

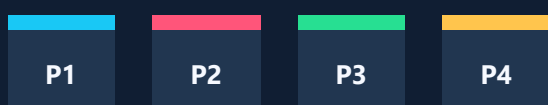


Couch Co-op Builder

User Manual

Version 1.0.0 · Unity 6.3 LTS

Local multiplayer sessions, authored lobbies,
gameplay input and independent cameras.



Contents

Use the clickable chapter and section links to navigate this manual. Unity asset paths are provided so you can locate the referenced files in the Unity Project window.

1. Introduction

Player scope

2. Requirements, Installation and Quick Start

Installation

Quick Start

Default lobby controls

3. Setup Wizard

Setup Wizard

Outputs and completion

Run setup again

4. Character Catalog and Previews

Create or choose a catalog

Validation and selection state

Preview host

Public API

Prefab and fallback resolution

Framing

Camera and resource isolation

Gameplay prefab fallback limitations

Lobby integration

5. Lobby Templates and Existing Canvas

Supplied lobby presentations

Existing Canvas

Input and navigation wiring

Visual ownership and previews

Apply, extend and check

6. Player Sessions and Identity

Scene handoff and authority

Public access

7. Gameplay Representation and Spawning

Scene setup

Notifications and lifetime

Spawn Selected Character

8. Gameplay Input Integration

Lobby input and controller profiles

Gameplay input binding

9. Input Binding Profiles and Rebinding

Identity and shared profiles

Service and storage

Settings before joining

Capture, paths and conflicts

Reset and action schema updates

Simultaneous action conflicts

10. Shared and Split Screen Cameras

1. Overview

2. Shared vs Split Screen

3. Setup Wizard quick start

4. Manual setup

5. Runtime toggle and layouts

6. Built-in layouts 1-8

7. Player camera sources

8. Camera ownership

9. Drop Out / Rejoin and permanent Leave

10. DeviceLost

11. AudioListener

12. UI / HUD

13. URP and Cinemachine

14. Custom layout provider

15. Performance

16. Samples

17. Troubleshooting

11. Persistent Sessions

Open and run

What persists

Concepts to study

12. Device Lost, Drop Out and Rejoin

Reserved controller replacement during a session

Optional gameplay Drop Out and Rejoin

13. Configure Supplied Samples

Preview and explicitly apply
Unity 6 Build Profiles and shared scenes
Reverting

14. Sample Index and Integration Examples

Sample index
Gameplay examples

15. Rocket Rumble Menus, Settings and Lifecycle

Main Menu and lobby
Audio settings
Controls settings
Pause and Device Lost
Drop Out and Rejoin
What to study and adapt

16. Rocket Rumble 2D

Open and play
Gameplay controls
Arena and jump
Shield, respawn and match
Integration to study
Try your integration

17. Rocket Rumble 3D

Open and play
Gameplay controls
Arena, movement and jump
Shield, respawn and match
Integration to study
Try your integration
Optional Split Screen cameras

18. Split Screen Playgrounds

Try it
Integration to read
Presentation and limits
Try your integration

19. Included Smart UI Focus & Navigation

Manual and components
Samples
Owning both products and updates

20. Troubleshooting

21. Advanced API Integration

Owned analog selector navigation
Public integration map

22. Support and Resource Notices

Included content
Unity resource dependencies

CHAPTER 01

Introduction

Couch Co-op Builder configures local multiplayer joining, independent lobby navigation, character selection and gameplay handoff for Unity uGUI and the Input System. Optional integrations provide owned gameplay actions, saved binding profiles and Shared Camera / Split Screen management. Your game supplies its UI design, gameplay rules, movement and camera follow behavior.

Start with **Tools > LevelArgento > Couch Co-op Builder > Setup Wizard**. Smart UI Focus & Navigation is included; no separate purchase is required.

Player scope

Workflow / example	Capacity
Framework and Existing Canvas	1–8 local players; typically 2–8 for co-op
Supplied authored lobby templates	Up to 4 visual player slots
Rocket Rumble 2D / 3D	2–4 players
Split Screen Playgrounds	1–8 players

For five to eight lobby players, provide your own P1–P8 visual roots and use Existing Canvas. The Wizard wires those authored controls; it does not generate additional graphics. Both the lobby and your gameplay scene must accommodate the chosen capacity. More players/cameras require profiling your own content and hardware.

CHAPTER 02

Requirements, Installation and Quick Start

Installation

Use **Unity 6.3 LTS**, **Unity Input System**, and **Unity UI (uGUI / TextMeshPro)**. The supplied visual and game samples use the **Universal Render Pipeline (URP)**. Install the Unity Registry packages through Package Manager and enable the Input System in Player Settings. Core session, input, lobby and gameplay integration is not inherently tied to URP. Smart UI Focus & Navigation is included.

Before opening UI samples in a fresh project, use **Window > TextMeshPro > Import TMP Essential Resources**. Unity UI supplies the TMP APIs; this separate import supplies the default fonts, materials, shaders and TMP Settings used by the examples. Couch does not bundle those standard resources. If compatible Essential Resources are already present, keep them and your TMP configuration; do not overwrite them unnecessarily. No TMP Examples & Extras import is required. This is a one-time customer setup step.

Quick Start

1. **Open Setup Wizard.** Choose **Create New Lobby** for an editable supplied presentation, or **Integrate Existing Canvas** for your own saved UI.
2. **Characters:** assign a Character Catalog and **Maximum Local Players**. Use **Generate from Prefabs...** if you have no catalog: add your playable prefab assets, choose an existing Assets folder, **Review Plan**, then **Generate Reviewed Assets**. Assign the resulting catalog back in the Wizard.
3. **Lobby UI:** choose a supplied style, an existing customer-owned Assets folder and a new asset name. For Existing Canvas, assign its shared Start Button and the required controls for each player.
4. **Gameplay:** choose your saved Gameplay Scene, enabled in the project scene configuration. **Open Gameplay Scene Additively** lets you inspect its Gameplay Player Setup. Configure representation and input on those existing components using [Chapter 7 — Gameplay Representation and Spawning](#) and [Chapter 8 — Gameplay Input Integration](#). API-only consumers can use their own game integration.
5. **Cameras:** leave management disabled to keep your cameras, or **Enable Shared / Split Screen Camera Management** and assign the required gameplay references. Your game supplies camera follow/framing; see [Chapter 10 — Shared and Split Screen Cameras](#).
6. **Review and test:** resolve blocking issues, review output paths, scene-file changes and project scene-list changes, then **Apply Setup**. Use **Open Lobby Scene**, save, and **Test Lobby in Play Mode**. Join, choose characters, confirm every joined player, then submit the shared Start Game control.

The Wizard copies a supplied scene/prefab or wires your Existing Canvas. It configures the catalog, capacity and gameplay destination; optional camera wiring uses an existing gameplay setup. It does not create gameplay movement or configure input adapters. Save existing lobby/gameplay scene edits after Apply. Scene wiring supports Undo; created assets and project scene-list changes must be reviewed separately when reverting.

Default lobby controls

Source	Join	Browse characters	Confirm	Leave
Keyboard A	Q	A / D	F	G
Keyboard B	P	Left / Right	Enter	Backspace
Gamepad	Start	Left stick / D-pad	South face button	Select / Share / Create
Generic joystick / supported HID	Controller Profile Join	Profile navigation	Profile Confirm	Profile Leave

Down moves focus toward shared Start in the supplied lobbies (S for Keyboard A, Down for Keyboard B). Moving UI focus does not clear Ready. Start requires at least one joined player, all joined players Ready, and their input available; Rocket Rumble additionally requires two players. Two logical keyboard roles share one physical keyboard. More players need compatible controllers. Gameplay controls are configured separately and can differ from lobby controls.

CHAPTER 03

Setup Wizard

New to Couch Co-op Builder? Begin with [Chapter 2 — Requirements, Installation and Quick Start](#). This guide explains setup actions, saving and reruns, then points to concrete examples. Player capacity is explained in [Player scope](#); default lobby controls are in [Chapter 2](#).

Setup Wizard

Open **Tools > LevelArgento > Couch Co-op Builder > Setup Wizard**. Its six steps are **Start**, **Characters**, **Lobby UI**, **Gameplay**, **Cameras**, **Review**, followed by completion.

Step	What to choose / provide
Start	Create New Lobby copies supplied authored UI; Integrate Existing Canvas wires your saved UI. No setup preset is saved.
Characters	A nonempty Character Catalog, stable unique Character IDs, and Maximum Local Players . Generate from Prefabs... opens the generator; assign the resulting catalog afterward.
Lobby UI	New Lobby: Individual Panels, Shared Stage or Character Carousel, an existing Assets folder and a new Lobby Asset Name . Existing Canvas: assign authored controls as described in Chapter 5 — Lobby Templates and Existing Canvas .
Gameplay	Saved Gameplay Scene , enabled in the project scene list. The current Wizard button Add Gameplay Scene to Build Settings registers it immediately through <code>EditorBuildSettings.scenes</code> ; this changes project scene configuration, not the scene asset. Use Open Gameplay Scene Additively and Select Gameplay Player Setup in Inspector for representation configuration. Gameplay input is a separate choice on each Gameplay Player Binding.
Cameras	Optional Enable Shared / Split Screen Camera Management . Assign your gameplay setup and camera source; choose 2P/3P layouts. 4–8 are automatic. See Split Screen Cameras .
Review	Correct blocking errors; inspect paths, scene wiring, asset saves and project scene-list changes, then Apply Setup .

The built-in character-selection lobby requires a catalog that passes **Run Catalog Validation**, including Gameplay Prefabs. The Wizard can show a missing-prefab warning without blocking Apply; that warning does not mean lobby confirmation will work. Assign those prefabs before testing character selection. API-only gameplay metadata lookup has a separate, looser contract; see [Gameplay Integration](#).

Outputs and completion

Create New Lobby copies the chosen authored prefab to your output folder and creates a scene using that editable copy. It assigns catalog/capacity/destination, disables unused supplied slots, and registers the lobby in the project scene list. Source templates remain unchanged. Existing scene/prefab paths are never overwritten. Choose an existing folder under Assets using **Browse...**; create your own folder in the Project window first if needed.

Integrate Existing Canvas wires required references and adds/reuses nonvisual services, input contexts and previews. It preserves authored positions, fonts, colors, sprites and layout. The lobby scene is marked dirty, not automatically saved. Optional camera wiring also marks the gameplay scene dirty; save it after Apply. Missing/duplicate required references and wrong-scene/hierarchy assignments are reported before applying.

Completion lists the applied workflow, capacity/catalog, lobby/gameplay paths and integration choices. Use **Open Lobby Scene**, save, then **Test Lobby in Play Mode**; the test action is enabled only when that lobby is active. Select the catalog or the copied prefab when customizing content. Open gameplay and follow [Chapter 7 — Gameplay Representation and Spawning](#) and [Chapter 8 — Gameplay Input Integration](#) to supply your game-owned objects, movement and input. Confirm all joined players and test shared Start and return to the originating lobby.

Scene wiring supports **Edit > Undo**. Created disk assets and project scene-list registration are separate saved changes, not guaranteed to reverse with scene Undo. Review identifies any operation that saves other dirty assets. Failed New Lobby creation removes its own partial outputs.

Run setup again

Start Another Setup clears only temporary Wizard choices. It does not remove previous scenes/assets. A new lobby needs a new name/folder. To update a configured lobby, choose Existing Canvas, assign its Canvas, and review the suggested player bindings; explicit catalog/capacity choices are preserved. Reapplying reuses bound services, EventSystems and preview contexts and deduplicates project scene-list registration. To extend capacity, author additional complete roots, raise Maximum Local Players and bind the new entries before applying.

Legacy Text and TextMeshPro (TMP) are separate assignments: use one implementation per label. Optional labels/graphics may remain empty. TMP labels need an authored font asset; the Wizard does not install or change Unity packages.



Figure 1. The Characters step assigns a catalog and maximum capacity. Scene UI design remains authored.

CHAPTER 04

Character Catalog and Previews

A **Character Catalog** is your ordered collection of selectable [CharacterDefinition](#) assets. Each definition has a stable **Character ID**, a display name, Gameplay Prefab and optional preview/portrait settings. Catalog order controls browsing, not identity. Begin with [Quick Start](#); configure lobby UI with [Chapter 5 — Lobby Templates and Existing Canvas](#).

Create or choose a catalog

1. Open **Tools > LevelArgento > Couch Co-op Builder > Character Catalog Generator**, or **Generate from Prefabs...** in the Wizard's Characters step.
2. Add your playable prefab assets using **Add Slot** or **Add Selected Prefabs**. Choose an existing **Output Folder** under Assets.
3. Choose **Create a new Character Catalog** and its name, or assign **Existing Catalog** to append eligible definitions.
4. **Review Plan** lists creation/reuse decisions and any blocking problems. **Generate Reviewed Assets** creates the reviewed definitions/catalog without modifying source prefabs or overwriting existing assets. Saved disk assets are not guaranteed to reverse with Undo.
5. Assign the resulting catalog to the Wizard, or select an existing catalog in its **Character Catalog** field. Inspect its definitions and use **Run Catalog Validation** before testing selection.

You can also create Character Catalog/Character Definition assets from Unity's **Create > Couch Co-op Builder** asset menu and assign definitions manually. Keep Character IDs nonempty and unique. Renaming display names or reordering entries need not change IDs; changing an ID breaks existing references to that selection. Reuse the established ID when updating a definition.

Validation and selection state

Use a nonempty catalog with no null/duplicate entries, blank/duplicate IDs or missing Gameplay Prefabs. The built-in lobby selection requires that valid catalog even when your gameplay later binds existing objects or uses a custom provider. The Wizard may allow Apply with a missing-prefab warning; assign those prefabs before expecting lobby confirmation to succeed.

If an ID no longer exists, is ambiguous or its catalog is missing, the gameplay context reports unresolved character metadata instead of inventing a substitute. Correct the definitions/catalog, validate, and select again. API-only gameplay metadata lookup does not require a prefab; selected-character spawning does. See [Gameplay Integration](#) for that distinction.

[SelectedCharacterId](#) belongs to the joined session and persists through lobby/gameplay transitions until permanent Leave. **Ready** belongs to the lobby scene and resets on reconstruction. Preview focus, committed selection and Ready are separate: browsing/moving focus alone does not commit a Character ID or clear Ready. Confirm commits the choice; external committed-selection/catalog changes can invalidate Ready.

Preview host

CharacterPreviewHost is a scene-local runtime presenter. It accepts a CharacterDefinition, creates a presentation-only clone, frames its child renderers, and exposes a small RenderTexture suitable for any lobby view.

Public API

- Show(CharacterDefinition) displays or replaces the active definition.
- RefreshPresentation() recalculates framing without replacing the clone.
- Clear() releases the clone, camera, light, RenderTexture, and runtime layer lease.
- SetOutputSize(width, height) adjusts the bounded 64-1024 pixel output.
- ActiveDefinition, ActiveInstance, OutputTexture, FallbackPortrait, DisplayMode, and StatusMessage expose current presentation state.
- PresentationChanged lets a layout update its prefab, portrait, or placeholder view.

The host does not know about player sessions, input, Ready state, scene loading, or a specific lobby layout.

Prefab and fallback resolution

The host uses CharacterDefinition.SelectionPreviewPrefab. That property resolves an explicit preview prefab first and otherwise returns the gameplay prefab. If neither is available, or if the resolved prefab has no child Renderer, the host exposes the definition's portrait. When no portrait exists, DisplayMode.Placeholder preserves the character name and a reason through the consuming view.

MeshRenderer, SkinnedMeshRenderer, SpriteRenderer, and other Renderer subclasses are discovered on all children. An Animator is optional. Its controller's default state plays normally; Preview Settings > Animation State can request an existing state without requiring a humanoid rig or a LevelArgento-specific controller.

Framing

All child renderer bounds are combined, including off-center children. Perspective framing accounts for texture aspect ratio and expands when an animated pose exceeds the original bounds. Per-character settings provide position, rotation, scale, framing padding, and an optional Animator state/layer. These adjustments affect only the runtime clone.

Camera and resource isolation

Each active host owns one disabled camera, directional light, and 256x256 RenderTexture. The camera renders manually only while that host's renderers are temporarily enabled. The host automatically leases an unused, unnamed runtime layer rather than requiring a fixed project layer or changing TagManager settings. Other preview hosts remain disabled during that render, so player previews cannot render each other's instances. The camera never reads or modifies the customer's Main Camera.

The host releases all native and scene objects on Clear, disable, destruction, Leave, and scene teardown. It does not create a camera or texture per catalog entry.

Gameplay prefab fallback limitations

A gameplay prefab can contain scripts, input components, physics, audio, or other match-only behavior. The host instantiates it beneath an inactive runtime root, disables all cloned MonoBehaviours, physics collisions/simulation, and play-on-awake audio before activating the clone, and leaves Animator components available.

Unity invokes a cloned MonoBehaviour's Awake during Instantiate before the host can disable that component. OnEnable, Start, and later callbacks remain suppressed, but gameplay prefabs with unsafe Awake logic are not suitable as direct preview fallbacks. Other component-specific initialization can also have side effects. Assign a dedicated SelectionPreviewPrefab when a gameplay prefab is expensive, depends on gameplay services, or contains behavior that should never exist in a lobby. The source prefab and its assets are never modified.

Lobby integration

The supplied lobbies create one preview host per joined panel. Browsing changes only that host's presentation. Confirming uses CharacterSelectionService to commit SelectedCharacterId and set scene-local Ready. Leave clears the departing panel's preview; returning to the lobby reconstructs it without replacing the session or InputUser. See [Lobby and Existing Canvas](#) for authored view assignments.

CHAPTER 05

Lobby Templates and Existing Canvas

Use **Tools > LevelArgento > Couch Co-op Builder > Setup Wizard** to copy a supplied presentation or connect your own Canvas. Begin with [Quick Start](#); [Chapter 3 — Setup Wizard](#) explains the current steps, outputs and saving. Smart UI Focus & Navigation is included.

Supplied lobby presentations

Create New Lobby offers Individual Panels, Shared Stage and Character Carousel. Each has four authored visual slots; the Wizard disables unused slots for a smaller capacity. They use the same character-selection and shared Start services. Framework capacity is described in [Player scope](#); Existing Canvas supports your own additional roots.

The copied scene/prefab belongs to your game. In Prefab Mode, edit anchors, positions, fonts, sprites, backgrounds and decorative behavior. Keep the functional binding references or assign your replacements. Use Prefab Variants where helpful. Keep your copy outside supplied paths so updates cannot replace your design.

Existing Canvas

Choose **Integrate Existing Canvas** when you have a lobby design or need five to eight authored player panels. This is a supported setup path. Save the lobby scene first and author one complete root for each supported P1–P8 entry; object names and sample hierarchy names are not requirements.

The Wizard's **Player panel** dropdown displays the configured players. Fields marked * are required. Supply these references for each player:

Assignment	Purpose / boundary
Player UI Root	Entire authored panel; one unique root per player on the assigned Canvas
Choices Root	Controls shown after Join; contained by that player's root
Waiting State Root	Your join prompt shown while the seat is empty; contained by that player's root
Character Selector	An authored Button inside Choices Root; one reusable selector browses the catalog
Selector Label	Either Legacy Text or TextMeshPro (TMP) beneath that player's root
Preview Root	Authored preview container inside the player's root
Preview RawImage	Displays the preview texture; inside Preview Root

Assign one authored shared **Start Button** on the Canvas. The Ready count label, focus highlight and per-player Start focus indicators are optional. Indicators can live elsewhere on that same Canvas; each assigned indicator belongs to one player.

Under **Optional authored presentation**, you may assign portrait/accent Images, focused/confirmed/previous/next character labels, identity/device/Ready labels, and Ready / Not Ready state roots. Optional previous/next labels are for custom presentations; the supplied carousel does not need them. Leave unused targets empty. Use one text implementation per role; TMP labels need an authored font.

Suggest unique components in this root only proposes references with exactly one candidate. It does not choose among multiple Buttons or RawImages. Review every suggestion and all required roots before applying.

Input and navigation wiring

The Wizard adds/reuses nonvisual services and one independent input/focus/preview context per configured player. Each has its own MultiplayerEventSystem, InputSystemUIInputModule, SmartFocusManager, SmartInputModeDetector and SmartFocusScope. The authored player root defines the player's UI boundary; the shared Start router is intentionally shared.

Advanced fields allow existing per-player EventSystems and preview hosts. Leave them empty to create missing components. Do not share the same EventSystem or input module between players. Keep each EventSystem with its input module in the saved lobby scene; its playerRoot must match that player's root. The Wizard reuses already-bound contexts on rerun.

[Default lobby controls](#) remain separate from gameplay input. Each player browses/confirm independently. Focused character previews and the committed selected Character ID are separate: browsing or moving UI focus does not itself commit a selection or clear Ready. Confirm commits the selected ID and sets Ready. Scene-local Ready resets when the lobby is reconstructed; intentional Leave clears the session's selection.

There is one shared Start Game action. Every joined session must be Ready and have available input; a reserved disconnected session blocks Start. Empty seats do not count. Start occurs only after an eligible player's submit, not automatically when the last player becomes Ready.

Visual ownership and previews

Your prefab/Canvas owns all visible UI: layout, player roots, Buttons, labels, Images/RawImages, frames and backgrounds. Runtime code updates assigned content, preview textures, functional visibility/interactability and navigation. It creates no replacement visual hierarchy and does not restyle your layout. Surviving sessions keep their root through scene transitions; intentional Leave permits the lowest available Player Index to be reused.

The preview host owns scene-local rendering resources. Preview content prefers a Selection Preview Prefab, then Gameplay Prefab, portrait, and named placeholder. Dedicated presentation prefabs are recommended when gameplay initialization has side effects. See [Character Catalog and Previews](#).

Apply, extend and check

1. Choose Characters and capacity, then connect the saved Canvas and every required player entry.
2. Choose the gameplay destination, optionally configure cameras, and resolve Review errors.
3. **Apply Setup**, then save the lobby and any modified gameplay scene. Existing Canvas scene wiring supports Undo; Project scene-list changes are separate.
4. Join and confirm independently; move focus to shared Start, enter gameplay, and return. Check that selection/identity persist, Ready resets, and authored layout remains yours.
5. To extend later, author more complete roots, increase Maximum Local Players, assign the new entries and Apply again. Existing contexts are reused. Do not duplicate EventSystems manually just to rerun setup.

For API identity, object/input ownership and optional reversible Drop Out, see [Gameplay Integration](#). For optional Shared/Split management, see [Split Screen Cameras](#).

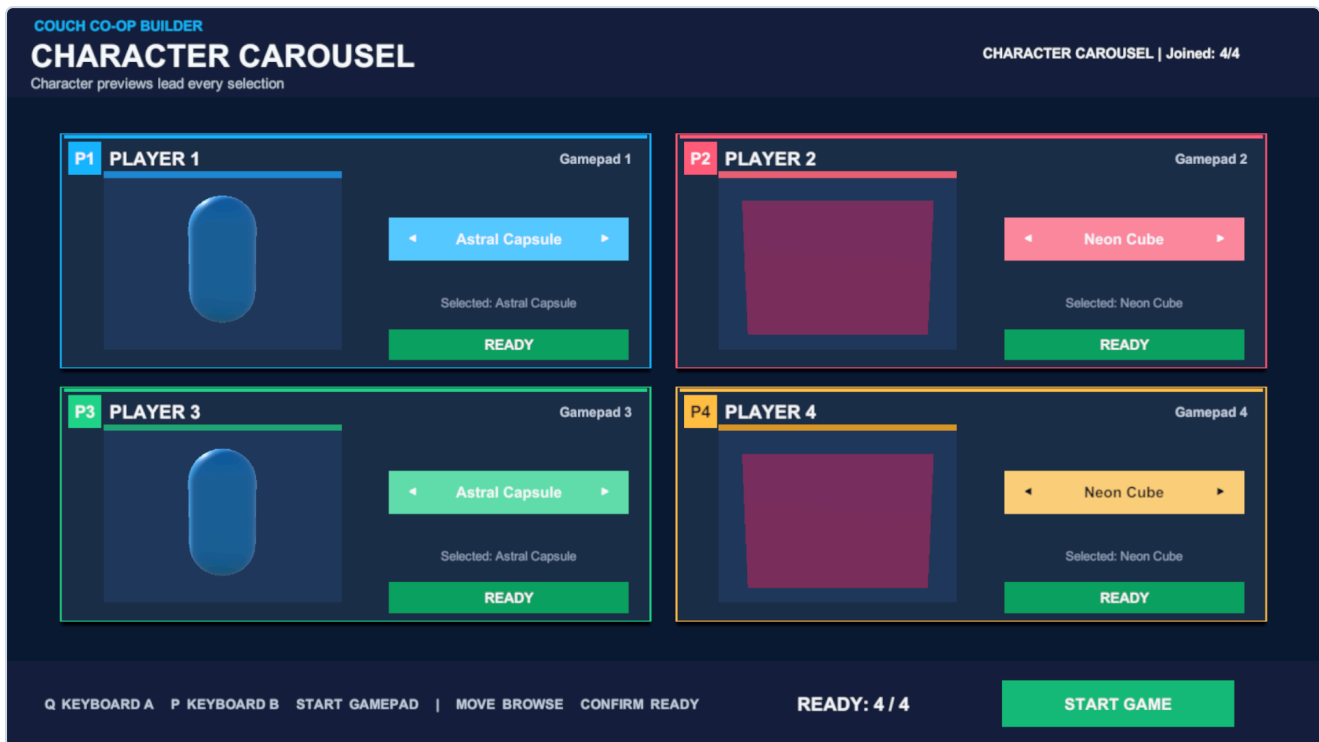


Figure 2. Character Carousel: four authored panels, independent selection and a shared Start control.

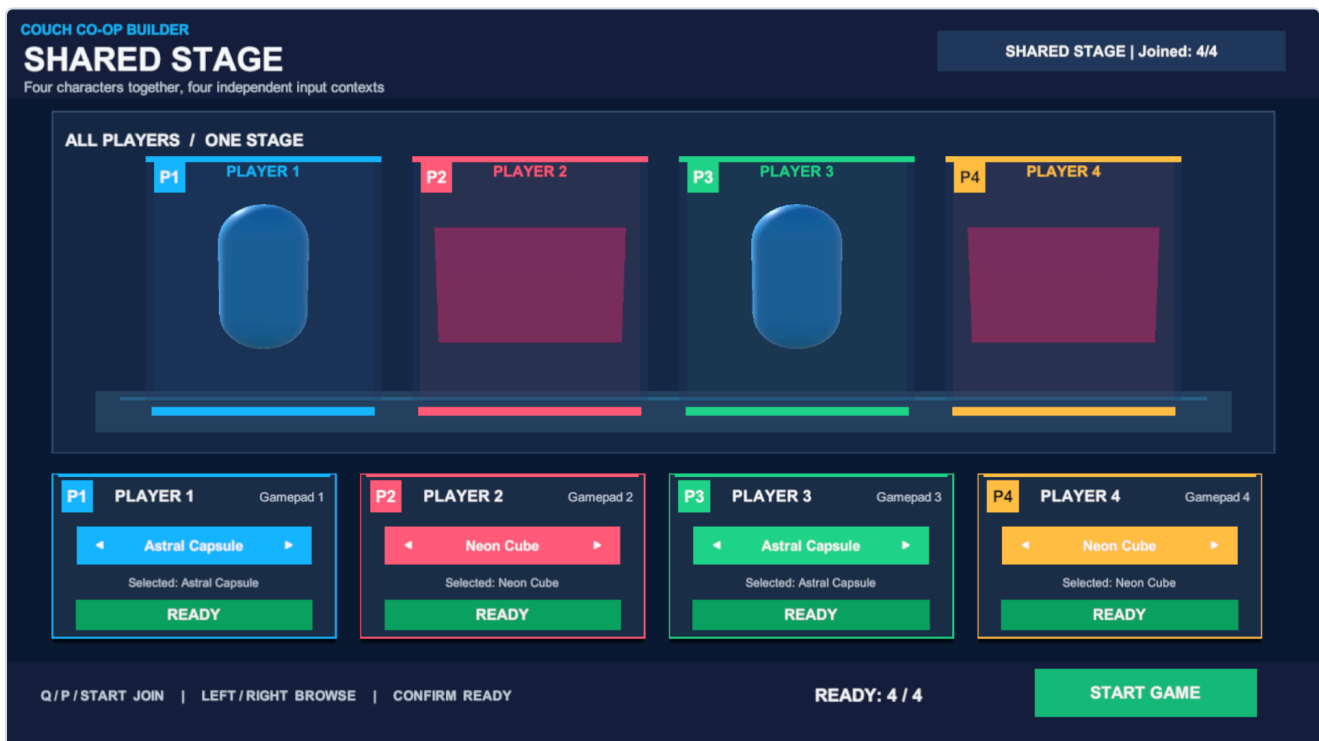


Figure 3. Shared Stage presents four independent player contexts together.

CHAPTER 06

Player Sessions and Identity

Scene handoff and authority

`CouchCoopBootstrap` preserves `MultiplayerCoordinator`, joined `PlayerSession` instances, character-selection IDs and the input services across scene transitions when its lifetime is `PersistAcrossScenes`. Those services retain the actual session InputUsers and actions. `PersistentScenePrototype` remembers the origin lobby and loads a selected gameplay scene; returning reconstructs scene presenters. Ready is scene-local and resets; character selection persists until explicit Leave.

The Setup Wizard selects a gameplay scene and configures the persistent transition. Configure gameplay representations and input separately on that scene's components; the Wizard does not choose a resolution mode, author spawn points, or configure input adapters. Its optional Cameras step works with an existing Gameplay Player Setup. Selection previews instantiate presentation content separately from gameplay.

`GameplayPlayerContext` references the authoritative session. The bootstrap-owned `GameplayPlayerContexts` collection is a read-only projection, not another session store. It retains no scene gameplay objects.

Public access

After initializing your bootstrap, access `bootstrap.GameplayPlayers`. Code without an injected bootstrap can use `CouchCoopBootstrap.Active` explicitly:

```
using LevelArgento.CouchCoopBuilder;

CouchCoopBootstrap bootstrap = CouchCoopBootstrap.Active;
if (bootstrap != null && bootstrap.IsInitialized &&
    bootstrap.GameplayPlayers.TryGetPlayer(0, out GameplayPlayerContext player))
{
    int index = player.PlayerIndex;                // Zero-based, 0 through 7.
    System.Guid sessionId = player.SessionId;      // Stable until Leave.
    string selectedId = player.SelectedCharacterId;
    CharacterDefinition character = player.CharacterDefinition;
    if (player.TryGetInputUser(out var user))
    {
        // Consume existing ownership; do not pair a new user here.
        foreach (var device in user.pairedDevices) { /* Your integration. */ }
    }
}
```

P1–P8 are one-based UI labels for zero-based `PlayerIndex` values 0–7. `InputSlot` identifies the input role/source; it is not a persistent controller or binding-profile identity. Saved input overrides belong to Input Binding Profiles, never a reusable `PlayerIndex`.

Lookup accepts either `PlayerIndex` or session Guid. `Players` is ordered by ascending `PlayerIndex` and may contain gaps after Leave: list position is not necessarily `PlayerIndex`. A returning controller after intentional Leave receives a new session at the lowest available index under the existing policy. Surviving contexts, IDs and indexes do not move. Cache a context for the lifetime of that session, not for an assumed permanent controller player number.

`InputUser` is the existing user, never a new pairing. `PairedDevices` reports current pairing; Keyboard A and B have distinct users but intentionally share the keyboard device. Read `IsJoined` and `IsInputAvailable` separately: a disconnected controller can reserve a joined session while input is unavailable. An ended session exposes an invalid default `InputUser`. Re-read pairing/availability when needed; do not retain a snapshot as proof of connection or use inactivity as disconnection evidence.

`CharacterDefinition` resolves the selected ID against the current configured catalog only if that ID has one match. It does **not** require `GameplayPrefab`. `CharacterStatus` distinguishes no selection, resolved metadata and unresolved metadata. `TryGetCharacterDefinition(out definition, out failureMessage)` explains missing catalogs, missing IDs and ambiguity. Missing metadata does not invalidate session/input ownership or block custom resolution. API-only mode produces no configuration errors for absent objects or character metadata.

Built-in lobby selection requires a fully valid catalog, including Gameplay Prefabs. Context metadata lookup is independent of that lobby rule. The built-in Spawn Selected Character strategy requires the selected definition to have `GameplayPrefab`; Existing Objects, Custom Provider and API Only do not check that requirement.

Selected Character IDs and session ownership persist across lobby/gameplay scene changes. Ready belongs to the current lobby UI. Intentional Leave ends a session; physical Device Lost keeps its reservation. Advanced recovery and reversible Drop Out rules are explained in [Session Lifecycle](#).

CHAPTER 07

Gameplay Representation and Spawning

[Quick Start](#) covers the initial lobby workflow. This guide explains gameplay context, object ownership, input and session lifecycle for **1–8 players**. Joining/selection do not prescribe how your game represents a player; spawn points and PlayerInput are optional.

Representation mode	Choose it when
Spawn Selected Character	The confirmed Character Definition supplies a Gameplay Prefab that Couch should instantiate and clean up.
Bind Existing Objects	Characters, vehicles or other representations already exist in the scene; bind each authored object by Player Index.
Custom Provider	Your pool, team/grid or creation system chooses and owns a live object for each context.
Handle It Myself / API Only	Your architecture consumes the public contexts and manages its own representations. No object setup is required.

Resolution chooses the object; input configuration separately chooses how it consumes owned controls. Your game supplies movement, physics, scoring and match rules. [Shared Camera and Split Screen](#) is an optional module for assignment/viewports; your game supplies camera follow/framing. No Couch base class or prefab inheritance is required.

Scene setup

Add **Couch Co-op Builder > Gameplay Player Setup** to a scene object. Set Mode in the Inspector. Bootstrap may be explicitly assigned; otherwise the component uses `CouchCoopBootstrap.Active`. Initialize the bootstrap before enabling the setup (automatic execution order accommodates a normally initialized persistent bootstrap).

For **Handle It Myself**, the component creates and binds no gameplay objects and requires no object array/provider. The component itself is optional when your game only uses the public context API.

For **Bind Existing Objects**, assign an object at each required zero-based array index. Index 0 belongs to P1, index 7 to P8, independent of object names or hierarchy. Objects must belong to the setup's scene. The framework adds/reuses `GameplayPlayerBinding` and assigns its `Context`; it preserves transforms, parents and active state. Unused authored objects remain unassigned with their authored active state, including objects authored inactive. It does not activate/deactivate, instantiate or destroy them. Your game controls visibility and gameplay lifetime.

For **Custom Provider**, assign a MonoBehaviour implementing `IGameplayPlayerProvider`. Runtime code can alternatively call `Configure(mode, authoredObjects, provider, bootstrap)` with a plain interface implementation. This is a small example; replace the indexed decision with your own pool, vehicle or team logic:

```

using LevelArgento.CouchCoopBuilder;
using UnityEngine;

public sealed class MyGameplayProvider : MonoBehaviour, IGameplayPlayerProvider
{
    [SerializeField] private GameObject[] authoredVehicles;

    public bool TryResolve(GameplayPlayerContext context,
        out GameObject playerObject, out string failureMessage)
    {
        playerObject = context.PlayerIndex < authoredVehicles.Length
            ? authoredVehicles[context.PlayerIndex] : null;
        failureMessage = playerObject != null ? null : "No vehicle assigned to this player index.";
        return playerObject != null;
    }
}

```

The provider receives the persistent context including character metadata and input ownership. It returns one live scene `GameObject` (never a prefab asset), or returns false with an explanation. Resolution attempts occur when sessions enter the enabled setup; repeated `Initialize()` does not retry failures or repeat successful calls. Rejoin after reversible Drop Out requests a fresh association for the same session. Disable/re-enable or call `Configure` to begin a new resolution cycle. Explicit lifecycle owners can call `Release()` and then `Initialize()`. Never assume the framework will retry or instantiate a fallback. Two active contexts cannot bind the same object. The provider owns creation and reclamation; subscribe to `PlayerReleased` if it needs to return objects to a pool.

Read `GameplayPlayerBinding.Context` on a resolved object, or use `setup.TryGetPlayerObject(playerIndex, out object)` from a scene system. Consumers may subscribe to `ContextChanged`, where null means released. Composition is optional for API-only integrations and requires no base-class inheritance or `PlayerInput`. Character changes update the live context and do not automatically replace the resolved gameplay object.

Notifications and lifetime

- `GameplayPlayers.PlayerAdded` / `PlayerRemoved`: joined/left context. Subscribe, then replay `Players` for existing sessions.
- `GameplayPlayers.SelectionChanged`: selection/catalog notification. Built-in spawning uses the first selection of a waiting live join; existing representations are not replaced.
- `GameplayPlayerSetup.PlayerResolved` / `PlayerReleased`: scene object association/release. Subscribe before `Initialize` if you need initial replay notifications.
- `GameplayPlayerSetup.ResolutionFailed`: failure for that context; also logged with player index and session ID.

Unsubscribe scene-owned consumers in `OnDisable`. The setup unsubscribes before teardown and clears its scene references on disable/destroy; bindings clear their Context without reconstructing UI. Destroying a bound object releases the association immediately and reports a useful error on the next surviving setup frame. Deferring that diagnostic avoids false errors during ordinary scene unloading. The bootstrap stores no resolved objects, so scene reload gets fresh bindings and one provider call per joined session per visit.

Existing live Join creates a context and resolves it if a scene setup is enabled. Leave removes its context and releases its binding. Built-in owned instances are destroyed; customer-owned Existing/Custom objects are preserved. An explicit rejoin creates a new session and resolves the available index. Physical input loss keeps existing reservation semantics and does not rerun a provider. These are association semantics only: gameplay

scoring, match participation, respawn and pool reclamation remain customer responsibilities. The optional [reversible Drop Out policy](#) preserves the session while releasing scene bindings. See [Gameplay Input Integration](#) for optional input adapters.

Missing indexed bindings, missing/destroyed providers, false/null provider results, provider exceptions and conflicting object ownership report isolated failures. Other players continue resolving; there is no fallback strategy. Use character-resolution status/message for optional metadata failures. No scene-wide searches or per-frame hierarchy scans are used. Destruction detection checks at most eight cached object/binding references per setup frame, covering never-active objects whose Unity OnDestroy callback may be skipped; diagnostics process a bounded pending queue.

Spawn Selected Character

Spawn Points and PlayerInput are optional. This gameplay representation mode requires GameplayPrefab; Existing Objects, Custom Provider and API Only do not. The separate built-in lobby selection rule above still requires valid prefabs.

Choose Spawn Selected Character on GameplayPlayerSetup, then one placement strategy. Runtime owners can call `ConfigureSpawning(placement, points, origin, offset, placementProvider, bootstrap)`.

1. **Explicit Spawn Points:** assign zero-based indexed Transforms. PlayerIndex 5 uses element 5 even when other sessions have left. World position/rotation come from that Transform; its scale does not affect the prefab. Missing/destroyed required points fail only that index; there is no reuse or world-zero fallback.
2. **Origin + Offset (Single Origin And Offset):** assign one Origin Transform and Per Player Offset. Position is `origin.position + origin.rotation * (offset * PlayerIndex)`; rotation is `origin.rotation`. Offset is expressed along the origin's local axes in world units; origin scale is ignored. With an unrotated origin at (10,1,5) and offset (2,0,0), P1 is (10,1,5), P2 is (12,1,5), P3 is (14,1,5). Rotating the origin rotates the formation. XY offsets support 2D, XZ offsets support 3D. An origin is required; there is no implicit world-zero placement.
3. **Custom Placement Provider:** assign a MonoBehaviour implementing `IGameplaySpawnPlacementProvider`. Runtime configuration also accepts a plain interface implementation. The provider receives the real context and returns a world Pose; it never needs to create the character or implement `IGameplayPlayerProvider`. Each attempted spawn calls it once. A false result/exception/invalid pose affects only that player.

```

using LevelArgento.CouchCoopBuilder;
using UnityEngine;

public sealed class MyPlacement : MonoBehaviour, IGameplaySpawnPlacementProvider
{
    [SerializeField] private Transform gridOrigin;
    public bool TryGetSpawnPose(GameplayPlayerContext context,
        out Pose pose, out string failureMessage)
    {
        pose = default;
        failureMessage = null;
        if (gridOrigin == null)
        {
            failureMessage = "Assign the grid origin.";
            return false;
        }
        Vector3 offset = new Vector3(context.PlayerIndex * 3, 0, 0);
        pose = new Pose(gridOrigin.position + gridOrigin.rotation * offset,
            gridOrigin.rotation);
        return true;
    }
}

```

The selected ID resolves through `GameplayPlayerContext` against the current catalog. Only the definition's `GameplayPrefab` is instantiated. Missing/ambiguous character IDs or a missing prefab fail with the existing `ResolutionFailed` event and diagnostic. `SelectionPreviewPrefab` and portrait are never gameplay fallbacks. The source prefab is never changed. Root scale and active state remain prefab-authored; placement supplies world position/rotation. A compatible `GameplayPlayerBinding` is reused, otherwise added to the runtime clone only. Clones are moved into the setup's scene even when another additive scene is active.

Ownership and live sessions

Built-in clones are framework-owned. Release, component deactivation, Leave and scene teardown unbind and destroy only those clones. `PlayerReleased` fires after unbinding and before destruction is scheduled; normal Play Mode Destroy completes at frame end. A disabled setup removes its built-in representations; re-enabling starts a fresh cycle. Existing and Custom Provider objects remain customer-owned and are never destroyed by this cleanup.

No additional `InputUser` is created, no pairing is performed and no action maps are switched. Physical Device Lost keeps the reserved session and its existing object; the live context reports input unavailability. Reconnection does not duplicate or respawn it. An intentional middle-player Leave destroys only that player's built-in instance and leaves survivors' indexes and transforms unchanged.

A live Join normally has no character selection: the built-in mode waits quietly for its first selection notification. That selection attempts one spawn. Existing joined sessions replayed at scene entry require a selection and report missing metadata as configuration failure. Failed attempts are not automatically retried; selection/catalog notifications do not repeat placement calls for an attempted context. To retry after correcting configuration, disable/re-enable, call `ConfigureSpawning`, or Release then Initialize. A character change after successful spawning updates the context but does not replace its representation; your game must implement respawn or mid-match character switching explicitly. Unexpected runtime destruction is released and diagnosed, with no automatic respawn.

The persistent bootstrap never stores clones. Resolution is driven by scene initialization, Join and the first selection of a waiting Join. Update checks only up to eight cached associations for destruction; it performs no catalog resolution, instantiation or global scene search.

Customer initialization timing

Unity Instantiate can invoke prefab **Awake and OnEnable before Context assignment**. Do not assume context is available in either callback. Use the existing `GameplayPlayerBinding.ContextChanged` notification on a prefab that already contains that component, or `GameplayPlayerSetup.PlayerResolved`. `ContextChanged` guarantees the binding's Context; `PlayerResolved` additionally means the setup lookup has been registered. Subscribe before enabling/initializing setup if you need all initial resolutions. A late consumer can replay contexts and `TryGetPlayerObject` instead.

For a prefab with an existing binding, Awake may subscribe to `ContextChanged`; unsubscribe in `OnDestroy` using Unity's `binding != null` check. A prefab without a binding should use a scene-level `PlayerResolved` consumer because the binding is added after `Instantiate`. Use these notifications rather than relying on prefab initialization order. Use the optional gameplay input strategies below for action ownership; movement remains customer-owned.

CHAPTER 08

Gameplay Input Integration

Lobby input and controller profiles

Lobby Join/Leave/Navigate/Confirm belong to the shared input services and scene UI presenters; configuring a gameplay action asset does not rebind them. [Default lobby controls](#) provide two logical keyboard roles plus recognized gamepads. Additional hardware needs a matching profile and usable controls, not merely a device visible in Input Debugger.

For a generic joystick or supported HID exposed through a Joystick-derived layout, create a `GenericControllerProfile` and assign it to **Shared Joystick Input** → **Profiles**. Add **Couch Co-op Builder > Joystick Control Discovery** temporarily to a scene object, enable **Diagnostics Enabled**, enter Play Mode and operate the device to inspect reusable control paths in Console, then configure Join, Leave, Confirm, Cancel and navigation. Keep a specific base layout/interface and exact hardware metadata filters. Reported HID VID/PID select the controller type; manufacturer/product provide the configured fallback when required HID IDs are unavailable. A reported HID mismatch is rejected rather than bypassed by text matching. Avoid a broad layout-only profile unless that scope is intentional.

One reusable profile can match identical controllers. Runtime device/display names (including appended numbers) do not identify physical hardware. The most specific matching profile is selected; ties are ambiguous and need corrected configuration or an explicit runtime profile assignment. Each joined device still gets independent actions, InputUser and session. Type/profile matching is not proof of unique physical identity across launches; saved overrides use [Input Binding Profiles](#).

Gameplay input binding

Resolution chooses **which object represents the player**. Input mode chooses **how that object consumes the authoritative session's input**. Configure them separately:

Input strategy is independent of representation mode. A resolved object can use Built-in Actions, PlayerInput Adapter or Custom Adapter when configured for that strategy. API Only can consume the context directly without a representation binding.

Configure **GameplayPlayerBinding** on the prefab or existing object. Its default is **Handle Input Myself**, so gameplay input evaluation remains opt-in. Runtime owners can call `binding.ConfigureInput(mode, settings, existingPlayerInput, adapter)` after resolution. Changing configuration explicitly releases the old input cycle. Inspector changes apply on the next binding cycle.

Four input strategies

1. **Handle Input Myself:** no gameplay actions, PlayerInput creation or gameplay method calls. Use `Context.TryGetInputUser` and read `PairedDevices`, `Slot`, `PlayerIndex` and `SessionId` directly. API-only resolution requires no binding at all. No missing-settings error is emitted.
2. **Built-in Actions:** assign `GameplayInputSettings`, which references a customer-owned `InputActionAsset` and map names for Keyboard A, Keyboard B, Gamepad and Joystick. The selected map provides Move/Look (Vector2) and PrimaryAction/SecondaryAction/Jump/Pause (Button). Those names/types are the built-in adapter contract; use Custom Adapter for another contract. Each player gets a clone, including

runtime binding overrides; only its selected map is enabled. Access `binding.Input` and its actions or `ActionTriggered`. No movement, physics or camera behavior is included.

3. **PlayerInput Adapter:** a limited, explicit bridge for an already-authored PlayerInput using native Unity Events. See the limitations below. PlayerInput is never required or automatically added.
4. **Custom Adapter:** assign a MonoBehaviour implementing `IGameplayInputAdapter`, or pass a plain interface instance to `ConfigureInput`. `TryBind` receives the real Context and existing `GameplayPlayerBinding`. `Unbind` is called once when that input cycle ends, including rollback after `TryBind` fails/throws. Make `Unbind` safe after partial initialization. Use one adapter instance per representation; sharing one live instance is rejected. Custom code owns its created resources and must clean them up.

The binding's Context is the only authoritative player association. Input setup follows context assignment.

Awake/OnEnable may precede binding, especially on a spawned prefab. Subscribe to `InputBound` for context plus input readiness; inspect `IsInputReady` / `InputFailure`. `ContextChanged` is emitted after the initial input attempt; it does not guarantee a successful optional adapter. Scene systems can use `GameplayPlayerSetup.PlayerResolved`, then inspect input readiness. Late consumers replay the current binding state. Input failure leaves the valid representation/context in place, logs player/session details and raises `InputBindingFailed`; it does not silently change input strategy or prevent other players from binding.

Session ownership and keyboard roles

The built-in evaluator **does not create an InputUser, pair devices or call AssociateActionsWithUser**. The installed Input System supports one associated action collection per user: replacing it would reset the persistent lobby map's device/mask filtering. Instead, gameplay clones maintain explicit device lists from the existing user's paired devices and subscribe to `InputUser/device` changes. An empty list is kept explicit; it never becomes the null/all-devices fallback. Session availability and pairing govern map enabling. There are no per-frame searches, action cloning or input allocations in this evaluator.

Keyboard A/B intentionally share a physical keyboard and have distinct session users. Device filtering alone cannot separate them. As with the existing lobby architecture, they use **two explicit, disjoint logical action maps**. Both maps are validated against the owned keyboard: resolved controls must be individual keys and must not overlap across roles. Full-keyboard/wildcard/mouse bindings and missing/equal role maps are rejected. Align customer gameplay role maps with the customer's lobby roles; editing a lobby template does not automatically remap the separate gameplay asset. The supplied defaults use the established WASD/F and Arrow/Enter role controls, plus distinct gameplay-only buttons.

Gamepad maps evaluate only their session's paired device(s), independent of `Gamepad.current`, `Gamepad.all` ordering or device display names. Built-in Joystick maps optionally use **Context.JoystickProfile**, the profile actually assigned to that session. Move reuses its directional bindings, processors and effective layout paths. Optional Gameplay controls on `GenericControllerProfile` customize Look/Primary/Secondary/Jump/Pause. Empty Primary/Secondary/Pause use Confirm/Cancel/Join respectively; empty Look/Jump remain unbound. Leave remains the existing profile Leave control. Identity matching and join policy are unchanged. Turn off Use Joystick Profile in `GameplayInputSettings` to use a customer-authored joystick map instead. PlayerInput mode uses its authored action bindings rather than rewriting native action IDs.

Lobby and gameplay have separate action instances and callbacks. Disposable lobby presenters/UI modules unsubscribe on scene exit. Persistent session maps continue managing the existing Join/Leave/reservation behavior; they are not forwarded into gameplay and are not replaced with a new associated collection. A physical key/button can have an intentional meaning in each scene. Avoid assigning gameplay actions to existing Leave controls unless Leave is intended.

PlayerInput bridge: support and limitations

The installed PlayerInput exposes a read-only `user` and automatically creates/pairs its own user in `OnEnable`. There is no public API to adopt the existing session user or set its native `playerIndex`. This bridge **never enables PlayerInput's native lifecycle** and uses no private-field reflection or Input System modifications.

Requirements:

- Author PlayerInput **disabled**, on the same gameplay object as `GameplayPlayerBinding`, before it can be instantiated/activated. Built-in spawning rejects opted-in input prefabs with enabled PlayerInput components before `Instantiate`. An already-enabled component is incompatible; the adapter cannot undo auto-pairing that customer initialization already performed.
- Select **Invoke Unity Events**. Its existing persistent action events are retained and installed on the isolated cloned asset through the public actions getter.
- Assign a source action asset that is not enabled elsewhere; the bridge rejects an enabled source instead of disabling someone else's actions.
- Do not assign a UI Input Module. The bridge owns gameplay action evaluation only.
- Keep the gameplay object active while using the bridge. Deactivation releases its clone; activation begins a fresh input cycle.

The bridge temporarily assigns the cloned asset/current map to the runtime PlayerInput, preserves default map/control scheme/camera/notification/event configuration, and restores the original asset on release. Shared keyboard uses explicit `KeyboardA/KeyboardB` maps, validated as above. With optional `GameplayInputSettings`, its map names select the player's map; otherwise controller roles use `PlayerInput.defaultActionMap`. Only the selected gameplay map is enabled.

Native PlayerInput.user/devices/playerIndex remain invalid/empty/-1. The originating session's `InputUser`, paired devices and `PlayerIndex` remain unchanged and are exposed through `GameplayPlayerBinding.Context`. Native active-player registration, `PlayerInputManager`, automatic control schemes, `SwitchCurrentActionMap` and native device-lost/regained notifications are not supported by this bridge. Use `Context` availability and the owned `Input.Map` instead. Do not enable PlayerInput or call `ActivateInput/SwitchCurrentControlScheme` while bound.

`Send/Broadcast Messages` and native C# `onActionTriggered` are deliberately rejected: the installed PlayerInput gates those paths on its private input-active state. Native Unity action events do not have that gate and can be safely wired without a second user. C# consumers can instead subscribe to `binding.Input.ActionTriggered` or use Built-in/Custom adapters. This is a `UnityEvent` compatibility bridge, not adoption of the canonical user into PlayerInput.

Lifetime, live joins and failures

Input bindings follow representation binding, release, deactivation and scene teardown. Owned action maps are disabled/disposed, all framework callbacks are unsubscribed, native `UnityEvent` hooks are removed and the original PlayerInput asset is restored. Customer objects/source assets are not destroyed. Custom `Unbind` exceptions are diagnosed without stopping other players' cleanup. Do not retain an `Input` instance after release.

A live join binds input once when its representation resolves. The built-in spawner still waits for its first character selection. An intentional `Leave` releases only that player's gameplay input/object; survivors keep indexes/users/instances. `Device Lost` disables unavailable owned actions but retains the same session, object and adapter. The existing services restore ownership on reconnect; device/user notifications update filtering without creating another user, callback set or representation. Your game owns reconnection feedback and pause policy.

Missing settings/assets/maps/actions, wrong built-in value types, keyboard overlap, incompatible PlayerInput configuration and custom adapter exceptions are isolated input failures. Unexpected object destruction also releases input through the existing binding lifecycle and resolution diagnostic. Explicit reconfiguration or deactivation/activation starts a new input cycle; failures are not retried every frame.

Minimal custom adapter

The public factory is also useful when custom code wants the standard isolated evaluator. The returned object is owned by the caller, who must Dispose it. Other custom adapters may consume Context directly instead.

```
using LevelArgento.CouchCoopBuilder;
using UnityEngine;
using UnityEngine.InputSystem;

public sealed class MyGameplayAdapter : MonoBehaviour, IGameplayInputAdapter
{
    [SerializeField] private GameplayInputSettings settings;
    private GameplayActionInput input;

    public bool TryBind(GameplayPlayerContext context,
        GameplayPlayerBinding binding, out string failureMessage)
    {
        input = GameplayActionInput.Create(context, settings);
        input.ActionTriggered += OnAction;
        failureMessage = null;
        return true;
    }

    private void OnAction(InputAction.CallbackContext callback)
    {
        if (callback.performed) { /* Consume this player's action. */ }
    }

    public void Unbind()
    {
        if (input == null) return;
        input.ActionTriggered -= OnAction;
        input.Dispose();
        input = null;
    }
}
```

See [Input Binding Profiles](#) for runtime rebinding and profile-based persistence. Lobby controls, gameplay controls and saved profile identity are separate concepts.

CHAPTER 09

Input Binding Profiles and Rebinding

See [Quick Start](#) for lobby setup and [Gameplay Integration](#) for owned action binding.

This opt-in runtime API belongs to Couch Co-op Builder. A game such as Rocket Rumble can build its settings UI on top of it. The framework does not create a controls menu, change existing Join/Leave listeners, or rebind Smart UI Focus & Navigation. It changes the owned gameplay action clones exposed by [GameplayActionInput](#). Source InputActionAssets and GenericControllerProfile assets are not rewritten.

Identity and shared profiles

PlayerIndex is a reusable session seat displayed as P1–P8; it remains stable until permanent Leave. InputSlot separately identifies the reserved input role/source, and SessionId is stable for that session. None is a persistent binding-profile key. Editing "P3 Controls" means editing [playerBinding.Input.BindingProfile](#), the input source currently assigned to P3. After Leave, the next occupant receives its own source's profile.

Source	Default persistent identity	Consequence
Keyboard A	KeyboardA	Follows the logical A role across seat changes.
Keyboard B	KeyboardB	Follows the logical B role across seat changes.
Standard gamepad	Gamepad:Profile: plus a metadata family fingerprint	Same reported family shares saved overrides. Xbox and DualSense families do not inherit each other's overrides through seat reuse.
Generic joystick	Joystick:Profile: plus the matched GenericControllerProfile identity fingerprint	Uses the actual discovered profile; identical devices share that profile.

Gamepad family metadata includes the Unity layout, interface, manufacturer, product and HID VID/PID when reported. Display names, runtime device names, InputDevice.deviceId and serial numbers are not used as unique identity. A serial field alone is not evidence of cross-launch stability or uniqueness. A transport or layout change can select a different family fallback; this is a default/shared profile, not a claim of physical-device identity.

For generic joysticks, assign [PersistentBindingProfileId](#) once in the profile Inspector, or [ConfigureBindingProfileId](#) during configuration. Reuse that stable customer-owned ID when updating the profile asset. Existing assets with a blank ID use exact base-layout/interface/manufacturer/product/HID match fields. The fallback does not depend on asset/display names or generated binding GUIDs. Changing those match fields changes the fallback key. Use distinct explicit IDs for distinct profiles with otherwise identical match metadata.

Identical controllers cannot be reliably distinguished across launches by this fallback. They have independent actions, sessions and InputUsers, but share binding settings. Editing/resetting that shared profile updates every attached consumer of it. No fuzzy matching or invented physical identity is used.

A platform/customer integration with verified stable, unique physical identity can implement `IInputBindingDeviceIdentityProvider` and supply it to `InputBindingProfileResolver`. Returning true asserts both properties. Do not return a transient deviceId, enumerated display name, player seat, or unverified serial. Verified gamepad keys survive metadata/transport changes; verified joystick keys remain scoped to their controller profile. The framework itself provides no cross-platform guarantee of unique physical identity.

Service and storage

Keep one `InputBindingProfileService` for the game's controls settings lifetime, usually across scene transitions. It owns one complete versioned document in an `IInputBindingProfileStore`. The core model has no dependency on UI or PlayerPrefs.

`JsonFileInputBindingProfileStore` is the simple built-in desktop/mobile backend. Its default file is `Application.persistentDataPath/CouchCoopBuilder/input-bindings.json`. An explicit file path is also supported. It writes a sibling temporary file and replaces the previous document. Customers can implement the two-method store interface using their own save system. Save must replace the whole document atomically, or throw without changing the previous document. Use one writer per store. Platforms without suitable filesystem support, including WebGL/cloud save integrations, should supply their own backend.

Load explicitly and handle a failure before exposing editing/resetting. A failed load retains the last successfully loaded document and live bindings. An empty or missing file means defaults. Unsupported versions, malformed data and duplicate records are rejected. Do not silently overwrite an unknown or unreadable save by continuing with defaults after a failed load.

```
var profiles = new InputBindingProfileService(new JsonFileInputBindingProfileStore());
if (!profiles.TryLoad(out string loadFailure))
{
    // Show/report loadFailure and stop editing until the save is resolved.
    profiles.Dispose();
    return;
}

// After an existing GameplayPlayerBinding has bound its input:
var playerProfile = profiles.Attach(playerBinding.Input);
bool saved = playerProfile.TryRebind(
    "PrimaryAction", 0, "<Gamepad>/buttonNorth", out string failure);
```

Attach after gameplay input is available, for example from the existing `GameplayPlayerBinding.InputBound` callback. Reattach to the new input when the scene representation is recreated. The service automatically detaches when that input is disposed and retains saved overrides. It never creates another `InputUser` or replaces `InputUser.actions`. Custom integrations can attach to their own `GameplayActionInput.Create` result and remain responsible for disposing it.

An input can have one binding profile session. A stable optional `domain` (default `Gameplay`) separates different customer action schemas in the same profile. Keep the domain stable across scenes using the same schema. Only the owned gameplay map is enabled/modified, not every map in the source asset.

Settings before joining

`profiles.OpenProfile(kind, gameplaySettings, device, joystickProfile)` opens a settings preview using the same identity, storage, validation and reset operations. The returned

`InputBindingProfileSession.ActionMap` is an owned, disabled clone for display and editing. It creates no

PlayerSession, InputUser or synthetic device. Dispose the preview when changing profile or leaving settings. Later `Attach(playerBinding.Input)` consumes the saved overrides for that input source.

Both keyboard roles are validated in the preview even when nobody has joined. With no device, a preview can display defaults/saved data and reset, but cannot capture/rebind (`CanRebind` is false). A device-less gamepad preview represents the generic default Gamepad family; it is not a saved Xbox/DualSense physical identity. When a controller is connected, open its actual resolved family profile. Joystick previews require a configured GenericControllerProfile and use its identity, even when its device is absent. Connected joysticks must match that profile. `SharedJoystickInput.ResolveProfile` exposes the same strict discovery scoring and ambiguity result without enabling join listeners.

Capture, paths and conflicts

```
var capture = playerProfile.BeginInteractiveRebind("PrimaryAction", 0);
capture.Finished += result =>
{
    // Inspect Status: Succeeded, Rejected or Canceled, and Failure on rejection.
    // Refresh UI here. Dispose/unsubscribe UI handlers when the menu closes.
};
// The UI's Cancel button or teardown calls capture.Cancel()/Dispose().
```

Only one capture can run per service. The target map is disabled during capture; other players' maps keep working. Capture accepts and suppresses only events from the target's paired devices. Runtime device paths restrict capture but are never saved. Other controllers and another player's Escape key cannot choose/cancel a binding. Cancel is explicit, or occurs at the default ten-second timeout, device loss/disable/unpair, input disposal, session disposal or service disposal. The operation releases native resources and restores the target map when its input is still available. The timeout is configurable. A caller can optionally supply `cancelControl` to `BeginInteractiveRebind`; it must belong to the selected source and uses Unity's existing native cancellation path. It is reserved for cancel during that capture, not saved as an override. Without it, cancellation remains explicit or by timeout.

Programmatic overrides must be a concrete layout-relative path such as `<Keyboard>/h`, `<Gamepad>/buttonNorth` or `<Joystick>/button11`. An empty string unbinds. Runtime instance paths and wildcard/usage paths are rejected. The path must resolve on the source's paired device and fit the action's expected control type. Rebind individual composite parts by binding index, not a composite header. Keyboard sources accept individual keys.

Keyboard A/B share one physical keyboard. The service validates both logical role maps, including the defaults/saved role that has not joined, and rejects cross-role overlap before saving or mutating live actions. Conflicting capture is reported as Rejected with a reason. Reuse by different actions in the same gameplay map is also rejected; multiple bindings of the same action are valid. Identical logical button paths on separate gamepads/joysticks are valid; they are not cross-device conflicts.

The consumer should close or otherwise gate its lobby controls while editing if captured keys also have existing Join/Leave/UI meaning. This foundation does not replace those listeners or implement a modal controls menu.

Reset and action schema updates

- `playerProfile.ResetAction(actionName, out failure)` resets that profile/action in the current domain.
- `playerProfile.ResetProfile(out failure)` implements Reset Player: reset the current source's persistent profile, across its domains and attached consumers.

- `profiles.ResetAll(out failure)` clears all persisted framework profile overrides, including profiles with no currently attached player.
- The service also exposes profile-based `ResetAction/ResetProfile` for settings screens that do not have a live player.

Reset restores the authored defaults captured at attachment, including any customer-authored template overrides, processors and interactions. It does not rewrite source assets. Changes validate using scratch maps and save before live maps change. A failed store save leaves live actions and the document unchanged. Finish/cancel capture before another mutation or reset. `ProfilesChanged` signals successful changes; `PersistedProfileKeys` lists profiles with saved overrides. All service operations are intended for Unity's main thread.

Binding addresses use action name and original structural binding information, not Unity-generated GUIDs (joystick actions are reconstructed with fresh GUIDs). If an action/binding schema changes, unmatched stored addresses are retained and listed in `playerProfile.UnappliedBindings`; they are not applied to a different binding. Plan an explicit customer save migration or reset for schema changes.

Dispose the service at application teardown. Dispose/cancel captures when their UI closes. Detaching a live session restores its owned map's defaults; disposing the underlying `GameplayActionInput` detaches without touching destroyed assets.

Simultaneous action conflicts

Profile validation rejects one physical control assigned to different actions in the selected gameplay map/domain, including composite parts and parent controls. Two bindings of the same action are valid. Both keyboard role maps are compared because they share a Keyboard. Separate controller devices are not compared. Lobby/UI maps and other binding domains are independent and may reuse controls. `InputBindingProfileService.LastConflict` exposes control display/path, both semantic action names and profile keys after a rejected transaction. Rejection leaves live bindings and storage unchanged. Authored defaults must also be valid for their simultaneously active map; resets are validated before saving.

CHAPTER 10

Shared and Split Screen Cameras

[Quick Start](#) covers setup; [Player scope](#) explains capacity. This guide describes optional camera integration.

1. Overview

`GameplayCameraManager` is an optional, scene-local module for shared and individual player cameras, with built-in Split Screen layouts for **1-8 local players**. It manages camera assignment, rendering state and `Camera.rect`.

You own camera behavior. The framework does not prescribe follow, orbit, aim, chase, zoom, collision or multiple-target framing. Your camera prefab or provider can contain your own behavior. The Playground and Rocket Rumble cameras are sample implementations, not universal camera controllers.

Configure gameplay players normally first; see [Chapter 7 — Gameplay Representation and Spawning](#). The manager requires a `GameplayPlayerSetup` that resolves representations. If your game creates its own objects, use that setup's existing Custom Provider mode with an `IGameplayPlayerProvider` returning those objects. `HandleItMyself` does not resolve representations for camera management.

No camera module is required for input, sessions, selection or gameplay integration. Cinemachine is not required.

2. Shared vs Split Screen

- **Split Screen OFF:** your Shared Camera renders the full screen; assigned individual cameras stop rendering.
- **Split Screen ON:** individual cameras render for active resolved participants; the Shared Camera stops rendering. If no individual camera can be resolved, Shared remains a fallback when available.

Changing mode preserves `PlayerSession` identity, `PlayerIndex`, `InputUser`, selected character and gameplay representation. It does not reload scenes or recreate gameplay players. Switching back to Shared requires a Shared Camera reference.

3. Setup Wizard quick start

1. Configure your gameplay scene's `GameplayPlayerSetup` and representation source, then save the scene.
2. Open **Tools > LevelArgento > Couch Co-op Builder > Setup Wizard**. Choose the lobby workflow, catalog, capacity and gameplay scene normally.
3. On the **Cameras** step, check **Enable Shared / Split Screen Camera Management**. It is unchecked by default. Unchecked means no manager is added, no camera fields are required, and existing camera setup is left untouched.
4. Use **Open Gameplay Scene Additively** to load the chosen gameplay scene alongside the lobby. Assign its **Gameplay Player Setup**; the Wizard suggests it only when exactly one candidate exists. Assign your existing **Shared Camera** from that scene.
5. Choose **Camera Prefab**, **Camera On Gameplay Representation**, or **Custom Provider**, with the source described below. Prefab/provider configuration is required even when the initial mode is Shared, so runtime Split switching has a usable source.
6. Choose **Default Camera Mode: Shared / Split**, the two-player layout and the three-player layout. Four through eight use automatic built-in layouts. Advanced options allow an existing custom layout provider

and an AudioListener policy.

- Complete the authored lobby/Canvas bindings. On **Review**, inspect validation and the plan, then **Apply Setup**. The Wizard adds one `GameplayCameraManager` on the assigned setup object if none exists, or reuses the single existing manager in the gameplay scene. It assigns references and defaults, without creating camera objects or providers.
- Save the gameplay scene after Apply**, and save the lobby for Existing Canvas. The gameplay scene is marked dirty, not automatically saved. Enter Play Mode, join players and call `SetSplitScreenEnabled(true)` from your Settings UI.

Existing Canvas lobby and camera edits form one scene Undo operation. New Lobby output assets cannot be deleted by scene Undo; camera wiring still supports Undo. Project scene-list additions are a separate project-configuration effect. New Lobby refuses to overwrite existing output assets and removes partial outputs if creation fails.

Rerunning reads the single existing manager's configuration when you assign the setup object or open its scene through the Wizard. Review those suggested references before Apply. Reapplying reconciles one manager, without duplicate cameras or providers. Multiple managers produce an actionable validation error rather than a guess about which one to replace. Turning the checkbox off leaves any previously configured manager intact.

4. Manual setup

- Add `GameplayCameraManager` to the gameplay scene, normally on the `GameplayPlayerSetup` object.
- Assign **Player Setup**, your **Shared Camera**, and **Player Camera Source** with its prefab/provider where applicable.
- Set **Split Screen Enabled**, **Two Player Layout**, **Three Player Layout**, and optionally the layout provider/audio policy in the Inspector.
- Author camera behavior and test both modes with real joined gameplay participants.

For code-driven setup, use `Configure(GameplayPlayerSetup setup, Camera shared, GameplayCameraSource source, GameObject prefab = null, IGameplayCameraProvider provider = null)`. It releases the previous camera assignment cycle. Settings toggles should use `SetSplitScreenEnabled` instead of repeatedly calling `Configure`.

5. Runtime toggle and layouts

These are independent Settings actions; `cameraManager` is your assigned `GameplayCameraManager` reference. Include `using LevelArgento.CouchCoopBuilder;` in scripts using these types:

```
cameraManager.SetSplitScreenEnabled(true); // Split
cameraManager.SetSplitScreenEnabled(false); // Shared
```

```
cameraManager.SetTwoPlayerLayout(TwoPlayerViewportLayout.Vertical);
cameraManager.SetThreePlayerLayout(ThreePlayerViewportLayout.TopTwoBottomOne);
```

Read state through `SplitScreenEnabled`, `TwoPlayerLayout` and `ThreePlayerLayout`. The Wizard default is serialized scene configuration; your Settings UI can change it at runtime. The module does not persist preferences or generate a mandatory Settings screen.

Public event	Use
<code>SplitScreenModeChanged</code> (<code>Action<bool></code>)	Reflect Shared/Split changes in your Settings UI.
<code>ViewportLayoutChanged</code> (<code>Action</code>)	React after rect/rendering-state changes, including layout and participant updates.
<code>PlayerCameraAssigned</code> (<code>Action<GameplayPlayerContext, Camera, GameObject></code>)	Connect camera behavior or HUD to its current context and representation.

Subscribe/unsubscribe with your component's normal lifecycle. `TryGetPlayerCamera(int playerIndex, out Camera camera)` finds an assigned active camera. Call `Refresh()` after an unavailable custom camera becomes ready to retry assignment.

6. Built-in layouts 1-8

Active views pack in ascending existing `PlayerIndex` order. Packing never rennumbers sessions: **P1 / P4 / P7 remain P1 / P4 / P7**, occupying three visual slots. Unity rect coordinates start at the bottom-left; row names below describe the visual top row first.

Players	Layout
1	Full screen
2	Horizontal (top/bottom) or Vertical (left/right)
3	Top 2 / Bottom 1; Bottom 2 / Top 1; Left 1 / Right 2; Right 1 / Left 2
4	2 + 2
5	3 + 2
6	3 + 3
7	4 + 3
8	4 + 4

Two- and three-player layouts are selectable; four through eight use automatic arrangements. In the automatic 4–8 arrangements, rows share screen height equally and columns fill each row. Rects cover the full screen without decorative gaps. Three-player enum values are `TopTwoBottomOne`, `BottomTwoTopOne`, `LeftOneRightStack`, and `RightOneLeftStack`. Two-player values are `Horizontal` and `Vertical`.

7. Player camera sources

Camera Prefab

Assign a prefab containing **exactly one Camera**, on the root or a child. The manager clones the full prefab once per resolved session when Split first needs it. Follow/zoom scripts on the rig are retained. No particular genre, follow script or Cinemachine package is required.

`GameplayCameraAssignment` is added to the clone root. Its `Context` and `Representation` identify the player and target; `AssignmentChanged` fires when bindings change, including rejoin. Author prefab `AudioListeners` disabled under the default framework policy to avoid transient duplicate listeners during cloning.

Camera On Gameplay Representation

Each resolved representation provides a Camera in its hierarchy. Ownership stays with its owner. The Wizard checks known authored existing objects and selected-character gameplay prefabs for missing Cameras. Dynamic/custom providers are not rejected merely because future cameras cannot be inspected: verify their resolved representations in Play Mode. A representation Camera cannot also be the Shared Camera.

Custom Provider

Assign your gameplay-scene `MonoBehaviour` implementing `IGameplayCameraProvider`. Its existing method is:

```
bool TryResolveCamera(GameplayPlayerContext context, GameObject representation,  
    out Camera camera, out string failureMessage);
```

Return a live scene Camera for that participant, not a prefab asset or a Camera assigned to another participant. The provider owns camera lifetime, behavior and pooling. The Wizard assigns your component; it does not invent an implementation. Multi-camera rigs/advanced stacks need your own setup through this source.

8. Camera ownership

The manager never deletes or reparents your Shared Camera, replaces follow scripts, or edits camera FOV, projection, clipping planes, renderer settings or behavior. At runtime it controls assigned cameras' enabled state and rects; the default audio policy also manages assigned-hierarchy listeners. Prior customer enabled/rect/listener states are restored on release.

Framework-created prefab clones belong to the manager. Customer representation/provider cameras stay customer-owned and are never destroyed by the manager. Disabling/destroying the manager releases its clones and restores customer state. A pooled provider should recycle cameras through its own representation lifecycle; the manager does not own that pool.

9. Drop Out / Rejoin and permanent Leave

Intentional gameplay **Drop Out** releases that participant's representation and removes its active viewport; remaining views repack without renumbering `PlayerIndex`. **Rejoin** restores the participant. Framework-owned camera clones remain disabled/reserved for the session and are reused with the new representation binding.

Representation cameras follow their representation's lifetime. Custom providers manage their returned cameras. **Permanent Leave** performs normal session cleanup and destroys framework-owned clones for that session; it is distinct from reversible Drop Out.

10. DeviceLost

Temporary device loss leaves the participant represented and its camera normally assigned. The manager follows representation lifecycle, not raw device events or inactivity. Your game may pause or show reconnection UI according to its existing session policy. `DeviceLost` does not silently become Leave or Drop Out.

11. AudioListener

Default `ManageAssignedCameraListeners` enables one listener among assigned player rigs: the first active rig with a listener, or Shared as fallback. Shared mode uses the shared listener. Prior listener states are restored on release.

Only listeners in assigned player rigs/representation hierarchies and the shared-camera hierarchy are managed. Other scene listeners remain yours. The Wizard creates no extra listeners and changes no prefab assets. Select **Developer Managed** (`DeveloperManaged`) if your game owns listener switching; `SetAudioPolicy(GameplayCameraAudioPolicy policy)` also changes policy at runtime. Neither policy prevents duplicate listeners you independently enable elsewhere.

12. UI / HUD

- **Screen Space Overlay:** global full-screen UI, appropriate for Pause, Settings and Winner. It is not cropped into player viewports.
- **Screen Space Camera:** an individual HUD can target its assigned Camera through `Canvas.worldCamera`.
- **World Space:** normal Unity behavior through relevant cameras.

Subscribe to `PlayerCameraAssigned` to connect your authored HUD to the camera using the provided context and representation. Rebind when assignments change. **The manager does not automatically clone customer HUDs.** Packed visual positions are not new player identities.

13. URP and Cinemachine

Normal URP Base Cameras support `Camera.rect`. Overlay stacks, renderers, post-processing and renderer features remain customer-controlled. Prefab source expects exactly one Camera; use a custom provider for multi-camera stacks and verify them on your target Unity/URP version.

Cinemachine is **not required**. There is no Cinemachine package dependency or built-in integration package. Developers may use Cinemachine in their own prefabs/providers; its target binding and advanced URP/stack configuration remain theirs.

14. Custom layout provider

In Advanced Camera Options or the Inspector, assign an existing scene component implementing `ICameraViewportLayoutProvider`. Empty uses built-in layouts. At runtime use `SetViewportLayoutProvider(ICameraViewportLayoutProvider provider); null` returns to the built-in provider.

The interface fills normalized rects in ascending active `PlayerIndex` order:

```
bool TryGetViewports(int viewCount, TwoPlayerViewportLayout twoPlayer,
    ThreePlayerViewportLayout threePlayer, Rect[] viewports);
```

The manager validates finite, positive, in-bounds, non-overlapping full-screen tiling and falls back to built-in layouts if custom output is invalid. The supported range is 1-8. The Wizard checks the component/interface reference; it does not execute customer layout code during setup.

15. Performance

One through eight layouts are supported. More cameras increase rendering cost; performance depends on resolution, scene complexity, shadows, post-processing, renderer features and hardware. Eight cameras do not guarantee high-setting performance on every device. Profile your own content/build and tune effects and resolution. No universal game performance is promised.

16. Samples

See the [Samples index](#) for entry points.

Demonstration	Purpose
Rocket Rumble 3D	Real game integration, fixed Shared Camera, individual 3D follow cameras, runtime Camera Settings and Drop Out/Rejoin; up to four players.
Split Screen Playground 3D	Independent 1-8 join/entry flow, perspective follow cameras and 5/6/7/8 layouts.
Split Screen Playground 2D	Independent 1-8 join/entry flow, orthographic cameras, simple platformer movement and Shared/Split switching.

The Playground guide links both entry scenes and explains the independent 1–8-player join flow. Sample follow/framing belongs to the game; the core has no dependency on those scripts.

17. Troubleshooting

Problem	Check
Camera fields missing in Wizard	Enable Shared / Split Screen Camera Management on the Cameras step.
Cannot assign gameplay references	Open the selected saved gameplay scene additively and assign its Gameplay Player Setup. Lobby objects and prefab assets are not gameplay-scene references.
Shared mode has no view	Assign Shared Camera; switching to Shared needs it even if initial mode was Split.
Split has no individual cameras	Resolve representations first and configure a camera source. HandleItMyself does not resolve objects.
Camera Prefab rejected	Assign an asset with exactly one Camera, including inactive children. Missing/multiple Cameras or scene objects are unusable for this source.
Representation lacks a Camera	Add it, choose prefab source, or use a custom provider. Test dynamically created representations after resolution.
Custom provider missing/invalid	Assign a live gameplay-scene component implementing <code>IGameplayCameraProvider</code> or <code>ICameraViewportLayoutProvider</code> , as applicable.

Problem	Check
Multiple managers on rerun	Reconcile them explicitly; the Wizard does not delete one or guess which to configure.
Duplicate AudioListener warning	Disable unrelated scene listeners, author prefab listeners disabled, and verify the selected policy.
HUD fills screen	Overlay is global; bind an individual Screen Space Camera Canvas to its assigned camera.
P numbers seem out of order after Drop Out	Views repack; PlayerIndex stays stable. P1/P4/P7 still identify those sessions.
Camera does not follow	Supply your behavior and bind targets through GameplayCameraAssignment or PlayerCameraAssigned.

`ValidateConfiguration(out string failure)` reports the first basic runtime configuration problem. Detailed resolution diagnostics are reported once per problem; `Refresh()` can retry a provider after its camera becomes available. Fix legitimate diagnostics rather than repeatedly clearing them. No proximity split/merge, animated/diagonal splits, HUD cloning, minimap or spectator system is provided.

CHAPTER 11

Persistent Sessions

This **four-panel** example demonstrates session/input ownership and selected Character IDs across lobby → game → lobby, with scene-local Ready. It is a UI/context example, not a movement, combat or production gameplay scene. See [Player scope](#) for framework capacity and [Chapter 3 — Setup Wizard](#) for setup.

Open and run

Use [Configure Supplied Samples](#) to register Persistent Sessions. Open `Assets/LevelArgento/CouchCoopBuilder/Prototypes/Persistent/PersistentLobbyPrototype.unity`, then enter Play Mode. The sample creates one persistent bootstrap only when none exists. Join using [default lobby controls](#); navigate each player's panel independently.

The lobby includes its own Persistent Sample Catalog (`Assets/LevelArgento/CouchCoopBuilder/Prototypes/Persistent/PersistentSampleCatalog.asset`), which reuses the shipped Astral Capsule and Neon Cube character definitions and preview prefabs. The character selector requires a valid catalog; replace the sample catalog on the lobby's `PersistentScenePrototype` with your own to browse different characters. Confirm each joined player, then focus **START GAME** and submit; the lobby requires all joined players Ready with available input. The game shows **RETURN TO LOBBY**, without a Ready label. Confirm it to return.

What persists

Session ID, zero-based Player Index, input role/slot, InputUser and selected Character ID persist. Each session keeps its corresponding panel across scene transitions; gaps do not compact. Intentional Leave ends that session and frees its seat/slot. The next Join can use the lowest available seat; survivors retain theirs. Physical Device Lost keeps the existing reservation.

Ready is scene-local and resets when the lobby is reconstructed. Navigation changes preview focus only; confirmation commits selection and Ready. Returning reconstructs scene-local UI/previews while keeping authoritative sessions and users. The sample supplies a minimal camera when no enabled scene camera exists.

Concepts to study

Read `CouchCoopBootstrap`, `PersistentScenePrototype` and the independent player presenters. A panel's [Character Preview Host](#) renders presentation content separately from gameplay. [Gameplay Integration](#) explains context APIs, representation ownership and advanced recovery.

Try two keyboard roles, then supported controllers: join/confirm independently, transition twice, Leave/rejoin a middle seat and compare surviving session/panel mappings. Test actual device loss separately from intentional Leave. Optional **Report Persistent Session Diagnostics** on the scene component prints ownership details when needed; verbose logging is off by default.

CHAPTER 12

Device Lost, Drop Out and Rejoin

Reserved controller replacement during a session

Same-instance `InputUser.DeviceLost` / `DeviceRegained` preserves the user, session, context, binding and representation. Gamepad/joystick services follow the regained owned device, including a replacement instance with the same runtime ID.

For a genuinely new device ID, Unity does not generate `DeviceRegained` by matching hardware descriptions. The framework can reclaim the existing user's reserved seat:

- Automatic reclaim requires one unavailable owner with the exact same nonempty product, layout and full device description, including a nonempty serial unique among the known controller owners.
- Without a serial, pressing the normal Join action can explicitly reclaim exactly one matching reserved seat. This is seat-reclaim intent, not proof of physical identity.
- The incoming device must have appeared after loss; controllers already registered when the loss occurred keep their ordinary Join semantics.
- Multiple matching reserved seats are ambiguous: reclaim and duplicate Join are rejected. Reconnect the original instances, or intentionally end those sessions.
- Differing or missing hardware descriptions are not guessed. Display-name suffixes and `deviceId` are never persistent identities. No cross-launch identity is claimed.

Reclaim clears only the existing user's obsolete lost pairing and pairs its replacement through `InputUser`. It does not create another session, user or gameplay object. `Device Disabled/Enabled` also updates controller availability; both shared keyboard roles report availability from their existing user's live paired keyboard.

Optional gameplay Drop Out and Rejoin

`MultiplayerCoordinator.TryLeave(sessionId)` remains a permanent Leave: it removes the session, releases its input slot, `InputUser` and scene binding, and allows the lowest available `PlayerIndex` to be reused. A gameplay scene can opt into reversible owned Leave actions by setting `coordinator.DropOutOnLeave = true` for that scene's lifetime and restoring `false` on teardown. The shared keyboard, gamepad and joystick services then route their owned Leave callbacks through `TryHandleLeaveRequest`.

Drop Out keeps the `PlayerSession`, `InputSlot`, `PlayerIndex`, selected character and input role/profile reservation. `PlayerSession.IsDroppedOut` and `GameplayPlayerContext.IsDroppedOut` distinguish it from `DeviceLost`; `IsInputAvailable` returns false for dropped-out gameplay. `PlayerDroppedOut` / `PlayerRejoined` events also reach `GameplayPlayerContexts`. `GameplayPlayerSetup` releases its scene binding on Drop Out and resolves the selected character again on rejoin. It does not choose a sample respawn point or preserve a game's score; those are scene-owned decisions. The matching input service accepts the normal Join control for the exact still-owned keyboard role or connected controller instance and calls `TryRejoin` without creating a new `PlayerSession` or `InputUser`. Device replacement still follows the conservative metadata-based reclaim policy above; ambiguous hardware is not guessed. An automatic `DeviceRegained` does not itself reactivate a voluntarily dropped-out participant.

`TryDropOut` and `TryRejoin` are explicit APIs for integrations that have validated the relevant input source. An explicit `TryLeave` remains permanent even while the opt-in owned-action policy is enabled. Main Menu shutdown uses permanent Leave. The opt-in flag is scene policy; customers should set it only where their gameplay represents dropped-out players and reset it when that scene ends.

CHAPTER 13

Configure Supplied Samples

ProjectSettings are not shipped with the asset. Multi-scene samples load their Lobby, Arena or return scene by name, so opening the first scene alone does not register its destinations for your project.

Preview and explicitly apply

1. In Edit Mode, open **Tools > LevelArgento > Couch Co-op Builder > Configure Supplied Samples**.
2. Select Rocket Rumble 2D, Rocket Rumble 3D, Persistent Sessions, or **Select All Samples**.
3. Review the target shown at the top and the **Scene list after configuration** preview.
4. Click **Configure Selected Samples**. No project configuration changes happen on import or merely by opening the window.
5. Open the selected sample's first scene below and enter Play Mode.

Sample	Scenes, in configured order
Rocket Rumble 2D	RocketRumble2DMainMenu, RocketRumble2DLobby, RocketRumble2DArena
Rocket Rumble 3D	RocketRumble3DMainMenu, RocketRumble3DLobby, RocketRumble3DArena
Persistent Sessions	PersistentLobbyPrototype, PersistentGamePrototype

The first scene in each row is its entry scene. Rocket Rumble scene assets are under [Samples/RocketRumble2D/Scenes/](#) and [Samples/RocketRumble3D/Scenes/](#); Persistent assets are under [Prototypes/Persistent/](#), all relative to [Assets/LevelArgento/CouchCoopBuilder/](#). The [Sample Index](#) gives complete entry paths.

Split Screen Playgrounds have an independent join/gameplay flow within their own scenes. They do not need this multi-scene configuration simply to enter Play Mode.

Unity 6 Build Profiles and shared scenes

When the active Build Profile has **Override Global Scenes** enabled, the tool updates that profile's scene list. Otherwise it updates [EditorBuildSettings.globalScenes](#), the shared scene list used by profiles without overrides. The target label states which list will change. Other profiles are not edited.

Only the selected samples' scenes are enabled, deduplicated and grouped in the order shown. Their block replaces the first selected-scene occurrence, or is appended if none was present. Unrelated scenes keep their relative order, enabled state and GUID. Existing sample scenes and product assets are not modified.

Running the same selection again produces the same scene list. It is safe to rerun after reviewing the target. Missing scene assets are reported and prevent applying; configuration is unavailable during Play Mode.

Reverting

Profile changes support Unity Undo. **Revert Last Configuration** can restore either target list while the window remains open and that same target list still matches the applied configuration. If another edit changes the list, Revert is disabled rather than overwriting that edit. Shared scene-list changes use this Revert action rather than Unity scene Undo. Review Build Profiles separately from scene-file saves.

CHAPTER 14

Sample Index and Integration Examples

Sample index

Before entering Play Mode, use [Configure Supplied Samples](#) for Rocket Rumble and Persistent Sessions. Split Screen Playgrounds run independently. Customize customer-owned copies rather than supplied assets.

Sample / capacity	Open	Study
Rocket Rumble 2D , 2–4	Main Menu (<code>Assets/LevelArgento/CouchCoopBuilder/Samples/RocketRumble2D/Scenes/RocketRumble2DMainMenu.unity</code>)	Platform combat, owned actions, settings, reversible Drop Out/Rejoin; fixed Shared Camera
Rocket Rumble 3D , 2–4	Main Menu (<code>Assets/LevelArgento/CouchCoopBuilder/Samples/RocketRumble3D/Scenes/RocketRumble3DMainMenu.unity</code>)	XZ combat, the same profile/settings flow, optional individual views and runtime Camera Settings
Split Screen Playgrounds , 1–8	3D (<code>Assets/LevelArgento/CouchCoopBuilder/Samples/SplitScreenPlaygrounds/Scenes/SplitScreenPlayground3D.unity</code>) / 2D (<code>Assets/LevelArgento/CouchCoopBuilder/Samples/SplitScreenPlaygrounds/Scenes/SplitScreenPlayground2D.unity</code>)	Direct join/entry, camera providers/follow examples, selectable 2P/3P and automatic 4–8 layouts
Supplied lobby templates , up to 4 visual slots	Copy a style with the Wizard	Authored panels, shared stage or carousel, previews, independent focus and one shared Start
Existing Canvas example, 2 authored roots	ExistingCanvasLobby (<code>Assets/LevelArgento/CouchCoopBuilder/Samples/ExistingCanvas/ExistingCanvasLobby.unity</code>)	Explicit UI references and reuse; author more roots for up to eight
Persistent sessions example , 4	PersistentLobbyPrototype (<code>Assets/LevelArgento/CouchCoopBuilder/Prototypes/Persistent/PersistentLobbyPrototype.unity</code>)	Identity/selection across lobby/game return; scene-local Ready
Included Smart UI samples	Quick Start, Focus Management, Navigation Showcase	Focus recovery, scopes, modals and navigation; no Couch player joining

The gameplay context examples can create an eight-capacity input bootstrap when opened standalone; standalone joining does not select a character automatically. To demonstrate selected-character spawning, enter through a configured lobby with confirmed selections. Use the sample host's **Return to Origin Lobby (Play Mode)** context action to return. None of these diagnostic examples supplies general movement or game rules.

Related guides: [Character Catalog and Previews](#), [Gameplay Integration](#), [Input Binding Profiles](#), [Split Screen Cameras](#), [Included Smart UI Focus & Navigation](#).

Gameplay examples

The [sample index](#) links every entry scene. Context examples support eight sessions and show ownership in the Inspector; they have no movement or match rules:

Example	What to study
API Only (Assets/LevelArgento/CouchCoopBuilder/Samples/Gameplay/GameplayApiOnly.unity)	Public contexts without representation bindings
Existing Objects (Assets/LevelArgento/CouchCoopBuilder/Samples/Gameplay/GameplayExistingObjects.unity)	Eight authored indexed objects; the inactive P8 object stays inactive
Custom Provider (Assets/LevelArgento/CouchCoopBuilder/Samples/Gameplay/GameplayCustomProvider.unity)	Indexed provider, one resolution per session per enabled cycle
Explicit Spawn Points (Assets/LevelArgento/CouchCoopBuilder/Samples/Gameplay/GameplaySpawnPoints.unity)	Eight indexed poses, selected Gameplay Prefabs and framework clone ownership
Origin + Offset (Assets/LevelArgento/CouchCoopBuilder/Samples/Gameplay/GameplaySpawnOriginOffset.unity)	Rotated formation and owned-input diagnostics on resolved clones
Custom Placement (Assets/LevelArgento/CouchCoopBuilder/Samples/Gameplay/GameplaySpawnCustomPlacement.unity)	A placement provider returning poses without creating objects

For selected-character handoff, use your own saved lobby, assign a valid catalog, target the chosen scene on PersistentScenePrototype and enable both scenes in the scene list used by the active Build Profile. Join/confirm, Start Game, inspect the context/binding, then use **Return to Origin Lobby (Play Mode)** on the sample host. Selection/session identity persists; Ready resets. The scenes can create an eight-capacity bootstrap when opened standalone, but Join alone does not select a character; spawning needs a confirmed selection.

Origin + Offset demonstrates the generic gameplay defaults: A WASD/IJKL/F/R/Space/T; B arrows/numpad look/Enter/Right Shift/numpad 0/Y; gamepads left stick or D-pad/right stick/South/East/West/Start, for Move/Look/Primary/Secondary/Jump/Pause respectively. Generic joysticks use their configured profile. These defaults differ from Rocket Rumble's sample controls. PlayerInputSampleCapsule is an authored example of the disabled UnityEvent bridge described above; it is not required for other strategies.

When adapting a sample, check independent input, Leave/rejoin of a middle seat, device loss and repeated lobby returns. Surviving indexes/users stay stable; each new scene gets fresh scene-owned bindings. See [Chapter 20 — Troubleshooting](#) and the isolated ResolutionFailed/InputBindingFailed diagnostic when configuration is missing.

CHAPTER 15

Rocket Rumble Menus, Settings and Lifecycle

This guide covers the shared behavior of [Rocket Rumble 2D](#) and [Rocket Rumble 3D](#), optional **2–4-player** sample games. Keep this folder when using either sample. Begin with [Quick Start](#) for framework setup; these menus and match rules are sample content.

Main Menu and lobby

Open the chosen sample's Main Menu scene and enter Play Mode. **PLAY** opens its lobby; **SETTINGS** contains Audio and Controls; **QUIT** exits a standalone build and leaves Editor Play Mode running. Main Menu creates no joined players.

Join with Q, P, gamepad Start or a matched joystick's Join control. Browse and confirm Cobalt, Ember, Mint or Orchid, then move focus down to **START GAME**. At least two joined players must be Ready with available input. Moving focus does not clear Ready. The [default lobby controls](#) are independent of gameplay bindings.

Menu panels support mouse, keyboard, gamepad and configured joystick navigation. Keyboard A uses WASD/F, Keyboard B arrows/Enter; gamepads use stick/D-pad/South; the supplied DragonRise profile uses stick/button2. Visible Back controls and keyboard G/Backspace/Escape or controller/profile Cancel return to the preceding panel. Smart UI Focus & Navigation manages selection recovery, scopes and modals. Release held controls before the next fresh menu press.

Audio settings

Master, Music, SFX and UI sliders apply immediately to the included sample mixer. Zero mutes; 100% is 0 dB. Defaults are Master 100%, Music about 25%, SFX about 56% and UI about 45%. **RESET AUDIO DEFAULTS** restores and saves them. Sample audio preferences use PlayerPrefs and survive scene changes/restarts; they are separate from input profiles.

One persistent sample audio service carries lobby music through menu/settings/lobby. Countdown has no arena music; GO starts arena music; victory stops it; lobby return restores lobby music. Jump, death, respawn, rocket launch, countdown, winner and UI feedback use included sample audio. The service releases itself when leaving Rocket Rumble scenes. This is sample audio integration, not a required framework audio system.

Controls settings

Choose Keyboard A, Keyboard B, a connected gamepad family or a configured Joystick profile. The device-less gamepad placeholder can show/reset defaults but **CHANGE** needs connected hardware. Gameplay rows cover Move, Jump, Fire, Shield and Pause / Winner Return. Lobby Confirm/Back/Leave are fixed, separate controls; Join is not offered for rebinding.

Bindings belong to Input Binding Profiles, never P1–P4 seats. KeyboardA/KeyboardB follow their logical roles; controller settings follow the resolved family/profile. Identical controllers without verified unique identity share settings, while keeping separate actions, sessions and InputUsers. Seat reuse cannot transfer another source's profile. See [Input Binding Profiles](#) for the storage abstraction and honest identity fallback.

Select **CHANGE**, then provide a control from the chosen device. Its keyboard Escape, gamepad East or joystick Cancel can abort; the visible Cancel control and timeout also work. Other devices cannot choose/cancel that capture. After capture, release held controls; focus returns to the Change button. Conflicting different actions in one gameplay map and Keyboard A/B overlap are rejected without changing the save. Separate controller devices may reuse equivalent buttons; lobby/UI maps may reuse gameplay controls.

- A row's **RESET** resets that profile/action; a Move reset restores all its parts.
- **RESET PROFILE** resets the selected source's profile after confirmation.
- **RESET ALL CONTROLS** clears all saved framework profiles after separate confirmation.

The sample uses the framework JSON store at `Application.persistentDataPath/CouchCoopBuilder/input-bindings.json`; customers can substitute `IInputBindingProfileStore`. Unreadable saves disable editing rather than silently overwriting data. Overrides conflicting with changed defaults require correction/reset; unmatched old binding addresses remain unapplied. Settings previews create no fake players or InputUsers.

Pause and Device Lost

The shared Pause panel reads each player's owned Pause action: defaults **T / Y / Start / DragonRise button11**. Joined players with available input can navigate using their existing lobby controls; mouse also works. Pause is available during countdown and gameplay; the Winner screen owns its separate return flow.

Pause sets `Time.timeScale` to zero, freezing physics, projectiles, scoring, countdown and respawn timers. Music/gameplay SFX pause; UI audio continues. Resume restores the same voices and timers. Movement, Jump, Fire and Shield require neutral controls followed by fresh input; shield cooldown is retained. Scene exit restores time.

Physical Device Lost auto-pauses and retains the session, representation and score. The panel identifies unavailable seats. Device regain/reclaim clears the warning but **does not resume automatically**; Resume remains blocked while required input is missing. Replacement-device ambiguity follows the framework's [reserved-controller recovery policy](#), with no invented cross-launch identity.

RETURN TO LOBBY keeps sessions, Player Index values and selected characters, including disconnected reservations. Ready and match score reset for the next round. **MAIN MENU** permanently ends all sessions; the next PLAY begins with zero joined players. Saved audio preferences and input profiles remain.

Drop Out and Rejoin

In an active match, the owned Leave control (**G / Backspace / Select / DragonRise button12**) requests reversible **Drop Out**. The session, Player Index, character, score, InputUser and profile remain reserved. The sample releases the representation and gameplay action clone, removes that player's rockets, and displays **DROPPED OUT / PRESS JOIN TO RETURN**. Survivors continue; Drop Out causes no death/score or automatic win.

The same role/controller's normal Join restores that session with a new selected-character representation at a safe authored spawn. Shield resets Ready; held gameplay controls need fresh input. Dropping out while dead cancels pending respawn. If everyone voluntarily drops out, **ALL PLAYERS LEFT / PRESS JOIN TO RETURN** pauses the arena; the first valid Join restores play. DeviceRegained alone does not reverse voluntary Drop Out.

This differs from Device Lost, which keeps the representation, and explicit permanent `TryLeave`, which releases the session/seat. See [framework lifecycle APIs](#) when integrating your own game.

What to study and adapt

Shared components implement authored menu panels, audio preferences, profile editing, Pause, lobby feedback and safe-point selection. They consume public framework APIs without creating another ownership model. Both games use Gameplay Player Setup for spawning and Gameplay Player Binding for isolated owned actions. Movement, combat, score, respawn, camera follow and match policy belong to the samples.

Test your copy with supported hardware: navigate settings without mouse, edit/reset a profile, return lobby/game, Drop Out/Rejoin a middle player, and test actual Device Lost separately. Surviving Player Index/session mappings should stay stable. These examples are local sample games, not networking, arbitrary-game camera controllers or performance guarantees.

CHAPTER 16

Rocket Rumble 2D

An optional **2–4-player** platform combat sample for Couch Co-op Builder, Unity 6.3 LTS, URP and Input System. It demonstrates selected-character spawning, isolated gameplay input, profile rebinding and persistent session flow. Its fixed Shared Camera, movement, combat and match rules are game content; framework scope is explained in [Player scope](#).

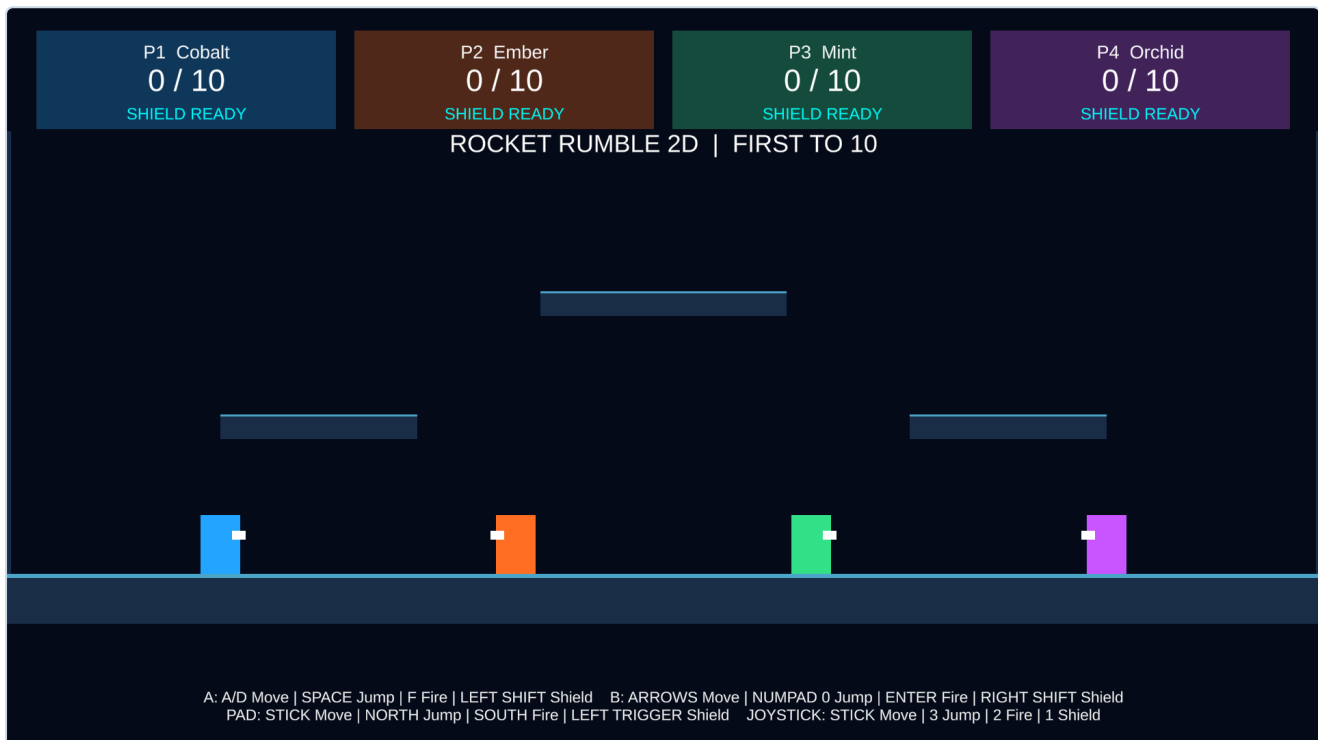


Figure 4. Rocket Rumble 2D uses one fixed shared view and independent player combat actions.

Open and play

Open RocketRumble2DMainMenu

([Assets/LevelArgento/CouchCoopBuilder/Samples/RocketRumble2D/Scenes/RocketRumble2DMainMenu.unity](#)), enter Play Mode and choose **PLAY**. Join at least two players, browse Cobalt/Ember/Mint/Orchid, confirm Ready and activate **START GAME**. First register its three scenes using [Configure Supplied Samples](#). Customize your own copies of the supplied scenes, catalog and prefabs; keep RocketRumbleShared when using this sample.

[Shared Rocket Rumble guidance](#) covers Main Menu, Audio/Controls, Pause, Device Lost, Drop Out/Rejoin and scene returns. Lobby controls are separate from the gameplay controls below.

Gameplay controls

Source	Move	Jump	Fire	Shield	Pause / winner return
Keyboard A	A / D	Space	F	Left Shift	T
Keyboard B	Left / Right	Numpad 0	Enter	Right Shift	Y
Gamepad	Left stick horizontal	North	South	Left Trigger	Start
Supplied DragonRise profile	Stick horizontal	button3	button2	button1	button11

Join/Leave defaults are Q/G, P/Backspace, Start/Select and DragonRise button11/button12. Gameplay actions are independently rebindable; this game does not consume Look or vertical Move. Up/stick Up never jumps. One press requests one grounded jump; holding through landing does not jump again. Dead/countdown/winner states reject Jump without audio.

Arena and jump

Rigidbody2D movement retains facing on release, including horizontal air control. Jump speed is 17.5, with body gravity scale 3 and airborne multipliers 1.4 rising / 2 falling. A floor jump reaches approximately 3.54 units, peaks around 0.40 seconds and lands around 0.76 seconds. The first platform is reachable; the center platform needs intermediate traversal. Solid undersides block upward passage, and side-wall contact is not grounding. Tune the supplied player prefabs for your game's scale.

Rockets fire horizontally with a 0.35-second cooldown, speed 18 and lifetime 2.5 seconds. They ignore their owner, consume on a valid hit and have no splash damage. One unblocked opponent hit causes one death and one attacker point; falling is an unscored death.

Shield, respawn and match

Hold Shield to raise a frontal energy plate while moving/firing. Releasing without a block costs nothing. A frontal incoming rocket is consumed without death/score and starts a five-second cooldown; rear attacks bypass protection. A second rocket can kill after the shield is spent. HUD displays Ready, Active or remaining cooldown. Death/respawn resets Shield and requires a fresh press.

The authored **3 → 2 → 1 → GO!** countdown blocks gameplay. Default first to **10 kills** wins, with **five-second respawn**. The same object, context, InputUser, selection and score survive ordinary death/respawn. Eleven authored floor/platform points prefer at least five units from living opponents, avoid the previous point when possible, and otherwise maximize separation. Missing respawn points fall back to initial authored spawns.

Victory stops gameplay/scoring, clears rockets and displays the winner. Any joined player with available input can use their Pause shortcut or the pointer Return button to return to the originating lobby. Identity/selection persist; Ready and the next match's score reset. Reversible Drop Out instead releases/recreates the representation while keeping score; [shared lifecycle guidance](#) explains that distinction.

Integration to study

- [GameplayPlayerSetup](#): Spawn Selected Character + Explicit Spawn Points; the framework owns instantiation and scene cleanup.
- [GameplayPlayerBinding](#): context and Built-in Actions; the match consumes [PlayerResolved](#) / [PlayerReleased](#) and replays already-resolved objects.
- [RocketRumble2DPlayerController](#) / [RocketRumble2DProjectile](#): owned actions, physics, facing and combat.
- [RocketRumbleMatchController](#) / [RocketRumbleRespawnPoints](#) / [RocketRumbleHud](#): session-owned score, safe respawn, authored TMP feedback and match flow.
- [Shared menus and settings](#): included sample audio and framework Input Binding Profile integration.

This sample uses one fixed Shared Camera. [Rocket Rumble 3D](#) demonstrates runtime Split Screen; [Split Screen Playgrounds](#) demonstrate up to eight participants. The 2D game is not a generic movement, matchmaking or arbitrary-content performance solution.

Try your integration

Start with Q/P, confirm independently and move to Start without losing Ready. Move/jump/fire/shield independently, die and respawn, then return after victory. Change Fire to an unused key in Controls and confirm that lobby Confirm stays unchanged. Test Pause/Resume, Drop Out/Rejoin and physical Device Lost separately; repeat with your supported controllers and check Console on scene exit.

CHAPTER 17

Rocket Rumble 3D

An optional **2–4-player** XZ combat sample for Couch Co-op Builder, Unity 6.3 LTS, URP and Input System. It demonstrates selected-character spawning, isolated actions, profile rebinding, persistent sessions and optional individual cameras. Movement, combat, score and follow behavior are sample-owned; see [Player scope](#) for framework capacity.

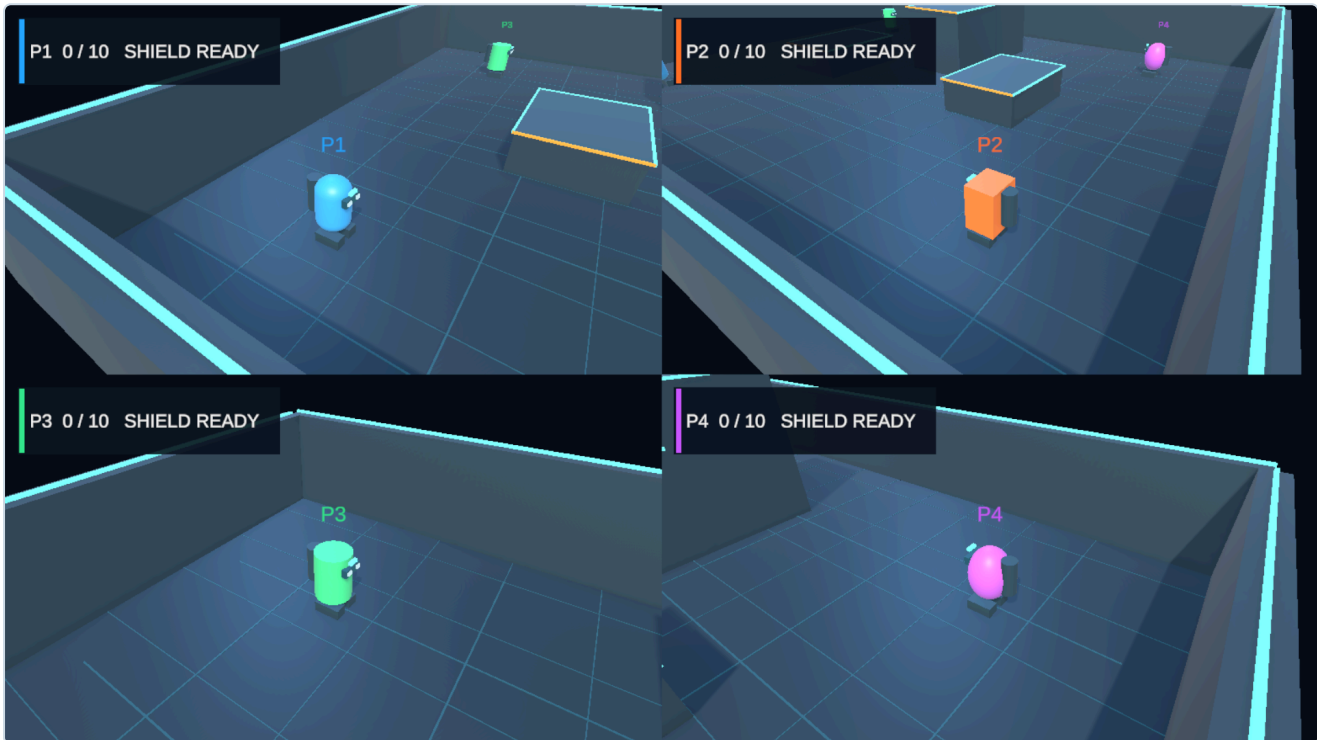


Figure 5. Rocket Rumble 3D: four assigned views with per-player HUDs. Pause and settings remain global overlays.

Open and play

Open RocketRumble3DMainMenu

([Assets/LevelArgento/CouchCoopBuilder/Samples/RocketRumble3D/Scenes/RocketRumble3DMainMenu.unity](#)), enter Play Mode and choose **PLAY**. Join/confirm at least two players, then activate **START GAME**. First register its three scenes using [Configure Supplied Samples](#). The supplied catalog has Cobalt, Ember, Mint and Orchid, with separate 3D gameplay and preview prefabs. Customize your own copies; keep RocketRumbleShared.

[Shared Rocket Rumble guidance](#) explains Audio/Controls settings, Pause, Device Lost, Drop Out/Rejoin and scene returns. Gameplay rebinding leaves lobby/menu controls unchanged.

Gameplay controls

Source	Move	Jump	Fire	Shield	Pause / winner return
Keyboard A	WASD	Space	F	Left Shift	T
Keyboard B	Arrows	Numpad 0	Enter	Right Shift	Y
Gamepad	Left stick	North	South	Left Trigger	Start
Supplied DragonRise profile	Profile stick	button3	button2	button1	button11

Join/Leave defaults are Q/G, P/Backspace, Start/Select and DragonRise button11/button12. Controls exposes all four keyboard Move directions, Jump, Fire, Shield and Pause / Winner Return; controller rows use vector Move. The sample consumes the framework's profile identity/conflict/reset/storage APIs, described in [Input Binding Profiles](#).

Arena, movement and jump

The arena has a 26×22 floor, boundaries, low/middle/high platforms, four indexed initial spawns and twelve safe respawn points. Move projects the shared camera's forward/right onto XZ, clamps diagonal speed to six units/second and retains facing when stopped. Move.y means forward/backward, never Jump.

Jump uses an independent grounded press at velocity 9.3. Per-player airborne acceleration is 2.2 times gravity rising / 3.2 falling; project gravity is unchanged. Approximate floor-jump height is 1.91 units, peak time 0.40 seconds and landing time 0.76 seconds. The 1.2-unit platforms are reachable; 2.8/4.2-unit platforms need intermediate traversal. Grounding accepts floor/platform normals, not side walls. Holding Jump through landing does not repeat it.

Rigidbody/Collider 3D physics, interpolated movement and frozen physical rotation keep presentation facing separate. Player capsules ignore one another; other collisions remain enabled. Rockets ignore their owner, travel horizontally at 20 units/second, expire after three seconds and consume once on hit/block/environment collision. They do not aim, home, splash or arc.

Shield, respawn and match

A frontal incoming rocket spends the held Shield without death/score and starts its five-second cooldown; rear/outgoing rockets bypass protection. Raising it leaves movement/fire available; release without a block costs nothing. Death/respawn resets Shield and requires fresh input. HUD shows identity, score, respawn and Shield state.

One unblocked opponent hit awards one attacker point. Ordinary death respawns the **same** object/context/InputUser after five seconds, retaining selection and score. Twelve authored points use XZ opponent distance: prefer at least six units separation, avoid the previous point when possible, otherwise maximize nearest-opponent separation. Falling below Y=-10 is unscored.

The **3 → 2 → 1 → GO!** countdown blocks gameplay. Default first to **10 kills** wins; victory freezes gameplay and clears rockets. Any available joined player's Pause shortcut or pointer Return button returns to the originating 3D lobby. Ready and the next match's score reset. Reversible Drop Out releases/recreates the representation while retaining score; Device Lost keeps it. See [shared lifecycle guidance](#).

Integration to study

Read the sample's [GameplayPlayerSetup](#) (Spawn Selected Character + Explicit Spawn Points), [GameplayPlayerBinding](#) (Built-in Actions), 3D player/projectile controllers, match/respawn/HUD components and camera settings integration. The framework owns sessions/users/isolated actions; the game owns combat, scoring, respawn, camera movement and presentation. [Shared menus and settings](#) supplies the common authored UI, input profiles and included sample audio.

This game has four authored slots. For an independent 1–8-player camera example, use [Split Screen Playgrounds](#). Neither sample promises camera collision/framing or performance for arbitrary customer levels.

Try your integration

Join Q/P, confirm and move to Start without clearing Ready. Test independent XZ move/jump/fire/shield, safe respawn and winner return. Use Controls to change a gameplay action while lobby Confirm remains unchanged. Test Pause/Resume, Drop Out/Rejoin and actual Device Lost separately. For cameras, navigate Pause → Camera Settings without mouse; toggle Shared/Split, change 2P/3P layouts, Back and reopen. Repeat with supported controllers and check Console on scene exit.

Optional Split Screen cameras

The saved Arena scene includes a GameplayCameraManager with the authored shared camera, an authored player-camera prefab, built-in layouts and the default single AudioListener policy. Split Screen defaults OFF. OFF keeps the original shared 47-degree view and full four-player HUD. ON creates one assigned follow camera per active representation (one to four), plus a compact P#/score/Shield card in each view. Countdown, Pause and Winner remain full-screen overlays.

The follow camera uses GameplayCameraAssignment and its matching GameplayPlayerBinding. It tracks position with a fixed heading inherited from the shared camera, not player facing. Gameplay movement continues to use the shared camera's orientation even while that camera is disabled for rendering. Wide horizontal views use a modestly narrower field of view and lateral arena bias so players stay visible near the edge. There is no orbit, aim camera or collision rig.

During a match, press Pause, navigate to CAMERA SETTINGS using Up/Down, and press Confirm/Submit to open it. Mouse users can click CAMERA SETTINGS on the Pause Card. Focus starts on Split Screen with the menu's orange selection accent. Up/Down selects Split Screen, the visible layout rows, Reset Defaults or Back; Left/Right changes the selected option (or selects a footer button). Confirm activates the focused control. Escape for keyboard roles or the owned controller Back action returns to the Pause menu, as does Back. The authored buttons also support pointer clicks. Resume, Lobby and Main Menu keep their existing pause behavior. Changes apply immediately, including while time is frozen. The 2-player choices are Vertical and Horizontal; the four 3-player choices are Top 2 / Bottom 1, Bottom 2 / Top 1, Left 1 / Right 2, and Right 1 / Left 2. Four players use the framework's 2x2 grid.

Rocket Rumble 3D alone saves these choices under RocketRumble3D.Camera.* using a replaceable IRocketRumble3DCameraPreferences adapter backed by PlayerPrefs. Reset Defaults restores OFF, Vertical and Top 2 / Bottom 1. Camera preferences are sample settings, separate from input binding profiles and the generic camera API. To customize the integration, edit the saved GameplayCameraManager, player-camera prefab and Camera Settings controls in your own copy of the sample.

A camera follows a respawned player without changing its session or owned input. Device loss retains the assigned view while Pause waits for recovery. Intentional Drop Out removes the view; the remaining cameras retile without changing their P# identities. Rejoin restores the reserved view. Returning to Lobby releases all scene-local player cameras. Camera rect layouts are deterministic; the sample does not add dynamic Split Screen transitions or camera collision.

CHAPTER 18

Split Screen Playgrounds

These **1–8-player** camera examples complement [Quick Start](#). Open SplitScreenPlayground3D ([Assets/LevelArgento/CouchCoopBuilder/Samples/SplitScreenPlaygrounds/Scenes/SplitScreenPlayground3D.unity](#)) or SplitScreenPlayground2D ([Assets/LevelArgento/CouchCoopBuilder/Samples/SplitScreenPlaygrounds/Scenes/SplitScreenPlayground2D.unity](#)) and enter Play Mode. These independent scenes use URP and the installed Input System/Smart UI Focus & Navigation integration. Their lightweight join screen needs no separate lobby scene, Rocket Rumble content or Cinemachine. They are not automatically registered in the project's scene configuration; add the chosen scene to the scene list used by your active Build Profile when building a sample.

Try it

1. Join with **Q** (Keyboard A), **P** (Keyboard B), or a controller's **Start**. The authored 4x2 grid shows eight equally sized P1-P8 cards with waiting/joined state and input source. Up to eight real input sources may join; the two keyboard roles share one physical keyboard. There is no Ready requirement.
2. Select **ENTER PLAYGROUND** with **F** (Keyboard A), **Enter** (Keyboard B), a controller profile's Confirm (**south face button** on standard gamepads), or a mouse click. One through eight joined players may enter. Existing representations, contexts and InputUsers are retained; gameplay movement/settings yield to the join screen until entry.
3. Move with **WASD**, **arrow keys**, or the controller stick. Jump with **Space**, **numpad 0**, or the gamepad's **west face button** (X on Xbox / Square on PlayStation). These scenes use the framework's default gameplay bindings.
4. Open **Camera Settings** with **Tab**, its mouse button, or a joined player's Pause binding (**T / Y / Start**). The Split Screen control receives focus. Player movement yields while the panel is open; world physics continues.
5. Use up/down to select and left/right to change. Keyboard A uses **W/S, A/D, F**; Keyboard B uses **arrows, Enter**; gamepads use **stick/D-pad, south face button**. Mouse arrows and Back also work. Close with **Escape**, gamepad **east**, or **Back**. Layout rows are hidden while Split Screen is OFF.
6. Toggle Shared -> Split -> Shared -> Split. Change both two-player layouts and all four three-player layouts. Camera settings are scene-local and begin with Split Screen OFF (shared view), two-player Vertical, and Top 2 / Bottom 1.
7. Drop out with **G / Backspace / controller Select**. Rejoin with the same source's **Q / P / Start**. Drop Out reserves the original session and PlayerIndex; it destroys that sample representation and recreates it at its spawn on rejoin. It is distinct from temporary DeviceLost, which retains the representation and view.

Configured generic joysticks use their [GenericControllerProfile](#) controls. The scene bootstrap includes the existing DragonRise profile; add other recognized profiles to its Shared Joystick Input list. Matching/pairing and gameplay binding resolution remain framework responsibilities.

Integration to read

- **PlaygroundSessionProvider** implements public [IGameplayPlayerProvider](#) and supplies a simple authored prefab per context to [GameplayPlayerSetup](#). It owns representation creation/destruction, eight spawn transforms, participant collision exclusions and palette. Sessions, InputUsers and input actions are provided by the framework.

- **PlaygroundJoinPanel** reads existing contexts for eight authored cards and invokes the authored Enter button from each player's owned Confirm action. It creates no sessions/devices/InputUsers. The join panel is the only navigation owner while visible; settings yield until entry. Spawns use two clear 3D plaza rows and eight non-overlapping 2D floor positions.
- **PlaygroundCameraSettings** calls public `GameplayCameraManager.SetSplitScreenEnabled`, `SetTwoPlayerLayout` and `SetThreePlayerLayout`. One settings router reads the existing owned menu actions. Smart UI Focus & Navigation manages focus recovery; the UI module handles pointers only, with no second move/submit/cancel router. All visible controls and HUD are saved in the scenes.
- The manager owns assignment, camera lifetime, active viewport packing and the single AudioListener policy. Both scenes use its camera-prefab source and the same default layout provider. Active `PlayerIndex` values pack in ascending order even when there are gaps.
- **PlaygroundFollowCamera3D** reads its public `GameplayCameraAssignment.Representation`, follows smoothly with a fixed 45-degree heading, and uses a little more distance for narrow viewports. Its perspective prefab is independent of player facing. Shared 3D uses a fixed authored high-angle overview that covers the usable playground.
- **PlaygroundCamera2D** supplies orthographic behavior. Individual cameras follow the assigned representation; the shared camera frames the active group using the public setup lookup. Follow size, maximum shared size and simple level bounds are inspector settings. Extreme separation zooms the shared view out rather than tethering players.
- **PlaygroundPlayer3D / PlaygroundPlayer2D** read only their own `GameplayPlayerBinding.Input`. They demonstrate move, grounded jump and facing. Falling below the sample floor returns the same representation to its spawn; there is no combat, score, health or full game loop.

The follow scripts are sample implementations. GameplayCameraManager does not require these behaviors. Replace them with your project's camera behavior while keeping the public assignment.

Presentation and limits

The 3D deck has four color districts, a central plaza, stepped terraces, ramps, a bridge and modest obstacles. The 2D stage has four districts, increasing platform heights and small floor gaps. Player colors and compact P1-P8 markers match the framework conventions.

See the authoritative [1–8 layout table](#) for selectable 2P/3P and automatic 4–8 arrangements. Viewport badges use one-based P1–P8 labels for the actual zero-based PlayerIndex, not the packed viewport number. The header and Camera Settings stay global above the cameras.

Eight cameras render the scene eight times. Small viewports reduce detail/readability and increase rendering cost. This sample has one unshadowed directional light in 3D, unlit materials in 2D, no post-processing and no particles. Profile your own content and target hardware. Shared overview players can be small; the 3D follow example has no occlusion avoidance or camera collision rig, so geometry can briefly obscure a player. The 2D bounds limit empty edges in individual views; an extremely wide shared view necessarily shows space outside the vertical level.

Try your integration

Join the player count your game supports, enter, and toggle Shared/Split Screen. Confirm each view follows its owning representation and only one assigned-camera AudioListener is active. Test two-/three-player choices and automatic four-/five-/six-/seven-/eight-player layouts. Drop Out a middle player: views pack by ascending PlayerIndex without renumbering the surviving sessions. Rejoin the same source, test actual Device Lost separately, then exit Play Mode and check for teardown errors.

Customize your own copies of the saved scenes and prefabs directly. These examples demonstrate camera integration, not a complete game or a promise of camera framing/performance for every level.

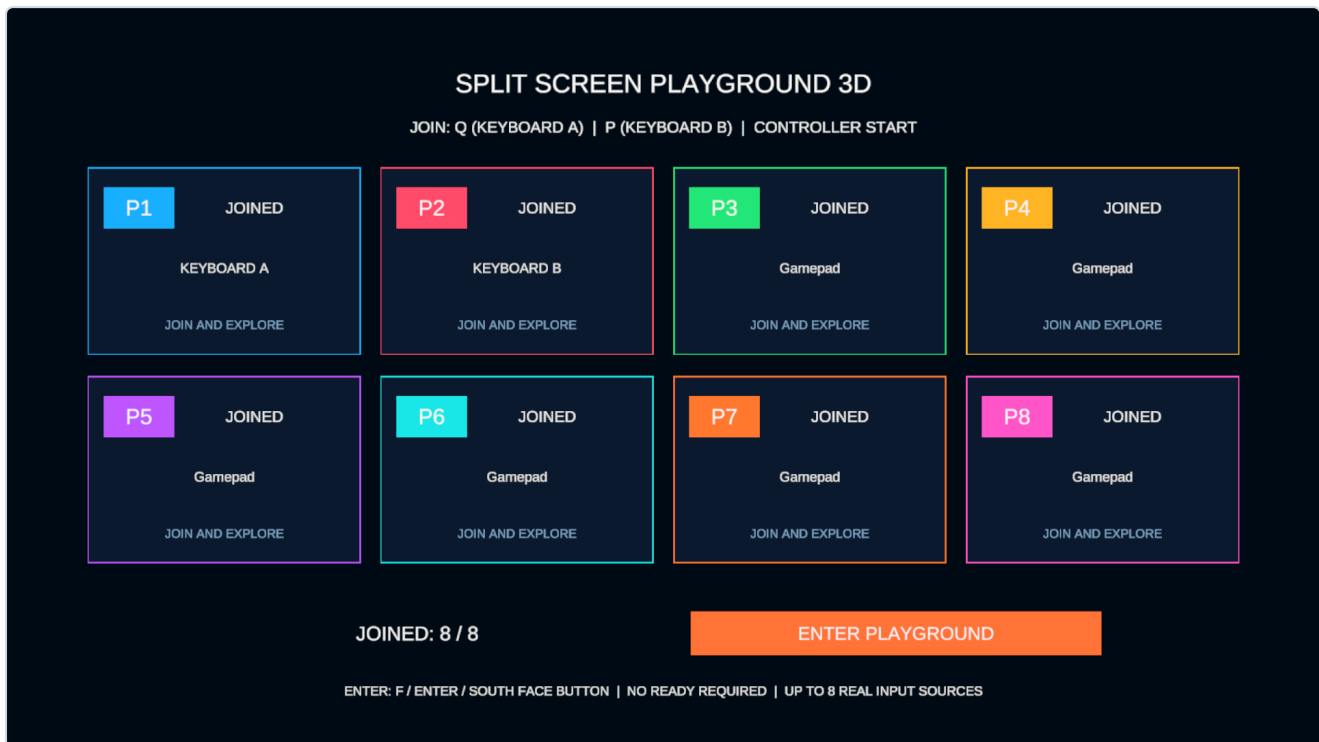


Figure 6. Eight authored Playground join cards. This independent scene does not require character Ready.

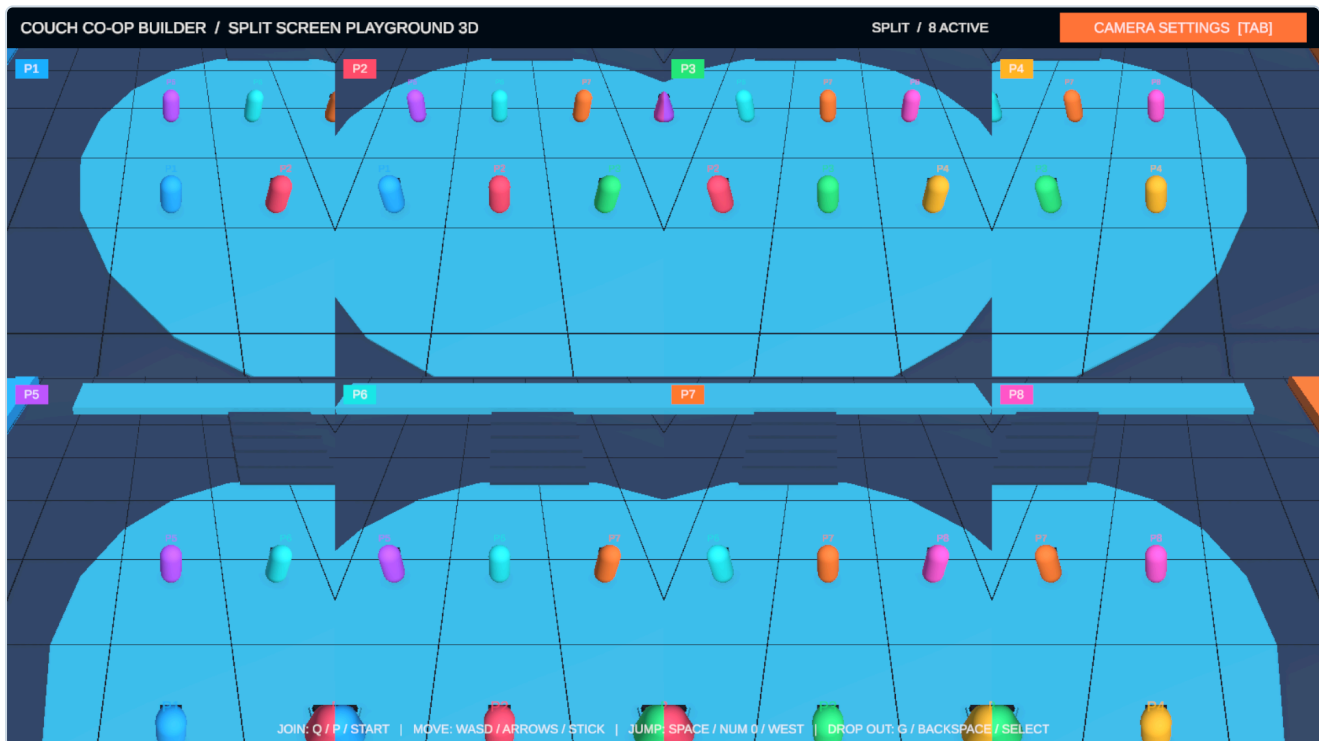


Figure 7. The Playground demonstrates eight independent camera assignments. Profile performance for your own content.

CHAPTER 19

Included Smart UI Focus & Navigation

Smart UI Focus & Navigation is **included** with Couch Co-op Builder: runtime, inspectors, customer documentation and examples. No separate purchase or installation is required. It is also sold independently by LevelArgento. Begin with [Quick Start](#) for multiplayer setup; Smart UI provides focus/navigation rather than Couch player/session ownership.

It supplies independent focus, focus recovery, directional navigation and focus scopes/modals for authored UI. Couch integrates independent local-multiplayer UI contexts on top of this module.

Manual and components

Use the offline PDF manual ([Assets/LevelArgento/SmartUINavigation/Documentation/Smart UI Focus & Navigation - Documentation.pdf](#)) or documentation entry

([Assets/LevelArgento/SmartUINavigation/Documentation/README.txt](#)) for SmartFocusManager, SmartFocusScope, SmartInputModeDetector, modal focus, navigation and layout constraints. Unity UI and Input System are required Unity packages. The manual's import instructions also apply when this module arrives as part of Couch Co-op Builder.

Couch's [Existing Canvas workflow](#) creates/reuses independent player focus contexts and uses one shared Start router. Keep player EventSystems and input modules separate; use the manual for advanced focus behavior.

Samples

Open a scene and enter Play Mode. These examples demonstrate Smart UI itself; they do not create Couch player sessions or a multiplayer lobby.

Open	Purpose
QuickStart (Assets/LevelArgento/SmartUINavigation/Samples/QuickStart/QuickStart.unity)	Basic automatic focus, navigation and pointer switching
FocusManagementSample (Assets/LevelArgento/SmartUINavigation/Samples/FocusManagement/FocusManagementSample.unity)	Focus scopes, restoration and modal behavior
NavigationShowcase (Assets/LevelArgento/SmartUINavigation/Samples/NavigationShowcase/NavigationShowcase.unity)	Navigation choices and authored UI layouts

Owning both products and updates

Keep one canonical module at [Assets/LevelArgento/SmartUINavigation/](#) ; do not duplicate it in another folder. Both products use this shared location. Back up local edits before updating either product, because importing another revision can replace files while preserving serialized references. The included module does not grant a separate Asset Store download entitlement.

Couch can warn about an older, missing, mixed or unvalidated Smart UI revision. Use **Tools > LevelArgento > Couch Co-op Builder > Validate Smart UI Compatibility** to check manually. A supported installation is silent; the check never repairs files. If warned, restore a complete revision supported by your Couch release by re-importing/updating Couch or using a publisher-validated Smart UI revision. A version label alone is not proof of compatibility.

CHAPTER 20

Troubleshooting

Symptom	Likely cause	Action
Catalog missing, empty or stale	Definitions/IDs/prefabs are missing, duplicated or changed	Generate or assign definitions; select the catalog and use Run Catalog Validation . Keep Character IDs stable. The built-in lobby selection requires a valid catalog with Gameplay Prefabs, even if your gameplay uses existing/custom objects.
Too few player panels	Supplied visuals have four slots, or authored roots are missing	Use Existing Canvas with one complete root per configured player, or reduce the supplied-lobby capacity.
Controller does not join	Capacity is full/reserved, device is unavailable, or its layout/profile does not match	Check Unity Input Debugger, the assigned profile and its Join binding. A profile-matched HID must expose the required controls. Identical display names are not device identity. See device recovery .
Start Game unavailable	A joined player is not Ready, input is unavailable, or no player is joined	Confirm each player; reconnect reserved input or intentionally Leave that session. Check the target scene is enabled in the scene list used by the active Build Profile.
Existing Canvas cannot apply	Required assignments, hierarchy boundaries or saved scene are missing	Use each Player panel entry; assign fields marked * and resolve listed errors. Optional labels/graphics can stay empty.
Gameplay has no player objects/input	API-only mode is selected, representation references are missing, or no input mode is configured	Configure Gameplay Player Setup and Gameplay Player Binding, or consume GameplayPlayers through your own integration. Follow the isolated runtime diagnostic.
Split Screen has no views/follow	Representations or a camera source/follow script are missing	Resolve gameplay objects, assign the camera prefab/representation/provider, and supply follow behavior. See Chapter 10 — Shared and Split Screen Cameras .
Smart UI compatibility warning	Older, missing, mixed or unvalidated shared module	Back up local edits; restore a complete compatible revision. Use Tools > LevelArgento > Couch Co-op Builder > Validate Smart UI Compatibility . No automatic repair occurs.

CHAPTER 21

Advanced API Integration

Owned analog selector navigation

See [Lobby and Existing Canvas](#) for normal authored navigation. This advanced API is useful for custom selector presenters.

[InputNavigationGate](#) converts one input owner's Vector2 callbacks into intentional selector steps. Create one gate per route, feed both performed and canceled callbacks into [Read](#), and poll [Repeat](#) with the same monotonic time source. Dispose/reset the gate when its route is released. The character-selection lobby template and standalone persistent selector use unscaled Unity time.

Default [InputNavigationSettings](#):

- Activation threshold: 0.6 on the dominant axis.
- Release threshold: 0.3; hysteresis prevents noise from generating new steps.
- Initial repeat delay: 0.35 seconds.
- Repeat interval: 0.16 seconds, with at most one step per poll.

Crossing activation generates one immediate step. Center release rearms it. A clear direction reversal generates an immediate new step; a partial reversal cancels repetition in the old direction. Repeated analog callbacks do not themselves advance the selector. Keyboard and D-pad input retain immediate discrete behavior without these repeat timers.

[IsAnalog](#) recognizes sticks, including joystick sticks, and excludes keyboard keys and D-pad controls. Settings are exposed on each presenter integration; timer/latch state is owned by each session route, never by `PlayerIndex` or a global controller.

Public integration map

Need	API / guide
Read authoritative players and session identity	CouchCoopBootstrap.GameplayPlayers : Sessions and Identity .
Bind existing objects or resolve a customer provider	GameplayPlayerSetup : Gameplay Integration
Spawn selected characters and provide placement	Representation and spawning .
Supply custom gameplay actions	IGameplayInputAdapter : Gameplay Input
Store or migrate input binding profiles	Input and Rebinding .
Manage camera ownership, providers and layouts	Shared and Split Screen
Handle reserved input, Drop Out and Rejoin	Session Lifecycle

Read the ownership and initialization caveats before integrating callbacks. Session `PlayerIndex` is a reusable seat; it is not a save identity for character definitions, physical controllers or input binding profiles.

CHAPTER 22

Support and Resource Notices

Contact support@levelargento.com for support. Include your Unity version, input hardware, the workflow or guide involved, relevant Console messages and concise reproduction steps.

Included content

The Couch Co-op Builder payload contains LevelArgento first-party source, samples, branding and project-generated media, including Smart UI Focus & Navigation. No Unity/TMP Essential Resource font, shader or sprite files are bundled.

Unity resource dependencies

Unity Input System, Unity UI / TextMeshPro and Universal RP are supplied separately by Unity Package Manager under their accompanying terms. UI samples reference the standard Liberation Sans TMP font supplied by **Window > TextMeshPro > Import TMP Essential Resources**. Its SIL Open Font License 1.1 and copyright notice are provided by Unity in [Assets/TextMesh Pro/Fonts/LiberationSans - OFL.txt](#). Keep the licenses accompanying resources you import. See [Installation](#).