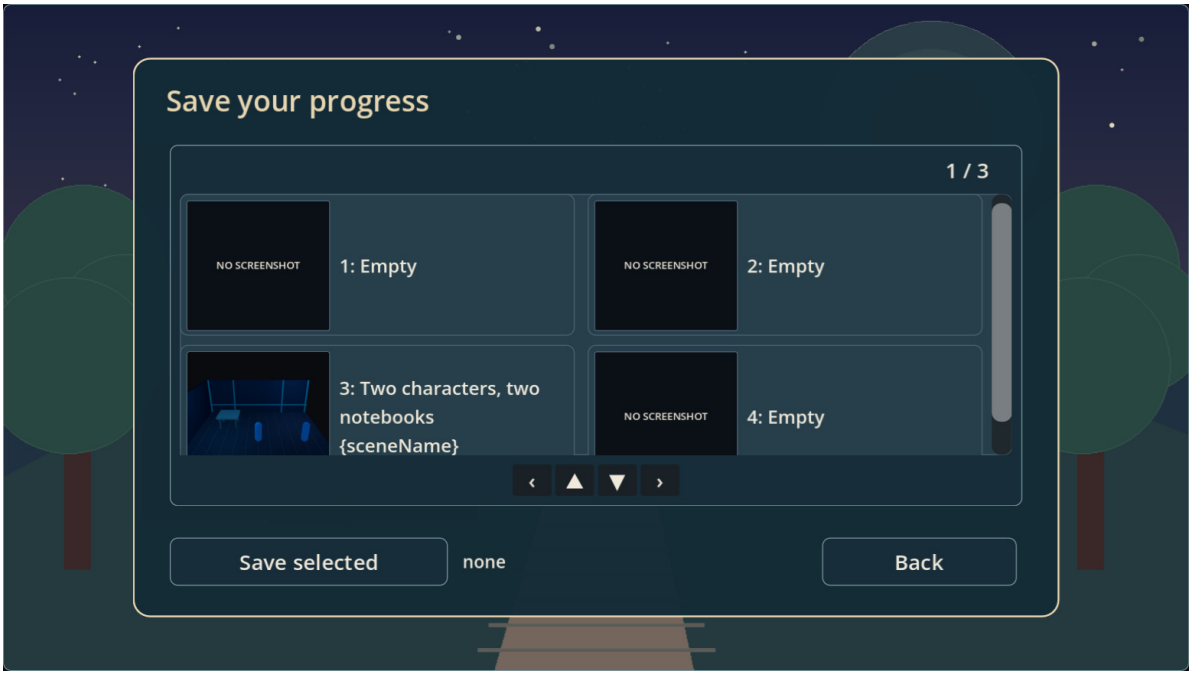

Figure 24. The custom Save Screen contains a real Save Game widget, plus ordinary command Buttons.

1. Create save_custom and load_custom. Give each a clear heading and a modal background shell.
2. Add a Save Game element to each. Use Save mode on save_custom and Load mode on load_custom.
3. Choose a slot range that fits the project’s configured number of Save Slots.
4. For this layout, use two columns, two rows and four positions per page.
5. Show screenshots, position names and the scene name. Leave enough height for the navigation controls.
6. Assign the two GUIs to Save Screen and Load Screen in Global Settings.

> The screenshot and slot data come from the save system. A background Image that resembles a save card is only decoration until a widget or script connects it to an actual saved position.

Keep saves deliberate and loads predictable

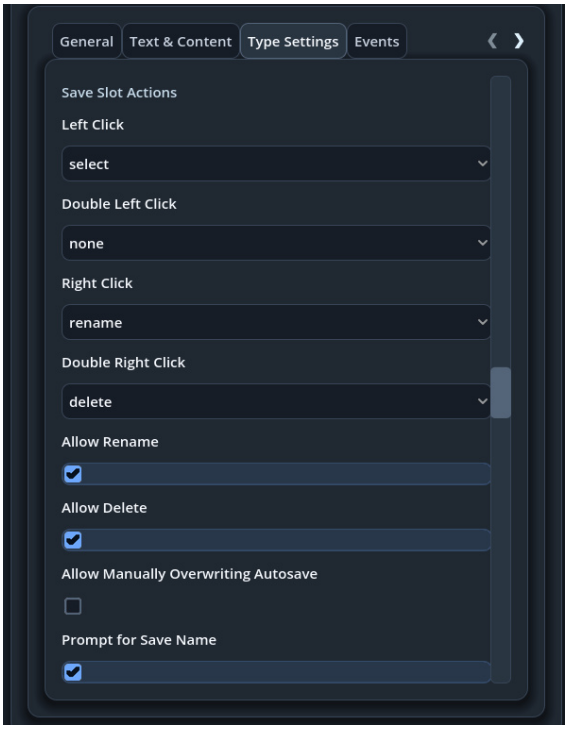

Figure 25. The widget can prompt for a name and confirm overwriting an existing position.

Setting or command	Recommended first setup
Click on a slot	Select; no automatic save or load.
Double-click	None until you intentionally choose a shortcut.
Save Button	Run save_selected, using the selection guard in menu_tools.
Load Button	Run load_selected.
Prompt for Name	On for the Save widget.
Confirm Overwrite	On.
Back	Run menu_back.

The ordinary selected-slot command can use a fallback slot when the player has not selected one. The supplied guard requires an explicit selection and reports “Choose a slot first.”

> Keep confirmation prompts sequential. Do not open another modal menu while a save-name or overwrite dialog is still waiting for an answer.

Add your own credits screen

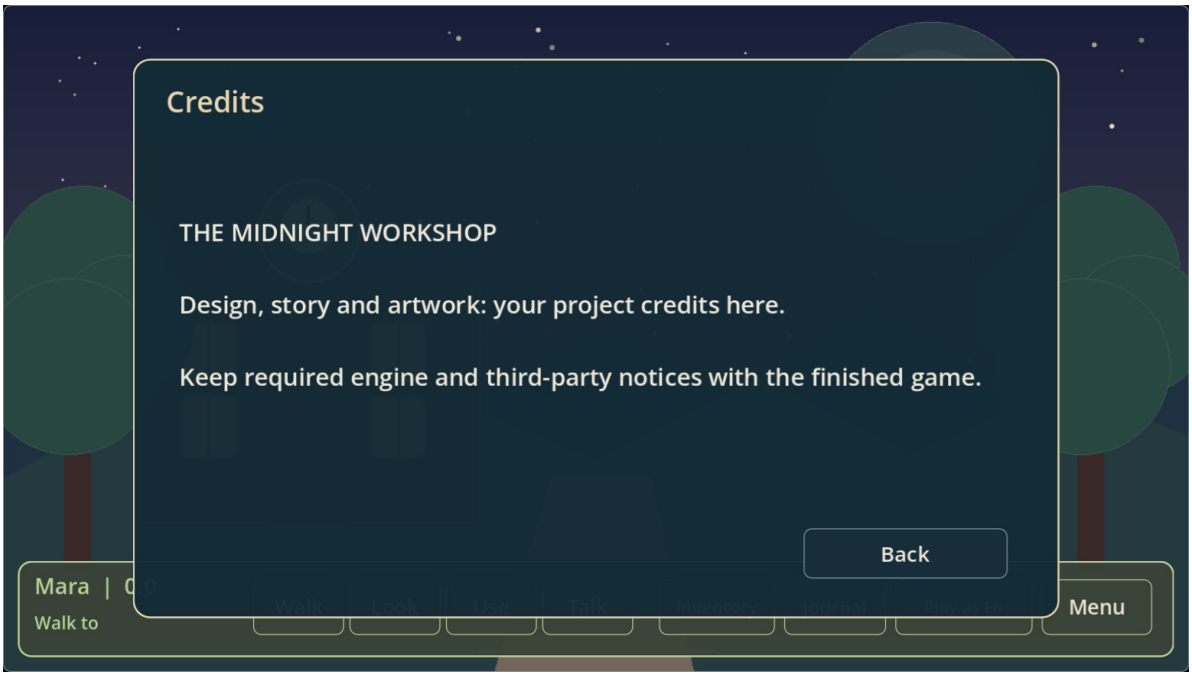

Figure 26. Credits can be another authored GUI with a clear route back to the title menu.

1. Create credits_custom with a background, a heading and readable credit text.
2. Use your own project credits, asset acknowledgements and required notices.
3. Make it modal if it takes the player out of ordinary gameplay.
4. Open it from the main menu with Show GUI.
5. Set its Back Button to System Main Menu. This re-enters the correct system screen after the modal handoff.
6. For long credits, use a scrollable reader rather than shrinking a page of names into tiny type.

There is no separate Credits assignment in the thirteen-role list. This extra GUI works through its opener and Back behavior. The built-in menu has its own credits view; replacing Main Menu gives you control over which route your players see.

> Changing the appearance of the credits screen is separate from the notices that need to accompany the distributed game.

Test the whole menu loop

1. Start with no saved games in a disposable test profile. Continue should be disabled.
2. Choose New Game. Check the configured startup action and correct starting character.
3. Open Pause, then Options, then Back. You should be at Pause.
4. Open Save. Press Save without selecting a slot; the guard should ask for a selection.
5. Select a slot, name it and confirm. Return to gameplay.
6. Open Main Menu. Continue should now be available.
7. Open Load and return with Back. You should still be at Main Menu.
8. Load a saved game and check the correct character, scene, inventory and private notes.
9. Repeat the key routes with a controller or touch input if the game supports them.

Quit and fullscreen need a standalone test. The embedded editor preview intentionally protects the editor window, so it is not a reliable place to judge those two operations.

> Keep one known-good menu copy until the replacement passes this loop. You can redesign a Button’s artwork without rebuilding the navigation each time.

6. Replace the game-options menu



Figure 27. The replacement Options Screen has real audio, text, language and accessibility controls.

An options menu has two jobs. It must show the player’s current settings when it opens, and it must apply the chosen values to the real runtime options. A Slider linked only to an ordinary variable does neither automatically.

1. Create a GUI with ID options_custom. Start hidden, use trigger-only activation, and make it modal.
2. Assign it as Options Screen in Global Settings.
3. Build the Sliders, Check Boxes, language List Boxes and three command Buttons described in this chapter.
4. Install options_tools as a Global Script, plus the small normal-script wrappers in Chapter 10.
5. Open it through open_main_options or open_pause_options, according to the Button’s location.

> The controller is part of this tutorial, not an engine modification. It reads the actual runtime options and the actual live GUI controls.

Set project defaults before designing overrides

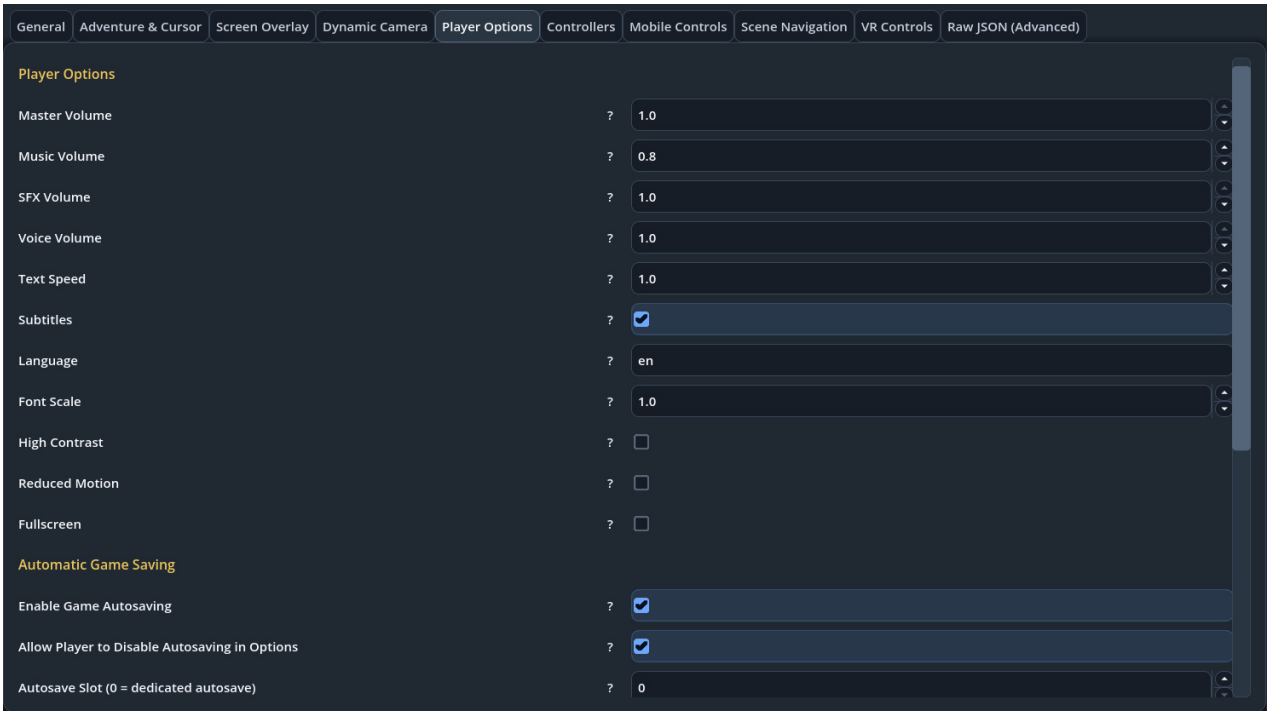

Figure 28. Player Options in Global Settings supplies the project defaults.

Defaults describe the experience you want for a first-time player. The player can then save preferences such as quieter music or subtitles. The custom menu should load their current preferences, rather than resetting every Slider to its designer preview value each time.

Kind of state	Where it belongs
Project defaults	Global Settings → Player Options.
Current player preferences	The runtime options dictionary.
Unapplied edits in this menu	The live Sliders and Check Boxes.
Quest progress and private notes	Saved game variables and other saved game state.

Editor Play uses project defaults when it starts. A standalone game can reload saved player preferences. This deliberate difference prevents an old test preference from hiding a change you made in the inspector.

Build six correctly ranged Sliders

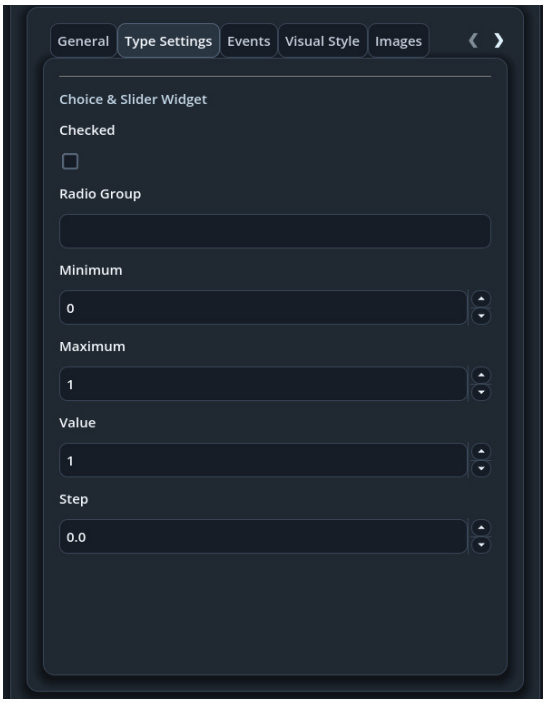

Figure 29. A volume Slider uses 0 to 1 with a small step, not a generic 0-to-100 range.

Element ID	Option key	Minimum / maximum / step
opt_master	masterVolume	0 / 1 / 0.01
opt_music	musicVolume	0 / 1 / 0.01
opt_sfx	sfxVolume	0 / 1 / 0.01
opt_voice	voiceVolume	0 / 1 / 0.01
opt_text	textSpeed	0.1 / 5 / 0.05
opt_font	fontScale	0.5 / 2.5 / 0.05

Place the labels at x = 180 and the Sliders at x = 400. In the sample, the six rows start at y = 160 and are spaced 58 pixels apart. Each Slider is 300 × 34. In Visual Style, start with Border Width 3 and a visible border color; a zero-thickness flat track can leave only the handle visible.

> Leave the Slider’s ordinary Variable field blank for this staged example. options_tools reads the control’s value on Apply. A variable named musicVolume is not the same thing as applying the musicVolume runtime option.

Add the five Check Boxes

Element ID	Option key	Meaning
opt_subtitles	subtitles	The game’s subtitle preference.
opt_contrast	highContrast	The runtime’s contrast preference.
opt_motion	reducedMotion	Reduced-motion preference.
opt_fullscreen	fullscreen	Windowed or fullscreen in a standalone game.
opt_autosave	gameAutosave	Player-controlled automatic saving, when the project allows it.

Use clear positive captions: “Subtitles” is easier to understand than a checked box labelled “Don’t hide subtitles.” Put the Check Boxes in one column, starting at 755, 154 with 49-pixel row spacing.

The helper disables Automatic Saving when the project’s save configuration does not allow the player to change it. It does not use the checkbox to override a project that has disabled autosaving altogether.

1. Give each checkbox the exact element ID in the table.
2. Use ordinary checked/unchecked feedback and readable captions.
3. Keep their Variable fields blank in this staged menu.
4. Test both an Apply and a Back / discard operation.

> A saved preference is not proof that every custom element responds to it. The accessibility section later in this chapter explains the limits of fixed-size authored GUI styling.

Make language dropdowns from List Boxes

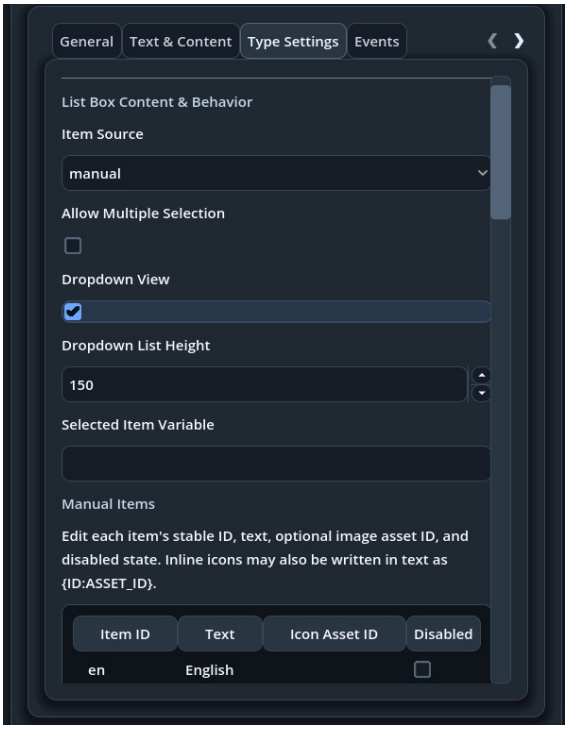

Figure 30. Dropdown View gives a List Box a compact selection control.

List Box ID	Item IDs and captions
opt_language	en → English; it → Italiano
opt_voice_language	auto → Follow game language; en → English; it → Italiano

1. Set Item Source to Manual and turn on Dropdown View. Allow only one selection.
2. Enter the IDs and captions above, or replace them with your game’s supported language codes.
3. Keep the game-language list and voice-language list separate.
4. The helper converts the voice list’s auto item to the runtime’s empty voice-language override.
5. Import actual game translations and translated voice assets through your normal translation workflow before advertising a language as supported.

> Adding “Italiano” to a dropdown does not translate the story or record Italian voice acting. The code is a selection; the translation and voice content still have to exist.

Load the current settings when Options opens

Step	What options_tools does
Open	Uses the optionsMenu system route and remembers the parent menu.
Read	Gets a copy of the current runtime options.
Fill	Sets each live Slider, Check Box and language selection.
Edit	Leaves changes in the controls until Apply is chosen.
Reopen	Reads the actual options again, discarding an older uncommitted edit.

This is why the sample has an opening wrapper for both Main Menu and Pause Menu. The helper’s default Back destination is Pause Menu; an explicit main-menu opener supplies Main Menu instead.

The global controller also fills the controls when it notices Options becoming visible through another route. That protects against stale Slider positions. It cannot infer an arbitrary Back history for a shortcut that bypassed your navigation wrapper.

1. Set music to 0.35 and Apply.
2. Leave Options and reopen it. Music should still read 0.35.
3. Change it to 0.10 and press Back without applying.
4. Reopen Options. It should read 0.35 again.

Make Apply change and save the real options

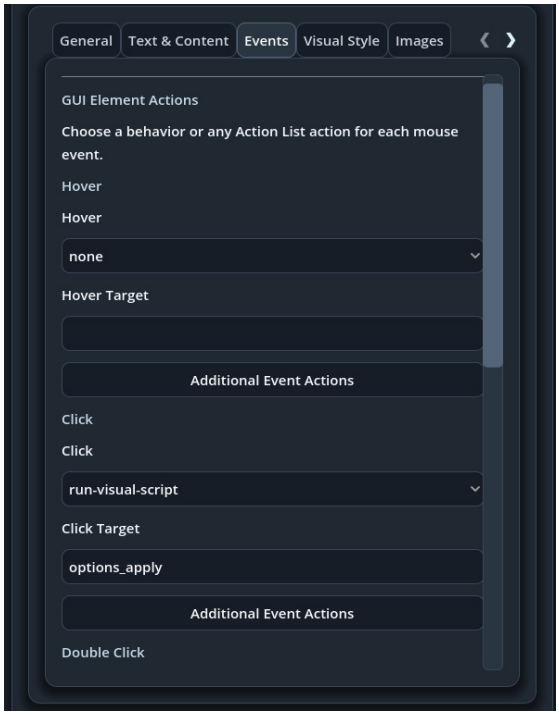

Figure 31. Apply runs the options_apply wrapper, which calls the controller's apply function.

Apply starts from the current runtime options, then replaces the keys covered by your custom controls. That preserves unrelated preferences rather than discarding them when you redesign the menu.

1. Set Apply's Click behavior to Run Script/Run Visual Script and select options_apply.
2. On Apply, the helper reads and clamps the Sliders, reads enabled Check Boxes, and reads the language selections.
3. It calls apply_runtime_options with persistence enabled.
4. The runtime applies audio buses, display preference and its supported text/UI preferences, then saves the options file.
5. The helper reloads the controls from the applied values and reports "Options saved."

Change Game Setting is useful for a one-off action or visual node. This menu uses the runtime-options apply method because it also needs a complete, explicit persistence step.

Keep Defaults and Back predictable

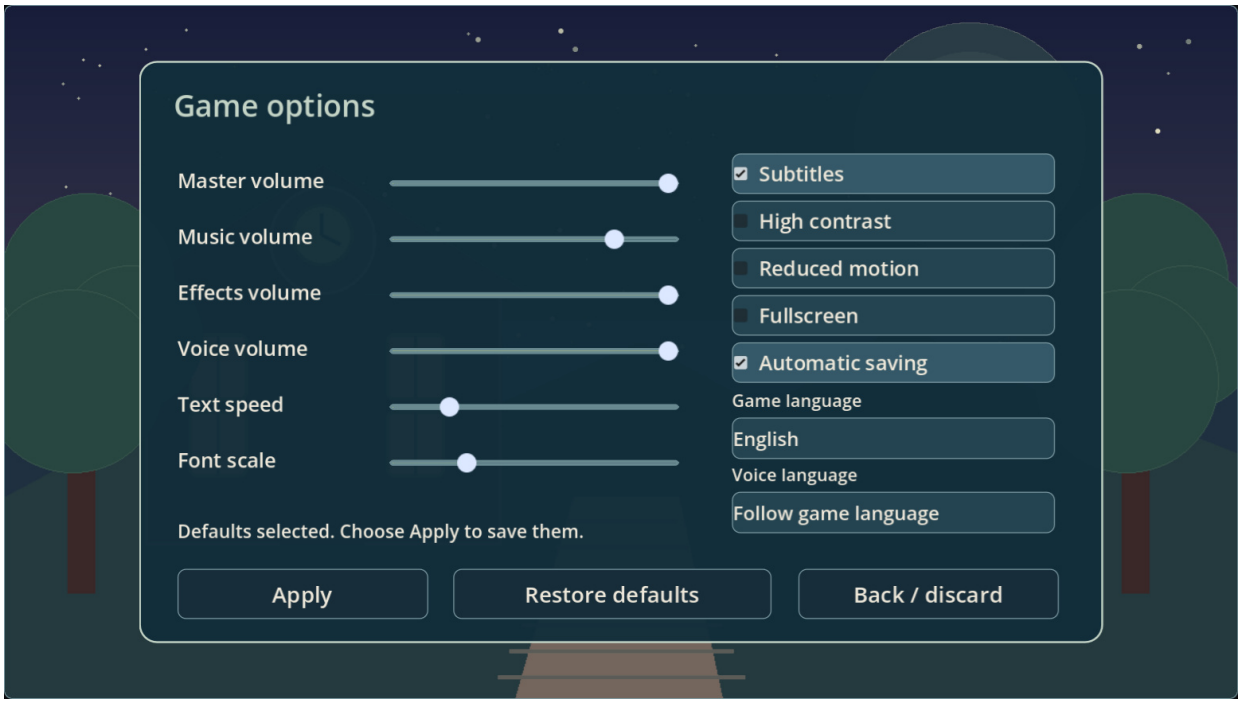

Figure 32. Restore Defaults fills the controls but waits for Apply before saving.

Button	This tutorial’s behavior
Apply	Apply the visible values and save them. Stay on Options.
Restore defaults	Fill the project defaults into the controls. Wait for Apply.
Back / discard	Return to the remembered parent menu. Discard unapplied control changes.

The built-in restore-default-options method applies and persists immediately. The tutorial’s defaults() function deliberately fills the controls instead, so Back can still discard the staged reset.

State the chosen behavior clearly in your own captions and messages. A player should not have to guess whether restoring defaults has already changed their audio setup.

> Defaults here resets preferences. It does not clear inventory, journal entries, save slots or story progress.

Test audio, text and language separately

Control	A useful real test
Master / Music / Effects / Voice	Play one source on each intended audio bus. Test quieter values and zero.
Text speed	Use an actual spoken/typewriter line, then compare the reveal timing.
Subtitles	Test spoken lines with and without recorded voice, including accessibility cases.
Game language	Open a screen containing text with a real translated entry.
Voice language	Play a line that has the selected language’s voice asset.
Automatic saving	Trigger an event that normally produces an autosave, then check the save system.

Do not judge the Effects slider by music that was accidentally placed on the wrong bus. A preference can be applied correctly while an incorrectly routed asset seems to ignore it.

The player’s saved preferences and a saved story position serve different purposes. Test quitting and relaunching the exported game as well as saving and loading the story.

> Only include options your game actually honors. A checkbox that changes a stored value but never changes the experience is unfinished UI.

Plan accessibility for your own artwork

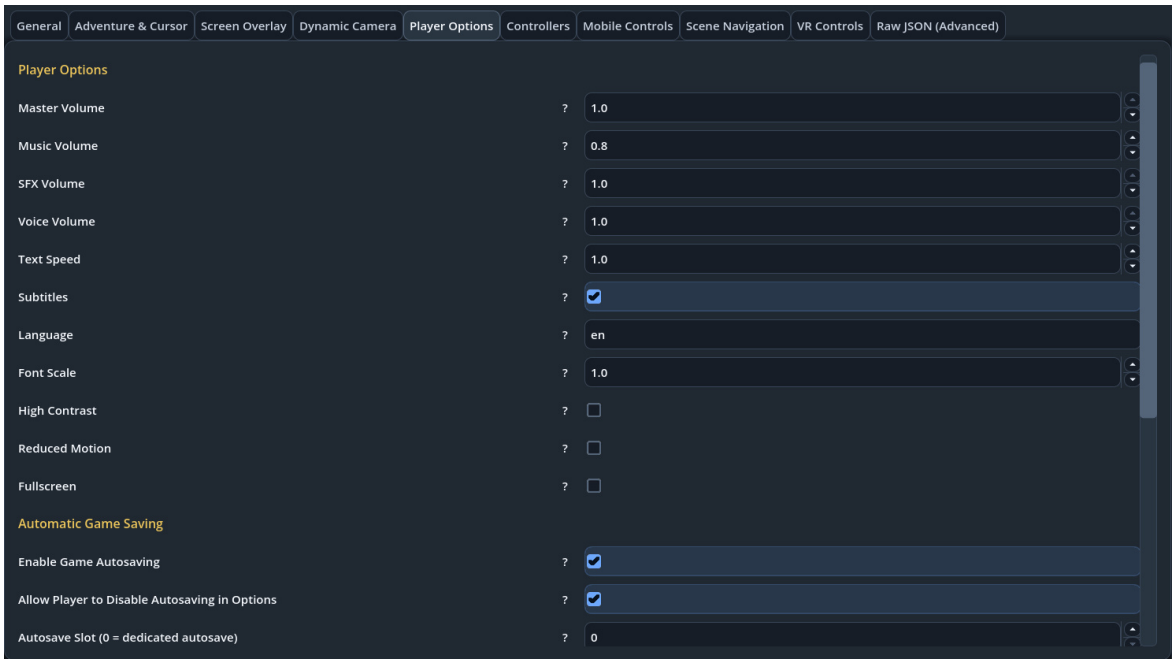

Figure 33. Project defaults include text and accessibility preferences; custom authored layouts still need their own checks.

The runtime’s font and contrast preferences do not guarantee that every authored GUI reflows. Your Labels and Buttons may have explicit font sizes, fixed dimensions, colors and shaders. Those choices can override the effect a player expects from a generic preference.

1. Give your normal GUI enough room for readable text before adding a font-size option.
2. Test whether each custom element actually changes when the runtime option changes.
3. For a guaranteed large-text interface, author a roomier alternate layout or add a tested style controller that resizes both text and containers.
4. Provide deliberate high-contrast artwork and borders for your own screens.
5. Use the reducedMotion preference to choose a short fade or cut in custom animation scripts. Do not assume it automatically cancels every custom Tween.
6. Check touch targets, keyboard focus, selected states and long translated captions in the exported game.

> Fullscreen and saved preferences require a standalone test. Embedded Play deliberately leaves the editor’s window mode alone and starts from project defaults.

7. Replace speech, dialogue and message screens

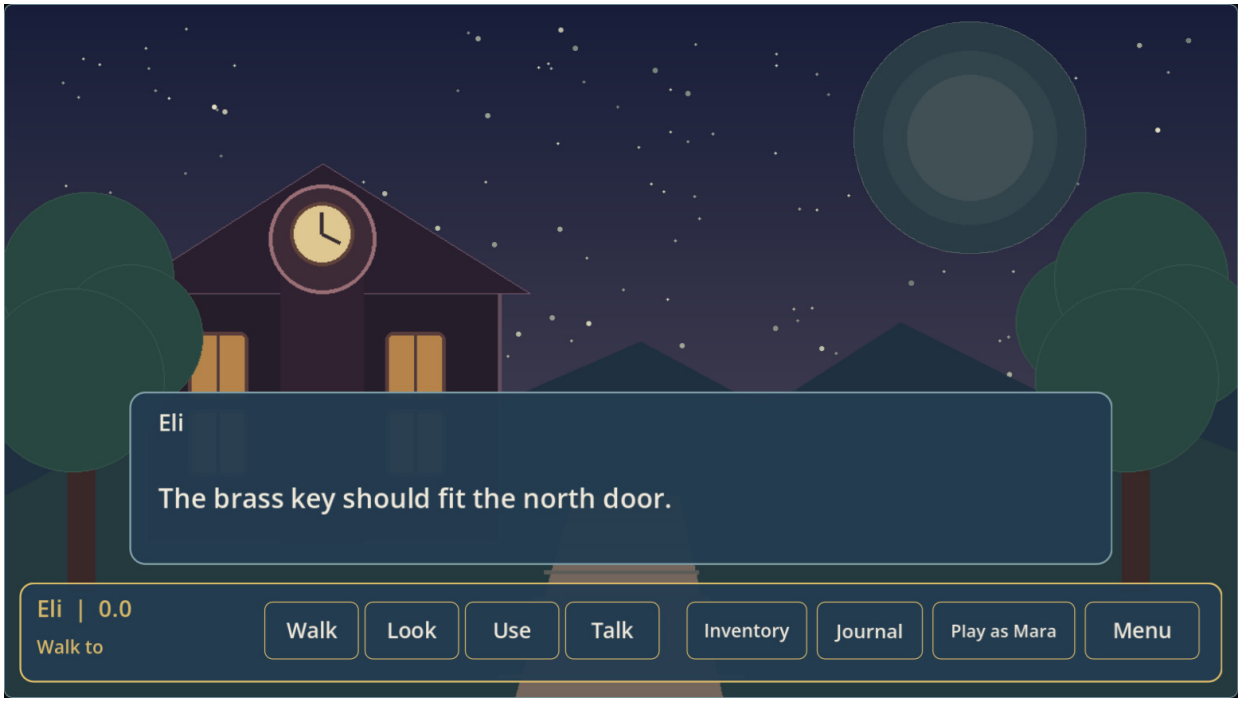

Figure 34. A real Character Say line is displayed in Eli’s personal speech GUI.

These screens receive live text from the dialogue system. Their special Group IDs tell the runtime which element should show the line or portrait. An ordinary decorative Label with the right-looking caption will not receive the next spoken line automatically.

Global role	Typical use
Dialogue	A line in a conversation graph.
Speech Bubble / Character Say	A character observation or other say action.
General Narrator Speech	Show Message and other general narration.
Dialogue Choices	The player’s available responses.

A personal Dialogue GUI overrides the active player’s Dialogue/Speech shell. The global narrator and choice roles remain separate.

Bind a spoken line with its Group ID

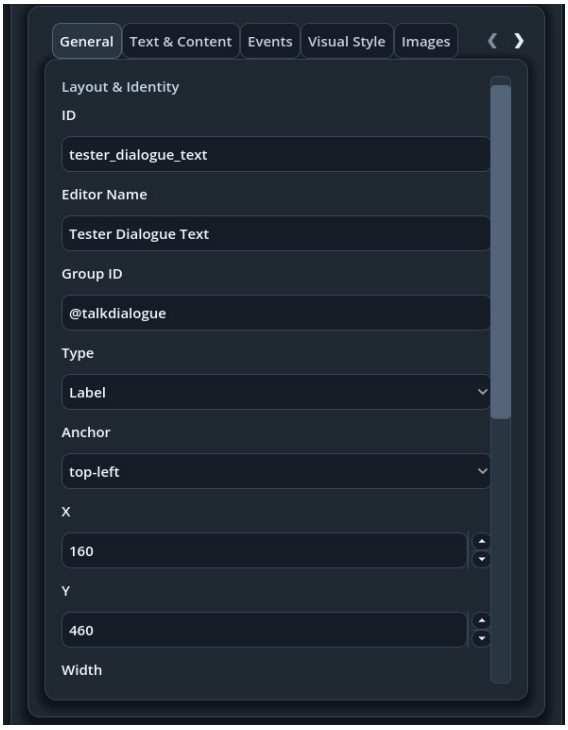

Figure 35. Set the text element’s Group ID to @talkdialogue.

1. Create dialogue_shared with a background Panel and a readable Label or Button for the spoken text.
2. In the text element’s General properties, set Group ID to @talkdialogue.
3. Do not put the Group ID in the element’s ordinary ID or Name field.
4. Assign the GUI to Dialogue, or clone and assign it to Speech Bubble / Character Say.
5. Keep the GUI hidden initially. The speech system shows and hides it.
6. Run a real Say or dialogue action and confirm the text changes.

Clicking @talkdialogue can continue a spoken conversation line when that line is waiting for input. A timed Character Say action still follows its own timing. General timed messages also use their message duration.

> Use one clearly designated text receiver for a simple first screen. Several matching elements can all be filled, which is useful for intentional duplicates but confusing when accidental.

Add portraits without disconnecting lip sync

Group ID	Use it on
@talkdialogue	The current spoken text.
@playerpic	An Image element for the active player’s portrait.
@NPCpic	An Image element for the conversation partner’s portrait.

1. Set up the character’s portrait assets in the Character Designer first.
2. Place two Image elements in the conversation GUI if you want both portraits visible.
3. Give them the exact Group IDs above, keeping the capitalization of @NPCpic.
4. Choose a Content Fit that preserves the portrait’s intended proportions.
5. Run a conversation in both speaker directions and check the correct portrait is shown.
6. Test the character’s configured portrait lip-sync mapping with actual speech. A still decorative Image is not a substitute for the bound portrait receiver.

The runtime fills these portrait Groups and connects them to the configured lip-sync system. Keep the portrait definitions, frame/viseme mapping and speech content consistent with the character setup.

> Replacing the dialogue background should not require rewriting the character’s mouth shapes. Preserve the binding Groups while changing the surrounding artwork.

Make a proper dialogue-choice screen

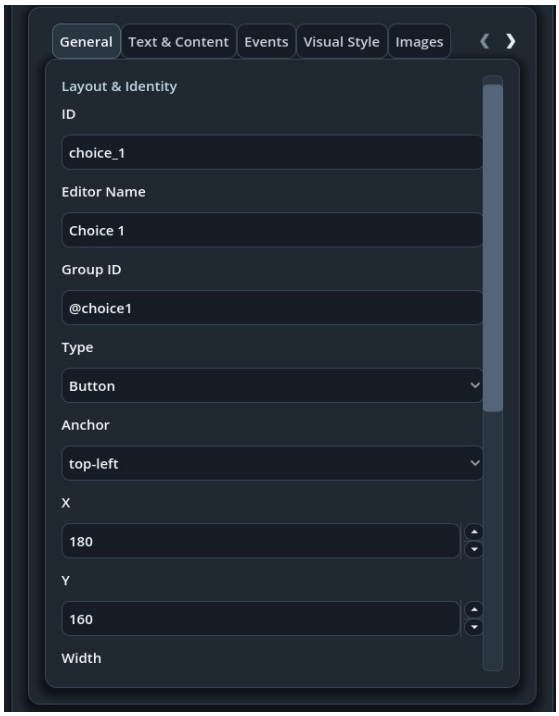

Figure 36. Choice Buttons use numbered Group IDs, which the dialogue system fills and handles.

Group ID	Job
@choice1 ... @choice20	The visible choice slots, numbered consecutively.
@choicescrollup	Show earlier available choices.
@choicescrolldown	Show later available choices.

1. Create choices_custom. Set its opening and closing transitions to Cut for this first working version. Add a readable row of Buttons for each visible choice slot.
2. Start with four Buttons using @choice1, @choice2, @choice3 and @choice4.
3. Add the two scroll controls when the dialogue may have more than four choices.
4. Assign the GUI to Dialogue Choices in Global Settings.
5. Use a real dialogue graph with more choices than the first page can show. Test scrolling and every displayed response.

> The choice Group already connects the Button to its response. Do not also hard-code a different dialogue branch on Click, or you can create two competing routes.

Test choices beyond the first four rows

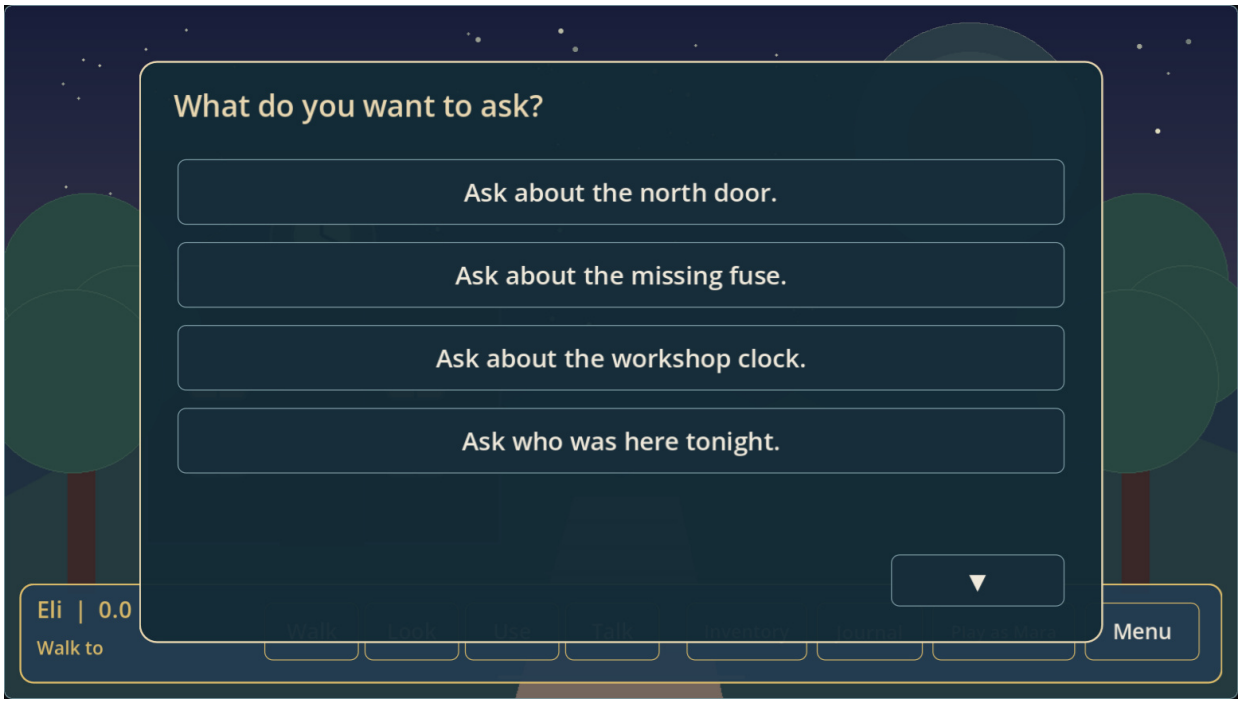

Figure 37. The assigned choice GUI is filled with real dialogue-choice text by the runtime.

1. Create a test dialogue with at least six available choices.
2. Open it through a normal dialogue action. The first four available entries should fill @choice1 through @choice4.
3. Use the later-choices control. Check that the displayed captions change to the later entries.
4. Choose one of those later captions. Confirm that the dialogue follows that choice, rather than a branch tied to the Button’s original placeholder text.
5. Test hidden, disabled and already-read choices if the game uses them.
6. Close or finish the conversation and check ordinary gameplay input returns.

> The Button’s Group ID identifies a visible slot. Its current caption and branch come from the dialogue; a placeholder caption typed into the designer is not the permanent choice.

Replace narrator messages and keep personal speech straight

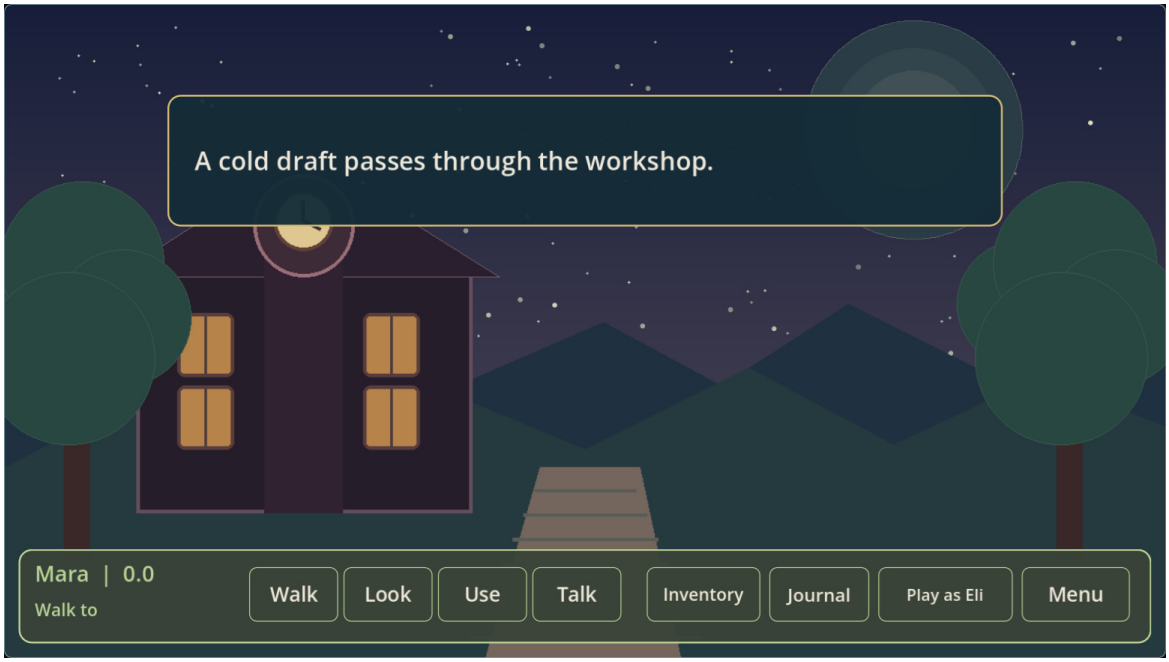

Figure 38. Show Message fills @talkdialogue on the assigned General Narrator Speech GUI.

Create narrator_custom with a background Panel and a text element using @talkdialogue. Assign it to General Narrator Speech. Use a real Show Message action to test its text, typewriter behavior and duration.

For personal dialogue shells, clone the layout and assign tester_dialogue to Eli and assistant_dialogue to Mara. When Mara is controlled, her shell is selected even if the line’s speaker is Eli. The speaker and portrait binding still determine whose words and portrait are shown.

Test	What should be true
Eli controlled; Eli speaks	Eli’s shell and the correct text/portrait.
Mara controlled; Eli speaks	Mara’s shell, with Eli still identified as the speaker.
A narrator message	The narrator GUI, not whichever personal speech shell happens to be active.
Player must choose a response	The assigned choice GUI remains usable.

> The personal dialogue assignment follows the active player. It is not a “give every NPC a different speech box” selector based solely on the current speaker.

8. Replace splash screens and startup presentation

Stage	Controlled by	When it appears
Engine boot image	Godot application boot-splash settings.	Before PCA’s game runtime can show its own screens.
Ordered startup media	Project Settings → Startup Splashes.	The image/video sequence during a normal full launch.
Timed Splash Screen GUI	Global Settings → Splash Screen and Splash Duration.	After the ordered media, before the title menu or normal startup action.
Main Menu	Global Settings → Main Menu; Show Main Menu on Start.	The player chooses New Game, Continue or another route.
Game introduction	Project Settings → Startup Action.	When the normal project startup sequence begins.

These stages can all be present, but they do not have to be. If you already have two logo images in Startup Splashes, an additional timed GUI logo may simply repeat the same presentation.

> Test Scene bypasses the normal startup presentation. Use Play / Restart or the exported game to check splash order and the title menu.

Edit the ordered startup sequence

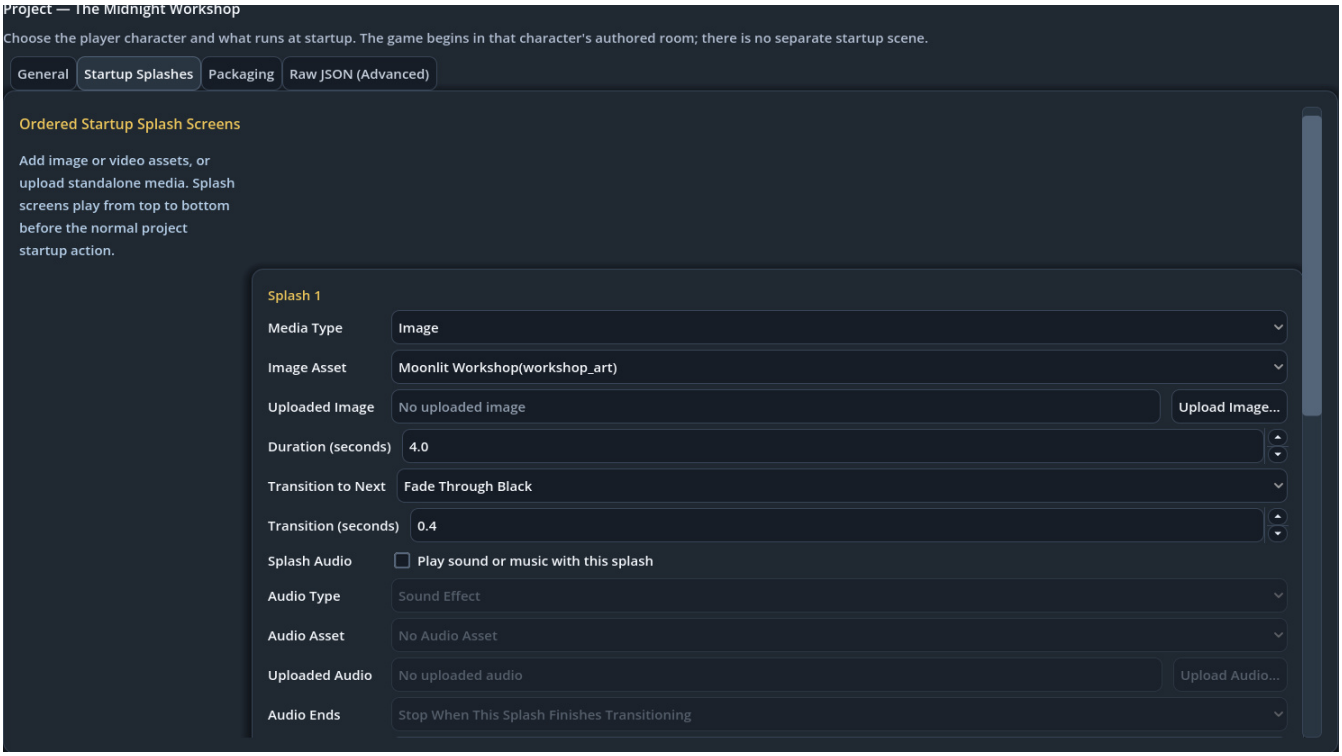

Figure 39. Startup Splashes is an ordered list in Project Settings.

1. Open Project Settings and choose Startup Splashes.
2. Add an entry and choose Image or Video as its Media Type.
3. Select an imported asset, or use Upload Image / Upload Video for standalone media.
4. Set a duration and an appropriate transition to the next entry.
5. Use Move Up and Move Down to put the entries in the intended order.
6. Remove an unwanted entry using its Remove control.
7. Apply Changes and Save Project before testing a full launch.

> Choose a single intended media source for each entry. Clear an obsolete imported asset selection if you are replacing it with an uploaded path, so you do not leave two possible sources to diagnose later.

Choose timing, transitions and skip rules

Setting	What it changes
Duration	How long an image, or a fixed-duration video entry, holds before transition.
Continue When Video Ends	Wait for a non-looping video to finish.
Use Authored Duration	Advance according to the configured duration. Useful when intentionally looping a short video.
Cut	Immediate change.
Fade Through Black	Fade the old visual out, then the next one in.
Crossfade	Blend the two visuals.
Wipe Left / Right / Up / Down	A directional transition between the visuals.
Skip rule	Cannot Be Skipped, Any Key Press, Any Mouse Click, or either input.

A short studio logo rarely needs a long compulsory hold. Start with a readable two- or three-second duration and a brief transition. Use a non-skippable entry only when its content really needs that treatment.

For a looping video, select a fixed authored duration. A loop that never reaches a final frame should not be used as an endless “wait until the video ends” instruction.

> Keep a video’s own audio and any separately assigned Splash Audio intentional. Otherwise the same music can play twice.

Control splash audio deliberately

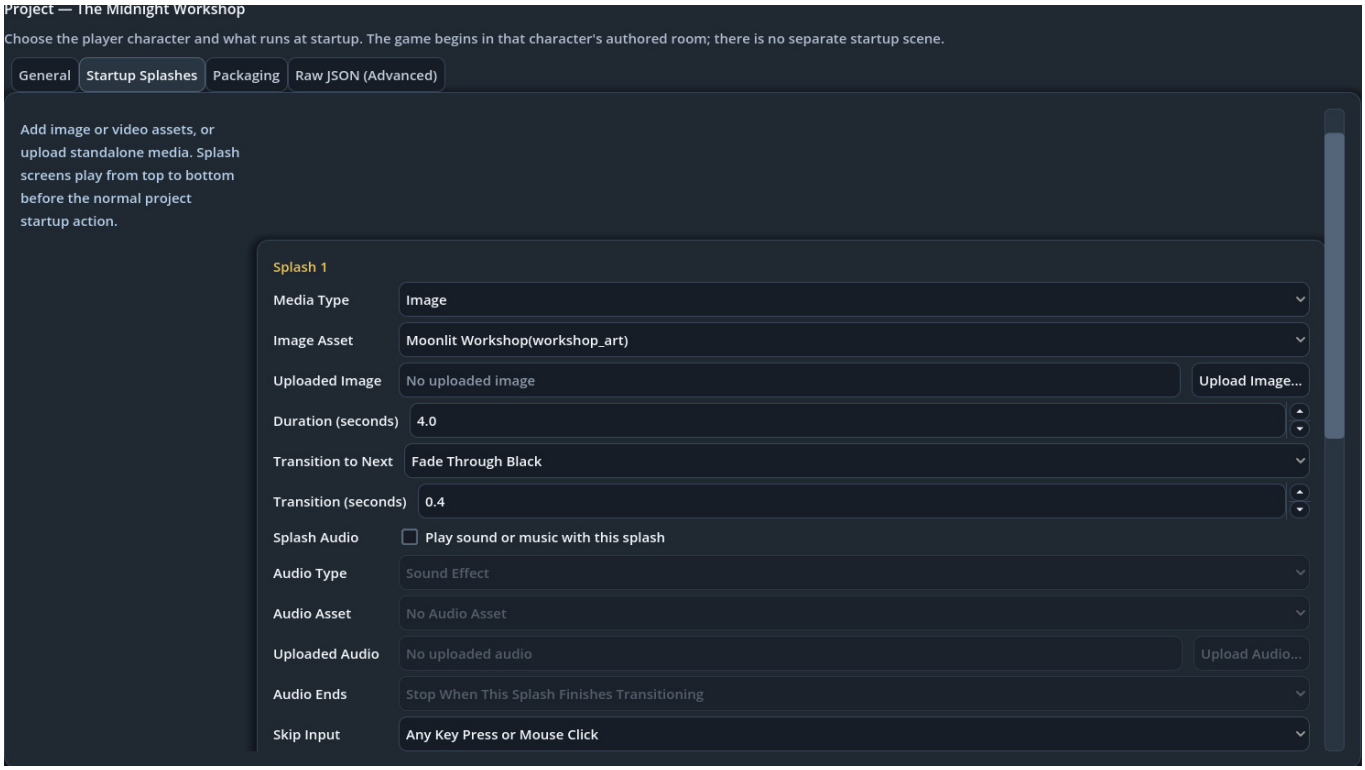

Figure 40. Each splash has its own audio source, audio-end policy and skip rule.

Audio Ends choice	Useful for
Stop When This Splash Finishes Transitioning	A short logo sting that belongs only to that visual.
Keep Playing Until the Audio File Ends	A short file that can continue across the next visual.
Keep Playing Until a Script Stops It	A deliberate musical handoff into the menu or introduction.

Choose Sound Effect or Music according to the intended audio bus. Imported Audio Asset and Uploaded Audio are alternate ways to supply the file.

For the script-stop policy, add Stop Startup Splash Audio at the exact point where that music should end. That action is available to ordinary action lists and the shared scripting surface.

> Test both waiting and skipping. A skipped visual should not leave an unwanted sting or a second copy of the title music playing underneath the menu.

Use a custom GUI as the timed splash



Figure 41. This splash is a real GUI with an emblem and text, not a baked full-screen logo image.

1. Create splash_custom as a hidden GUI.
2. Add a full-canvas background Panel or Image.
3. Add your studio emblem, studio name and any short presentation text.
4. Assign splash_custom to Splash Screen in Global Settings.
5. Set Splash Duration, for example 3.0 seconds.
6. Run a normal full launch. The ordered Startup Splashes play first, then this GUI appears for its configured hold.

This route is convenient when you want editable text, separate image elements or an authored entrance/exit effect. For several independently timed logos, the ordered media list is usually easier to manage.

> The startup-media skip dropdown belongs to those media entries. The timed Splash Screen GUI uses its separate duration; do not assume it inherits the preceding entry’s skip rule.

Put the title menu and introduction in the right order

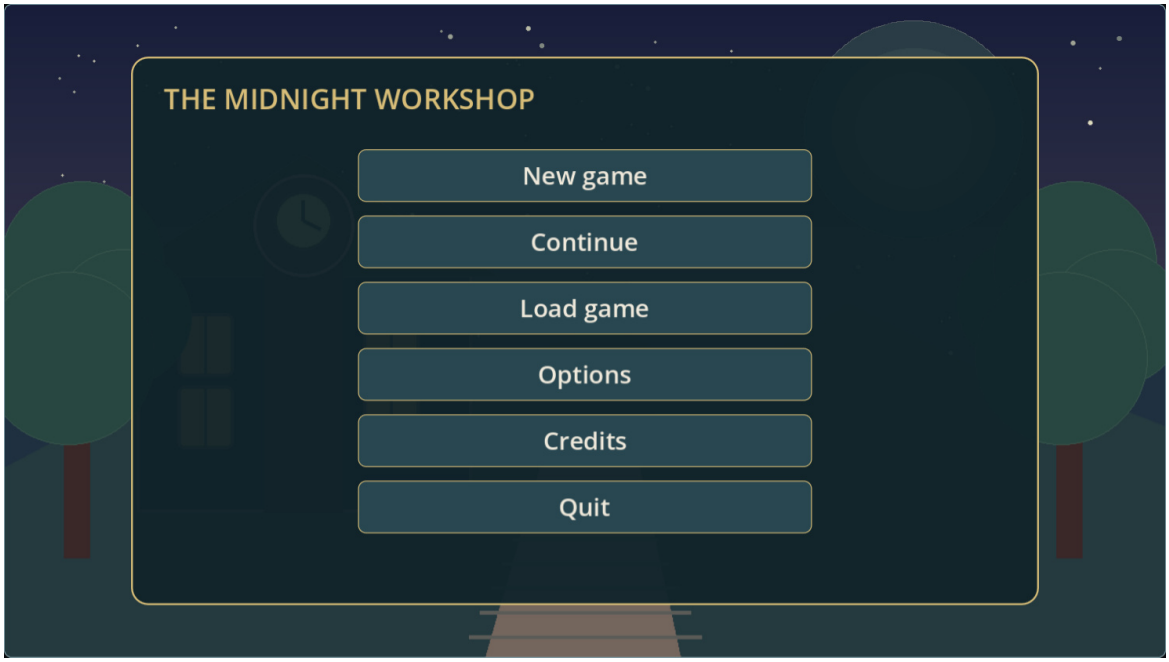

Figure 42. After the splash stages, the configured Main Menu is shown when Show Main Menu on Start is enabled.

Configuration	Normal full-launch flow
Show Main Menu on Start on	Ordered media → timed GUI splash → Main Menu. New Game then begins the configured startup action.
Show Main Menu on Start off	Ordered media → timed GUI splash → configured startup action.
Test Scene	Go directly into the selected scene for testing; skip the ordinary startup presentation.

Startup Action can begin at the chosen player’s location, run a cutscene, or run a script before ordinary gameplay. Keep a story introduction there rather than making the New Game Button manually reproduce a long list of room changes.

> Continue and Load restore saved progress; they should not replay an introduction as though the player had chosen a new game. Test the actual routes rather than attaching the introduction to every menu close.

Replace the engine boot image in an exported source project

A Godot logo seen before the game runtime starts is an earlier boot stage. PCA's Startup Splashes and Splash Screen GUI cannot run soon enough to replace that first image.

1. Export the game's Godot source project through the build tools, keeping it separate from the PCA editor project.
2. Open that generated game project in the matching Godot editor.
3. Open Project → Project Settings and find Application → Boot Splash.
4. Choose your PNG for Image, or turn Show Image off for a plain background color.
5. Set the background color and inspect the available scaling/filtering options.
6. Export that game project through Godot and test a cold launch of the result.

Godot's documented boot image accepts PNG. An empty Image setting uses the default engine image when Show Image is enabled. [1]

> This is an exported-source workflow. Assigning a PCA startup GUI does not patch the boot settings embedded in an already compiled runner. A fresh source export can also regenerate project settings, so preserve your custom export configuration and recheck it after regenerating.

Test startup from a cold launch

1. Close the running exported game completely, then launch it again.
2. Observe the engine boot image before the first PCA splash.
3. Check that the media entries play in their authored order.
4. Wait through the sequence once, then test the configured keyboard and mouse skips on another launch.
5. Check all audio stop/handoff points.
6. Check the optional timed GUI splash and its duration.
7. Choose New Game and confirm the correct starting character and introduction.
8. Return to Main Menu and choose Continue or Load. Check that saved progress is restored without restarting the story.
9. Repeat at the smallest supported window size and the intended fullscreen resolution.

Do not use Test Scene as the final startup check. It is useful precisely because it skips the stages you are trying to verify here.

> Keep only one copy of each intended logo stage. If a logo appears twice, compare Startup Splashes, Splash Screen GUI and the startup cutscene before deleting unrelated media.

9. Finish the presentation without breaking the controls

Part	Useful first choice
Persistent Main / Personal HUD	Cut for opening and closing. Keep character changes and menu handoffs visually stable.
Inventory / Journal	A brief fade or slide, with an obvious close route.
Main / Pause / Options	Start with Cut, then add a short tested transition.
Dialogue choices	Start with Cut. Check hidden/unavailable choices and scroll controls before adding an animated transition.
Button feedback	Separate normal, hover and pressed images or colors.
Selected item / selected entry	A persistent selected state, separate from a momentary press.
Decorative animated artwork	Separate Image frame sequence with consistent canvas dimensions.

The supplied HUD examples use Cut transitions. Keep that baseline when checking personal-HUD exclusivity and system-menu restoration. Add transitions to less permanent surfaces after their routes pass.

If an old HUD remains visible during a menu handoff, first return the HUD’s open and close transitions to Cut and retest. A transition/state issue can look like a wrong character assignment.

> Changing every surface at once makes a display error harder to isolate. Keep a working plain version while you polish one screen at a time.

Keep backgrounds and image states separate

Image or state	Use
GUI window background	Artwork for the whole authored window.
Background Panel / Image element	A deliberate layout layer inside that GUI.
Element background / hover / pressed image	Feedback for a particular interactive element.
Grid background, slot and navigation artwork	Independent surfaces inside inventory/save widgets.
Selected state	An item or entry remains selected after the pointer is released.

A pressed image should return to normal or hover after release. Do not confuse that temporary feedback with a checkbox that remains checked or an inventory item that remains selected.

Keep text and live widgets separate from decorative animation. An opening bag picture can move while the real inventory appears only when the bag is ready to be used.

1. Test normal, hover, held press, release over the control and release outside it.
2. Test the selected state separately.
3. Check Z Index and pointer blocking after adding each new background layer.
4. Check a static or reduced-motion version of the same screen.

Replace other interface entry points

Some controls are ordinary authored GUIs or other project settings, rather than another row in Custom System Interfaces.

Part	Where to work
Text-parser command entry	A single-line Text Box in an authored GUI, with Run Text Parser Command When Enter Is Pressed enabled.
Hint information	A Hint element in an extra GUI, with your own opener and close route.
Target-first verb popup	Project Settings → Verb Direct Popup GUI; choose an authored GUI with real Set Verb Buttons.
Game mouse cursors	Cursor Designer definitions and their verb assignments; check the game’s custom-cursor settings.

1. For a parser interface, enable the parser in the project’s Text Parser configuration first.
2. Create one shared command-entry GUI and add a single-line Text Box. Enable the Enter parser command setting.
3. Choose Clear Single-Line Text When Enter Is Pressed if the command should disappear after submission. Choose Return Focus To Text Box After Enter to keep the next command easy to type.
4. Give touch users a visible opener and test the on-screen keyboard in the actual game.

Keep one enabled shared parser receiver for this first setup. The runtime’s focus shortcut searches authored parser Text Boxes; several separate personal receivers can make that search choose a different GUI than you intended.

> A target-first verb popup and the adaptive control system’s stock interaction coin are separate mechanisms. Do not assume that styling one automatically reskins the other.

Check scaling, focus and mobile access

Situation	Check
Small window	Captions, journal reader and menu Buttons still fit.
Wide or tall display	The GUI’s actual window bounds and anchors match your intended presentation.
Touch	Inventory and journal have visible openers; no essential feature depends only on hover.
Controller / keyboard	Focus order reaches every command; selected and focused states are visible.
Long translation	Names do not disappear behind portraits, rounded corners or navigation arrows.
High contrast / large text	Custom authored fonts, artwork and layout genuinely meet the chosen presentation.

A window’s root bounds, a background Panel and an inventory widget can all have different sizes. When a screen opens from an edge or clicks seem to hit empty space, inspect the root window as well as the artwork.

> A screenshot is one state. Test the same screen empty, full, selected, scrolled and reopened before calling the replacement finished.

10. Install the working script helpers

These helpers are project scripts. Create them in the GDScript Designer; do not paste them into the PCA engine’s source files. They use the reference build’s runtime GUI classes, so keep the element IDs consistent with the layouts.

1. Create the GUI records, element IDs and variables described in the preceding chapters.
2. Open the GDScript Designer and add a script record with the stated ID.
3. Replace its source with the complete example. For a multi-part listing, paste the parts into the same record in order, with only one extends line.
4. Enable the record. Turn on Global Script only where the instructions say to.
5. For journal_tools, also subscribe to OnPlayerChanged and OnSaveLoaded in Event Functions.
6. Apply Changes, run Validate and save the project.
7. On an ordinary GUI or story event, select Run Script/Run Visual Script and choose the small wrapper record that performs the intended operation.

Global controller	Purpose
menu_tools	Parent routes for Save/Load, Continue availability, selected-slot guard.
options_tools	Load, stage, apply and persist the custom settings controls.
journal_tools	Keep personal notes and display the selected entry in a read-only reader.

A Global Script stays available while the game runs. An ordinary wrapper runs when called. Checking an event subscription is separate from making a script global.

Open the real pause-menu route

Record ID: open_pause. Global Script: off. Put it on the HUD’s Menu Button.

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

func run(_context: Dictionary = {}) -> Variant:
    Runtime.call("show_pause_menu")
    return true
```

The runtime chooses the assigned Pause Menu, so the Button continues to work if you replace pause_custom later. A direct Show GUI action would name a particular layout instead.

Other ordinary operations	Use the GUI behavior
Resume gameplay	System Resume
Start the configured new game	System New Game
Restore the newest saved position	System Continue Game
Open the title menu	System Main Menu
Exit the exported game	System Quit

Keep this wrapper small. Inventory and journal openers already have their own Toggle Inventory / Toggle Journal behaviors.

Menu controller - part 1 of 2

Record ID: menu_tools. Global Script: on. Paste Part 2 after Part 1 in the same record. No event subscriptions are needed for this controller.

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

var return_surface: String = "pauseMenu"
var was_main: bool = false

func on_global_ready(_context: Dictionary = {}) -> void:
    set_process(true)

func _process(_delta: float) -> void:
    var manager: Node = Runtime.get("authored_gui_manager")
    if not is_instance_valid(manager):
        return
    var main_open: bool = bool(manager.call("is_gui_visible", "main_custom"))
    if main_open and not was_main:
        var nodes: Dictionary = manager.get("element_nodes")
        var button: BaseButton = nodes.get("main_custom_button_1") as BaseButton
        if is_instance_valid(button):
            button.disabled = int(Runtime.call("latest_save_slot")) < 0
    was_main = main_open
```

The controller checks Continue when main_custom becomes visible. It scans saved positions once at that opening, rather than repeatedly checking the disk every frame.

main_custom_button_1 is the Continue Button in this layout. If you give it a clearer ID in your project, change that one lookup too.

Menu controller - part 2 of 2

```
func open(surface: String, from_surface: String = "pauseMenu") -> void:
    return_surface = from_surface
    Runtime.call("show_system_interface", surface)

func back() -> void:
    Runtime.call("show_system_interface", return_surface)

func slot_command(gui_id: String, element_id: String, command: String) -> void:
    var slot: int = int(GetGUIElementProperty(
        gui_id, element_id, "selected_slot", -1))
    if slot < 1:
        await SetVariable({"name":"save_status", "value":"Choose a slot first."})
        return
    await SaveGameWidgetCommand({"guiId":gui_id, "elementId":element_id,
        "command":command, "slot":slot})
```

open remembers the parent system surface. back opens that parent. slot_command uses the selected slot on the actual widget and returns without saving/loading if no position has been chosen.

The message uses the save_status string variable. Put $$save_status in a Label on both Save and Load screens so the guard produces useful feedback.

> The explicit command names are save-slot and load-slot. A selected-slot fallback is deliberately avoided here.

Options controller - part 1 of 4

Record ID: options_tools. Global Script: on. All four parts belong in this one record. Create the options_notice string variable used for short status messages.

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

const GUI_ID: String = "options_custom"
const NUMBERS: Dictionary = {
    "masterVolume":["opt_master", 0.0, 1.0],
    "musicVolume":["opt_music", 0.0, 1.0],
    "sfxVolume":["opt_sfx", 0.0, 1.0],
    "voiceVolume":["opt_voice", 0.0, 1.0],
    "textSpeed":["opt_text", 0.1, 5.0],
    "fontScale":["opt_font", 0.5, 2.5]
}
const CHECKS: Dictionary = {
    "subtitles":"opt_subtitles", "highContrast":"opt_contrast",
    "reducedMotion":"opt_motion", "fullscreen":"opt_fullscreen",
    "gameAutosave":"opt_autosave"
}
var return_surface: String = "pauseMenu"
var was_open: bool = false

func on_global_ready(_context: Dictionary = {}) -> void:
    set_process(true)

func control(element_id: String) -> Control:
    var manager: Node = Runtime.get("authored_gui_manager")
    if not is_instance_valid(manager):
        return null
    var nodes: Dictionary = manager.get("element_nodes")
    return nodes.get(element_id) as Control

func _process(_delta: float) -> void:
    var manager: Node = Runtime.get("authored_gui_manager")
    if not is_instance_valid(manager):
        return
    var open_now: bool = bool(manager.call("is_gui_visible", GUI_ID))
    if open_now and not was_open:
        fill(Runtime.call("get_runtime_options"))
    was_open = open_now
```

The dictionaries map runtime option keys to your element IDs. The small visibility check fills the controls if another route opens the Options GUI.

Continue with the next part in the same options_tools record. Do not add another extends line.

Options controller - part 2 of 4

```
func open(from_surface: String = "pauseMenu") -> void:
    return_surface = from_surface
    Runtime.call("show_system_interface", "optionsMenu")
    fill(Runtime.call("get_runtime_options"))
    was_open = true
    await SetVariable({"name":"options_notice",
        "value":"Change settings, then choose Apply."})

func fill(options: Dictionary) -> void:
    for key: String in NUMBERS:
        var slider: Range = control(str(NUMBERS[key][0])) as Range
        if is_instance_valid(slider):
            slider.set_value_no_signal(float(options.get(key, 1.0)))
    for key: String in CHECKS:
        var check: BaseButton = control(str(CHECKS[key])) as BaseButton
        if is_instance_valid(check):
            check.set_pressed_no_signal(bool(options.get(key, false)))
    select_language("opt_language", str(options.get("language", "en")))
    var voice: String = str(options.get("voiceLanguage", ""))
    select_language("opt_voice_language", "auto" if voice.is_empty() else voice)
    var autosave: BaseButton = control("opt_autosave") as BaseButton
    if is_instance_valid(autosave):
        autosave.disabled = not bool(Runtime.call("can_player_toggle_game_autosave"))
```

open records the parent menu. fill uses the applied runtime values, sets controls without emitting an artificial user change, and respects the project’s autosave-toggle permission.

Continue with the next part in the same options_tools record. Do not add another extends line.

Options controller - part 3 of 4

```
func select_language(element_id: String, code: String) -> void:
    var list: PCAListBox = control(element_id) as PCAListBox
    if not is_instance_valid(list):
        return
    list.selected_ids.clear()
    list.selected_ids.append(code)
    list.refresh()

func selected_language(element_id: String, fallback: String) -> String:
    var list: PCAListBox = control(element_id) as PCAListBox
    if not is_instance_valid(list) or list.selected_ids.is_empty():
        return fallback
    return list.selected_ids[0]
```

Language selections are List Box item IDs. The voice item auto is a GUI convenience; it is converted to an empty voice-language override when applying.

Continue with the next part in the same options_tools record. Do not add another extends line.

Options controller - part 4 of 4

```
func apply() -> void:
    var options: Dictionary = Runtime.call("get_runtime_options")
    for key: String in NUMBERS:
        var spec: Array = NUMBERS[key]
        var slider: Range = control(str(spec[0])) as Range
        if is_instance_valid(slider):
            options[key] = clampf(slider.value, float(spec[1]), float(spec[2]))
    for key: String in CHECKS:
        var check: BaseButton = control(str(CHECKS[key])) as BaseButton
        if is_instance_valid(check) and not check.disabled:
            options[key] = check.button_pressed
    options["language"] = selected_language("opt_language", "en")
    var voice: String = selected_language("opt_voice_language", "auto")
    options["voiceLanguage"] = "" if voice == "auto" else voice
    Runtime.call("apply_runtime_options", options, true)
    fill(Runtime.call("get_runtime_options"))
    await SetVariable({"name":"options_notice", "value":"Options saved."})

func defaults() -> void:
    fill(Runtime.get("project_default_options"))
    await SetVariable({"name":"options_notice",
        "value":"Defaults selected. Choose Apply to save them."})

func back() -> void:
    Runtime.call("show_system_interface", return_surface)
```

apply preserves unrelated options and persists the changes. defaults stages the project defaults in the controls. back changes system surfaces without applying the staged values.

> Use the small wrappers on the following page to call these functions from ordinary GUI events.

Create the menu and options wrappers

Each row below is a separate ordinary GDScript record. Global Script is off. Use the template once per row, replacing the three arguments to CallScript with the values in the table.

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

func run(_context: Dictionary = {}) -> Variant:
    await CallScript("options_tools", "open", ["pauseMenu"])
    return true
```

Record ID	Controller / function / arguments
open_pause_options	options_tools / open / ["pauseMenu"]
open_main_options	options_tools / open / ["mainMenu"]
options_apply	options_tools / apply / []
options_defaults	options_tools / defaults / []
options_back	options_tools / back / []
open_main_load	menu_tools / open / ["loadMenu", "mainMenu"]
open_pause_load	menu_tools / open / ["loadMenu", "pauseMenu"]
open_pause_save	menu_tools / open / ["saveMenu", "pauseMenu"]
menu_back	menu_tools / back / []

The complete template shown is open_pause_options. Keep quotes around string IDs and square brackets around the arguments array.

Create the selected-slot command wrappers

Record ID: save_selected. Global Script: off. Put it on save_custom’s Save selected Button.

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

func run(_context: Dictionary = {}) -> Variant:
    await CallScript("menu_tools", "slot_command", ["save_custom", "save_custom_slots", "save-slot"])
    return true
```

Record ID: load_selected. Global Script: off. Put it on load_custom’s Load selected Button.

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

func run(_context: Dictionary = {}) -> Variant:
    await CallScript("menu_tools", "slot_command", ["load_custom", "load_custom_slots", "load-slot"])
    return true
```

The guard reads the widget’s live selected_slot, not a guessed row number and not a label caption. Give the player a visible selected border before allowing Save or Load.

The project must contain both named GUIs and both named widgets. Validate can catch some broken references; a Play test still has to confirm the correct selected position is used.

Personal notebook controller - part 1 of 3

Record ID: journal_tools. Global Script: on. Subscribe to OnPlayerChanged and OnSaveLoaded. All three parts belong to the same record.

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

var previous: String = ""

func on_global_ready(_context: Dictionary = {}) -> void:
    set_process(true)
    render()

func on_player_changed(_context: Dictionary = {}) -> void:
    await SetVariable({"name":"journal_choice", "value":""})
    render()

func on_save_loaded(_context: Dictionary = {}) -> void:
    render()

func _process(_delta: float) -> void:
    var values: Dictionary = Runtime.get("variables")
    var current: String = ActivePlayer() + "|" + str(values.get("journal_choice", ""))
    if current != previous:
        render()
```

The player-change hook clears the old selected entry ID. The save-loaded hook redraws the reader from restored data. The small frame check notices the player selecting another title.

The event names in the subscription controls correspond to the on_player_changed and on_save_loaded functions. Defining those functions without the subscriptions is incomplete setup.

Personal notebook controller - part 2 of 3

```
func render() -> void:
    if not is_instance_valid(Runtime):
        return
    var values: Dictionary = Runtime.get("variables")
    var actor: String = ActivePlayer()
    var selected: String = str(values.get("journal_choice", ""))
    previous = actor + "|" + selected
    var body: String = "Choose an entry to read it."
    var entries: Array = values.get("journal_" + actor, [])
    for entry: Dictionary in entries:
        if str(entry.get("id", "")) == selected:
            body = str(entry.get("text", "")) + "\n\n" + str(entry.get("body", ""))
            break
    var manager: Node = Runtime.get("authored_gui_manager")
    if not is_instance_valid(manager):
        return
    var nodes: Dictionary = manager.get("element_nodes")
    for owner: String in ["tester", "assistant"]:
        var reader: TextEdit = nodes.get(owner + "_journal_body") as TextEdit
        if not is_instance_valid(reader):
            continue
        reader.editable = false
        if owner == actor:
            reader.text = body
```

The reader displays the selected entry’s title and body. Its editing is disabled, while normal scrolling remains available. A character with no selected entry gets a plain instruction instead of another character’s previous text.

The owner list contains tester and assistant. Extend it, and create the matching reader IDs, when adding more playable characters.

Personal notebook controller - part 3 of 3

```
func add_entry(entry_id: String, title: String, body: String,
        character_id: String = "") -> void:
    var owner: String = ActivePlayer() if character_id.is_empty() else character_id
    var values: Dictionary = Runtime.get("variables")
    var key: String = "journal_" + owner
    var entries: Array = values.get(key, []).duplicate(true)
    var found: bool = false
    for entry: Dictionary in entries:
        if str(entry.get("id", "")) == entry_id:
            entry["text"] = title
            entry["body"] = body
            found = true
            break
    if not found:
        entries.append({"id":entry_id, "text":title, "body":body})
    await SetVariable({"name":key, "value":entries})
    render()
```

add_entry duplicates the saved array, updates an existing matching ID or adds a new dictionary, then writes the array through Set Variable. The displayed list refreshes from that saved source.

Omitting character_id writes to ActivePlayer(). Supplying another reusable character ID writes to that owner’s array. Declare the intended owner arrays in the Variable Designer.

This is a saved personal-notes system. It does not change the engine’s shared quest manager or create independent per-character quest-state objects.

Call the notebook helper from a normal event

Record ID: remember_door. Global Script: off. Put this on the north door’s discovery action, or call it from an action list or a visual script.

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

func run(_context: Dictionary = {}) -> Variant:
    await CallScript("journal_tools", "add_entry", [
        "north_door", "The locked north door",
        "The brass key fits the lock. I should ask Mara about the clock."])
    return true
```

To teach Mara this fact directly, add "assistant" as the fourth item in the argument array, after the body string. To teach both characters, call add_entry once for each intended owner.

Parameter	What to keep straight
entry_id	Stable ID of the fact, such as north_door.
title	Short readable list caption.
body	The longer passage in the reader.
character_id	Optional owner; blank means the current active player.

For a Functions as Nodes workflow, expose add_entry with these parameters using your function definitions. The saved array format and ownership rule remain the same.

Use the helpers from visual scripting and action lists

1. Create the normal wrapper record first and test it with the GDScript Designer’s Validate control.
2. Open the object, GUI event or action-list editor where the operation belongs.
3. Add Run Script / Run Visual Script and select the wrapper by its name and ID.
4. Put the call after any condition that determines whether the operation is allowed.
5. For a sequence such as “learn the clue, add a note, then speak,” keep those actions in that order.
6. Use Play to test the actual event rather than relying only on the script’s Run button.

The shared wrapper route avoids rebuilding the same menu or notebook logic in several event editors. A visual graph can call remember_door; a GUI Button can call the same record; an ordinary script can call the controller function directly.

If you expose controller functions as visual nodes, keep the node’s parameter types and defaults clear. Character and GUI parameters should contain reusable IDs, while title and body parameters contain display text.

> Keep the controller IDs stable. Renaming a visible menu title is harmless; silently renaming options_tools or journal_tools breaks calls that still use the old ID.

11. Run the complete replacement-screen test

Test	Expected result
New full launch	Correct splash order, title menu and starting character.
Character switch	One personal HUD; correct bag and notebook owner.
Inventory transfer	Source loses and recipient gains the correct quantity.
Private knowledge	A discovery belongs only to the intended character until deliberately shared.
Save / load	Scene, active character, items and both notebooks restore.
Options Apply	Values affect the actual runtime and persist in the standalone game.
Options Back	Unapplied edits are discarded; return to the correct parent menu.
Dialogue / choices	Correct text, portrait Groups and selectable branches.
Pause / Resume	World input and the correct HUD return through the system route.
Standalone Quit / fullscreen	The game application responds without relying on editor-preview behavior.

Use disposable save slots for testing. Keep one known-good game copy while replacing the remaining screens.

Troubleshoot assignments and character state

Symptom	Check first
The default HUD is still visible	Main HUD assignment, Apply Changes, and any personal HUD override.
Both HUDs seem visible	Unique personal assignments; use Cut transitions as the baseline; restart Play.
The custom bag always belongs to Eli	A hard-coded Show GUI opener or a fixed-character owner setting.
Both bags show the same items	Project Settings → Inventory Mode may still be Shared.
Mara cannot be selected	Playable/switch-enabled flags, scene placement and active input locks.
Different journal covers show the same quests	That is the standard shared Journal source. Use private arrays for separate notes.
Private notes vanish after loading	Saved arrays, correct source IDs and OnSaveLoaded subscription.
A note duplicates every time	Use a stable entry ID; call add_entry rather than appending blindly.

For a return to the built-in presentation, test the specific role in a project copy. Main HUD has an explicit built-in/default choice. Inventory and Journal also retain legacy widget-discovery paths in this build; clearing an assignment can still find an existing dedicated widget GUI.

> Assign the intended Inventory and Journal GUI explicitly in a project with several candidates. Do not rely on the first matching widget being the one you meant.

Troubleshoot menus, options and startup

Symptom	Check first
Options Sliders move but audio does not	Apply wrapper, exact control IDs, 0-to-1 volume range and actual audio bus routing.
Options reopen at preview values	Global controller enabled; use the opening wrapper; fill from get_runtime_options().
Back goes to the wrong menu	Correct main/pause opener and remembered parent surface.
Save writes without a chosen card	Use the supplied selection guard and explicit slot command.
Menu closes but input remains blocked	Use System Resume, not only Hide GUI; check another modal screen or prompt.
Quit / fullscreen appears ineffective in Play	These deliberately behave differently in the embedded preview. Test standalone.
Spoken text never updates	Correct system role plus @talkdialogue in Group ID.
Choices look right but cannot select branches	Consecutive @choice Groups, assigned Dialogue Choices GUI, no competing Click action.
Splashes never appear in testing	Use full Play / Restart rather than Test Scene.
An early Godot logo remains	That is the boot stage; use the exported-source boot-splash workflow.

Reference notes and a finished baseline

The PCA steps, field names and scripting examples were checked against PCA 3D Adventure Studio 0.86.3.4. Screenshots in this book are actual editor and Play captures. The private capture scene supplies only the setting in which the interfaces can be seen; following the guide does not require a companion project.

Reference area	Checked behavior
Global Settings / Character Designer	System roles, personal surfaces and playable-character options.
Authored GUI runtime	Groups, widget ownership, menu behaviors and live control types.
Game runtime / save system	Character switching, options application/persistence, startup order and saved variables.
Quest / Journal implementation	Shared quest state versus separately stored personal notes.

[1] Godot documentation - ProjectSettings: Boot Splash

Official reference for the early boot image, PNG format, background color and Show Image behavior.

Keep a finished baseline containing one full startup, two playable characters, separate bags and notebooks, and a complete save/options/menu loop. Make visual changes against that known-good behavior.

> One PDF, no practice files required. The examples are project scripts and authored GUIs; the PCA engine code is not changed.