Unfinished

Full custom-content integration is not yet verified.

COMMUNITY MOD DEVELOPMENT GUIDE

BALL x PIT Modding Guide

Creating custom balls and characters

Community research and a practical learning path • First compiled 26 September 2026 • Updated 30 September 2026

You can start making BALL x PIT mods without being able to draw. The best first project is a small change to an existing mechanic with a simple placeholder image. A completely new ball or character also needs the game to recognize it in selection, combat, progression and possibly saves. That integration is the harder part.

This guide focuses on Windows PC mod development. It brings together public game-specific examples, official tool documentation and suggested engineering workflows. It is not an official game SDK. **Source-confirmed** means a linked source documents the detail; **proposed workflow** means a development approach that still needs testing against your installed game. The examples in this guide have not been compiled or tested in BALL x PIT. No complete, verified custom-ball or custom-character registration recipe is established here.

Use sections 1–5 to get started, 6–8 to investigate custom content, and 9–10 for artwork. Sections 11–14 cover testing, troubleshooting, packaging and a first-project plan. Every external reference is clickable; the entire guide works offline apart from those links.

0 Update notes and current evidence

As of 29 September 2026: the public resources reviewed for this update still do not establish an end-to-end recipe for registering a new ball or character. Sections 7 and 8 remain research plans. This means the method is unverified here; it does not establish that nobody has done it or that it is impossible.

This revision adds an optional Vortex installation route, version-specific save findings, an example of configurable plugin settings and a more concrete way to record custom-content research. These are additions to the guide, not claims that the tools or mods were released after 26 September.

Evidence should be read at three levels:

- **Tool documentation:** establishes how a loader or API works in general. It does not prove a BALL x PIT content feature works.
- **Game-specific source or author report:** establishes an implementation or a reported result for the stated project. It still needs compatibility testing on your installation.

- **Local runtime verification:** establishes that the exact build and setup you tested actually behave as expected. The community ball/character examples in this guide were not game-tested for this update. Section 6A separately reports local runtime tests on one installed game build.

The recent mod listings reviewed here still describe adjustments to existing systems: More Speed 1.1 adds speed controls; Equip Unlimited Balls and Passives 1.0.0 raises equip capacity; Unlimited Resources 1.0.0 changes resources; Local Coop 0.1.0 adds a second local player. Their published dates are 20–23 September 2026. Increasing the number of equipped balls is different from defining a new ball type. More Speed · Equip slots · Resources · Local Coop

1 What is actually available

Source-confirmed: community mods use both MelonLoader and BepInEx. Auto Pickup documents MelonLoader installation. More Speed documents BepInEx 6 Bleeding Edge for Unity IL2CPP. A mod written for one loader is not automatically a plugin for the other. Follow the dependency instructions for the particular mod or project you choose. Auto Pickup · More Speed

The public devopsdinosaur repository contains working-mod source projects for studying gameplay changes. It is useful evidence that patching game behavior is practical; it does not establish an official content API.

Think of mod scope in four stages:

- **A gameplay adjustment:** alter a value or existing action. No new artwork is necessary.
- **A visual replacement:** change how an existing item appears while keeping its identity.
- **A prototype ability:** attach your behavior to an existing ball or character to test the idea.
- **A separate content entry:** introduce its own identity, selection entry, behavior and progression. This requires discovering how the game handles each of those pieces.

A ball that appears during a test is not yet a complete new ball. It also needs a way to be obtained, displayed, upgraded and cleaned up correctly. A replacement can be a finished mod in its own right; label it clearly when sharing it.

2 Pick a development route

Recommended starting route for this guide: BepInEx 6 Unity IL2CPP, because its official documentation provides a structured plugin tutorial and there are game-specific installation examples. This is a workflow choice, not evidence that it is universally more compatible than MelonLoader.

Choose MelonLoader instead if your immediate goal is to build upon the devopsdinosaur examples. Its official repository documents installation, requirements and logs. Gameplay mod DLLs go in the game's `Mods` folder; loader plugins have a separate purpose and location. Use a loader version compatible with the example and your game build.

For early development, use one loader at a time. Keep a written record of your setup: game version and store, loader version or build number, Unity version reported in the log, template version, target framework and the date you generated the game references. This makes an update-related failure much easier to diagnose.

Optional installation through Vortex

Author-documented: the BALL x PIT Vortex Extension lists version 0.1.2, last updated 22 April 2026. It supports automatic discovery of Steam, Xbox Game Pass PC and the Steam demo, and offers a choice of MelonLoader or BepInEx. Its notes specify BepInEx Bleeding Edge build 755; do not interpret that pinned build as a guarantee of compatibility with every later game update.

The extension distinguishes BepInEx plugins and preloader patchers when deploying files. It also documents switching loaders by removing the current loader in Vortex before choosing another. A mod manager helps install and track files; it is not a new-ball editor or content SDK.

Recommended development practice: use either Vortex deployment or your own manual deployment for a given DLL. Mixing both can leave an old managed copy that reappears during deployment. Record which route owns the installed files and check the loaded plugin version in the log. Vortex support for the Xbox PC installation does not mean console mod support.

Useful tools, introduced only when needed:

- **C# editor and .NET SDK:** create and compile your plugin. A runtime alone does not include the compiler. Follow the BepInEx environment setup.
- **dnSpyEx:** inspect .NET assemblies, names and signatures. Project
- **UnityExplorer:** an optional in-game inspection UI for Unity objects. Select a build matching your loader and IL2CPP environment; compatibility with the current BALL x PIT build must be checked. Project
- **AssetRipper:** inspect and export Unity asset data for local analysis. Its output is not automatically a working mod or a complete recovered game project. Project
- **Cpp2IL:** an advanced IL2CPP analysis tool, useful when names and runtime observations are insufficient. Recovery is imperfect and version-dependent. Project

3 Prepare the game and install the loader

Preserve a baseline

Launch the unmodified game and verify it works. Close it before changing files. Locate its installation directory through your store client; use the directory containing the real game executable, not a desktop shortcut.

Copy the entire save directory somewhere separate before experiments. The community save project documents the Windows location as:

```
%USERPROFILE%\AppData\LocalLow\Kenny Sun\BALL x PIT\
```

Its examples include `meta1.yankai` and a game-managed backup. A game-managed backup is not a substitute for your own copy because it can change during play. Treat the save documentation as version-specific research. Save format reference

Newly included save findings

Repository-reported: v1.301 resource editing was tested on Android 16; v1.299 is a historical fixture. The documented format uses BinaryFormatter with UTF-16LE field names. Its resource arrays contain a marker that must be preserved, while stone is stored separately. The v1.301 analysis reports 23 serialized character slots. These observations do not establish Windows compatibility or a way to append modded content. Save research

Recommended practice: compare copies from before and after one ordinary in-game action. Use the repository's inspection and diff tools according to their documented version support before experimenting with edits. Do not assume that changing a slot count creates matching game data, selection UI or a valid unlock.

Keep source code outside the game folder. Install only your built output into the game. Keep an inventory of loader files you add, so you can reverse the installation without guessing which files belong to the original game.

BepInEx setup

1. Use the official Unity IL2CPP installation guide and its linked builds.
2. Match the package to the executable architecture. For a confirmed 64-bit Windows executable, select `Unity.IL2CPP-win-x64`.
3. Extract the loader into the game root and launch once. The initial IL2CPP generation can take longer than a normal launch.
4. Check that configuration and logs were generated, then exit.
5. Test one compatible existing mod before introducing your own. This separates loader problems from mistakes in your code.

BepInEx configuration is under `BepInEx/config`. Its official installation page calls the initial log `LogOutput.txt`; some BALL x PIT mod pages refer to `LogOutput.log`. Look for the actual `LogOutput` file produced by your installed build instead of assuming the extension.

Some community mods report a Unity 6 startup workaround: set `UnityLogListening = false` in the `[Logging]` section of `BepInEx.cfg`. Treat this as a specific reported workaround, not a universal setup requirement. Preserve the original setting and investigate the log first. Reported workaround

4 Build a plugin that only logs a message

The first milestone is deliberately small: your DLL loads and writes one recognizable message. Do this before referencing game classes or changing gameplay.

The official setup page currently lists the following template installation command. These are instructions for your future development setup, not commands already run for you:

```
dotnet new install BepInEx.Templates::2.0.0-be.4 --nuget-source
https://nuget.bepinex.dev/v3/index.json
dotnet new bep6plugin_unity_il2cpp -n BallPitFirstMod
dotnet restore BallPitFirstMod
dotnet build BallPitFirstMod -c Release
```

Check template and installed-loader compatibility if restoring or loading fails. Retain the template's IL2CPP framework settings rather than copying target-framework advice from a Mono tutorial. Setup documentation

An illustrative minimal `Plugin.cs` is:

```
using BepInEx;
using BepInEx.Unity.IL2CPP;

[BepInPlugin("local.ballpit.firstmod", "Ball Pit First Mod", "0.1.0")]
public sealed class Plugin : BasePlugin
{
    public override void Load()
    {
        Log.LogInfo("Ball Pit First Mod reached Load");
    }
}
```

Copy your plugin DLL from the build output into a dedicated folder under `BepInEx/plugins`, launch and find that message in the log. Keep the plugin identifier unique and stable. The IL2CPP entry point uses `BasePlugin.Load`; do not substitute a Mono-only `BaseUnityPlugin.Awake` example. Plugin tutorial

Only after that succeeds, reference needed game assemblies from `BepInEx/interop`. Copy relevant references to a private local `lib` folder and configure project references. Do not indiscriminately add `System.*`, `mscorlib` or `netstandard` assemblies from the game. Do not ship copied game libraries with your mod. The same tutorial explains references and build output.

Checkpoint: one startup message, no new error, normal title screen and normal exit. A successful build alone does not prove the game can load the plugin.

Add configurable settings before adding gameplay

Tool-documented pattern; illustrative code not game-tested: BepInEx binds options through `Config.Bind` and exposes the current setting through `ConfigEntry<T>.Value`. This lets you test balance settings without recompiling for each value. Configuration tutorial

Replace the minimal example with this settings-only version to learn the configuration path. It logs settings but does not create a ball, apply damage or install a patch:

```
using BepInEx;
using BepInEx.Configuration;
using BepInEx.Unity.IL2CPP;

[BepInPlugin("local.ballpit.firstmod", "Ball Pit First Mod", "0.2.0")]
public sealed class Plugin : BasePlugin
{
    private ConfigEntry<bool> _enabled = null!;
    private ConfigEntry<float> _echoRatio = null!;

    public override void Load()
    {
        _enabled = Config.Bind("Prototype", "Enabled", false,
            "Enable the experimental mechanic once implemented.");
        _echoRatio = Config.Bind("Prototype", "EchoDamageRatio", 0.30f,
            new ConfigDescription("Proposed fraction of original damage.",
                new AcceptableValueRange<float>(0f, 1f)));

        Log.LogInfo($"Enabled={_enabled.Value}; ratio={_echoRatio.Value}");
    }
}
```

Proposed check: build, deploy one copy and launch. Locate the generated config under `BepInEx/config`, exit, change the setting and relaunch. Confirm the log reflects the saved value. Restart during initial testing rather than assuming edits are reloaded live. A gameplay patch must explicitly consult the setting; merely binding an `Enabled` option does not turn code off automatically.

5 Understand patches through real BALL x PIT code

A patch attaches mod code to a method the game already calls. A prefix runs before the original method; a postfix runs afterward. A prefix can sometimes suppress the original method, which is a larger behavioral change. BepInEx includes HarmonyX for patching; its API resembles Harmony. BepInEx patching overview · Harmony patch concepts

Source-confirmed example: the Movement Speed mod targets `Player.MyFixedUpdate`. Its prefix assigns the player's `SpeedMult` and returns `true`, allowing the original method to continue. That is a useful example of modifying a value while preserving the normal update path. These names are verified in this source revision, not guaranteed in every game release. Movement Speed source

Source-confirmed example: Auto Pickup checks that `GameMgr.I` exists and the state is `GameState.kPlaying`, finds pickup objects and calls `StartPlayerPickUp(0)`. This illustrates waiting for an appropriate game state instead of manipulating gameplay from the title screen. Auto Pickup source

Read these as examples to investigate, not files to paste blindly. Auto Pickup's patch names `GameMgr.Awake`, while its postfix declares a `LevelMgr` instance parameter. That mismatch deserves review. Its broad object search is also not a model to repeat for every ball on every frame.

The Movement Speed project imports `solution_private.targets` and has automatic deployment steps. Review those paths before building; a fresh checkout may need local configuration. Its shared targets use `net6.0` and MelonLoader-generated assemblies. Those are facts about that project, not settings to copy into every BepInEx project. Project file · Shared targets

Proposed first patch: observe one low-frequency event and log it once before changing it. Verify the exact target signature and overload. Add one reversible change, then retest. Repeated multiplication of a persistent value can compound unexpectedly; determine where the original value is established and restored.

6 Map the game before adding content

Proposed workflow: make a small research notebook. For each discovered type or method, record the full name, assembly, parameters, when it is called, what state it changes and the game build tested. Use real identifiers from your installation. Words such as “ball registry” below describe responsibilities, not verified class names.

Investigate the following path in order:

1. **Identity:** what distinguishes one ball or character? Look for enums, numeric IDs, names, data objects or lookup tables.
2. **Selection:** how does the game build the menu, reward list or starting loadout?
3. **Creation:** what constructs, clones or retrieves the actual object?
4. **Behavior:** where do firing, collisions, damage and special effects happen?
5. **Lifecycle:** what happens on spawn, reuse, death, return to base and restart?
6. **Persistence:** does the game save that identity, and how does it validate it on load?

Use assembly inspection for signatures and a compatible runtime inspector for live objects. Change only one thing during an observation, such as equipping a different ball, then compare the data and behavior. Log transitions rather than every frame.

IL2CPP interop assemblies are bridges for calling native game code. They do not necessarily contain readable original C# method bodies. An assembly browser can still be useful for structure. Cpp2IL may recover additional behavior, but its own documentation describes the tool as work in progress. Do not infer a complete algorithm from an empty wrapper. Cpp2IL documentation

The best result of an initial investigation is a reproducible finding: “this exact method fires once when this ball is created.” That is more useful than a large list of plausible class names.

6A Locally verified BALL x PIT observations

Source: local runtime capture, not a public SDK. On 29 September 2026, a separate MelonLoader 0.7.3 observer and runtime bridge were tested against BALL x PIT 1.301 (Unity 6000.0.62f1 IL2CPP). The generated interop build identity was `1.301:d447e578-124b-4133-a5c9-0aa521fb5621`. Other game builds need fresh inspection. The observer did not install a complete custom-content system.

Open the offline verified API browser for selected declared signatures and enum values from this exact installation. This is a declaration index: a method name alone does not prove its behavior or make it safe to patch. These observations are independent of the BepInEx example project above; the local tests used MelonLoader.

A repeatable evidence-first workflow

1. **Fingerprint the installation:** record game version, Unity/IL2CPP status, loader version and generated interop build identity. Preserve the DLL and log for the build you are testing.
2. **Inspect declarations:** identify exact overload signatures and relevant enum values in the interop output. Do not equate generated wrapper names with verified native behavior.
3. **Observe one ordinary run:** capture the active scene, game state, turn, battle clock and loaded object relationships. Record the same identifiers on the screen transition you need to understand.
4. **Change one reversible path:** if a hook is justified, log one call first, then modify only a narrow compatible case. Test startup, active play, pause, run end and normal quit before expanding scope.
5. **Keep an evidence record:** note the exact game/loader build, hook signature, what you changed, expected/actual result, and whether the result came from native gameplay or a managed test fixture.

What the local captures actually show

Area	Observed evidence	Limit
Spawn integration	The MelonLoader bridge used postfixes on `LevelMgr.GetPiecesToSpawn(LevelWave, int, Il2CppSystem.Random)` and `GridMgr.CreatePiece(GridPieceInst, bool)`.	Not a complete custom enemy/ball registration mechanism. Later Savanna CST runs had no confirmed modified waves.

Area	Observed evidence	Limit
	Narrow Desert tests confirmed actual created enemies.	
Arena	`GridMgr` borders and spacing were read. The observed grid spacing was 1.125 raw game units; Savanna borders expanded around turns 60–63 and 120–123.	No safe wall-writing, entity-push or editor-tile conversion method has been verified.
Definitions	57 catalog enemy entries were verified. Expanded capture saw 199 references: 79 `InfoDB.GridPieces`, 88 `LevelInfo.EnemyList`, eight `BossInfo`, 24 `PreviewEnemies`.	References are not unique assets. Global grid pieces are not automatically enemies.
Collision	Collider shapes, bounds and some `GridMgr` links were captured. The grid contact filter used mask 8704; Physics2D reported Script simulation.	These settings do not prove which query or contact code gameplay uses. No ball-contact samples were captured.
Battle UI	States included `kPlaying`, `kPaused`, `kLevelUp`, `kPickTreasure`, `kFoundBlueprint`, `kRevive`, end states and declared bonus/egg states.	Some state names are declaration-only. Do not infer that every UI screen freezes Unity time or has the same resume behavior.

Snapshot object IDs are local to one game session. Scene captures are incremental, not atomic; a file labeled “start” may have been overwritten at a later turn. The latest preserved Savanna run ended at turn 264 with zero capture errors, but that is evidence about observation reliability, not proof of a custom-level loader.

Still unknown for new balls or characters: the verified registration/selection path, prefab construction, save/progression integration, asset preview generation and safe teardown. Continue the research plans in sections 7 and 8; do not treat these declarations as an end-to-end recipe.

New-content registration research

Local 1.301 declarations, not a finished registration hook: `InfoDB` holds `Heroes/BaseHeroes/HeroOrder`, `Passives/PassiveOrder` and `Chars/CharHousing`. A ball candidate is `HeroInfo : UpgradeInfo` with `HeroType`; a passive candidate is `PassiveInfo : UpgradeInfo` with `PassiveType`; a character is `CharInfo` with `CharType`. Selection and save declarations also refer to these types. This points to multiple registries and consumers, so appending one array or inventing an enum value would be an unverified and likely incomplete experiment.

Live main-menu inventory (game 1.301, 30 September 2026): the loaded registries have 23 characters, 90 heroes/balls (21 base heroes), and 72 passives. Every loaded type matches its array index; each corresponding enum has one trailing `kNum` name. All character starter upgrades resolve to a loaded hero; `CharHousing` also has 23 entries. `HeroOrder` contains all 90 hero slugs exactly once. `PassiveOrder` contains 71 of 72 passive slugs, omitting only Spinach (slot 0), which is marked unavailable and has no icon. These are observed facts, not a tested way to extend the registries. No new ball, passive or character has been registered. The game also has a separate `PickupType`/`PickupMgr` path for collectible drops, so a new "item" needs to be scoped as a passive, pet upgrade or pickup. See the detailed research ledger for evidence, unknowns and the experiment order. The local Cpp2IL run produced raw native instructions but only stub reconstructed C# bodies for the key methods, so it did not prove the registration algorithm.

Verified normal level-up trace: the game offered `kHero:hupg_freeze`, `kPassive:pass_stoneeffigy` and `kHero:hupg_eggsac`. After Freeze was selected, the in-memory hero roster changed from `kEarthquake` to `kEarthquake,kFreeze`. A second level-up offered `kPassive:pass_eyeball`; selecting it added `kEyeball` to the in-memory passive roster while the ball roster stayed unchanged. These confirm one existing ball and one existing passive choice-to-roster path. A separate main-menu probe also created temporary, unregistered ball, passive and character definition clones and read back their test-only names and IDs; ball/passive `GetIdx()` matched the assigned numbers. Those numbers coincide with the trailing `kNum` enum sentinels. A read-only battle capture found in-memory progression arrays of 23 characters, 90 ball stats and 72 passive stats, matching the definition registries. The inspected native constructor also contains allocations with literal sizes 23, 90 and 72, matching the live progression arrays. A new ID must be tested against this fixed sizing as well as registration. A one-update test then expanded the ball/passive registry arrays and order lists, read the new slots at indices 90 and 72, and restored the original registry references. This verifies in-memory registry swapping only. The inspected native `LevelUpUI.PopulateUpgrades()` method also loops with fixed bounds of 90 balls and 72 passives, so expanding those registries alone is unlikely to expose the new indices in normal choices. A separate character trace observed `char_shade` selected and `kShade` in the following battle. All 23 character progression entries aligned by type with the definition array, while housing was absent for the default and influencer characters. Across 21 distinct observed level-up screens and 57 displayed choices, none of the shown slugs remained in the corresponding available-new list at capture time. One screen held 18 remaining base-ball and 37 remaining passive entries while showing three passive choices. This is consistent with drawn choices being removed before display, but the exact mutation sequence is unverified; these lists are not complete pre-draw candidate pools. Selection of a new ID, its behavior, and disk save/reload remain untested.

7 Develop a custom ball in stages

Current status on 29 September 2026: new-ball registration remains unverified in the reviewed resources. This section is a proposed workflow, not a verified registration tutorial. A suggested learning project is an "Echo Ball" that produces one weaker follow-up effect after a qualifying hit. The name, behavior and balance values are your design; they are not built-in BALL x PIT content described by this guide.

What would establish a verified method

A convincing example should supply source code, an exact game and loader build, and reproducible steps showing a separate ball identity entering the game's normal obtain-and-equip path. It should show the ability during a run, level progression where applicable, and a clean run restart. If the content is persistent, save/load and missing-mod behavior must be explained too. An isolated spawned projectile or a recolored existing ball establishes only that narrower result.

Proposed research record: use the following fields for each discovery. Enter actual names and results; do not turn the placeholders into supposed game APIs.

```
Game version and store:
Loader build and reference-generation date:
Question being tested:
Observed type and exact method signature:
Assembly containing that type:
Action that triggers it:
Expected observation:
Actual observation and log excerpt:
Confirmed scope and remaining uncertainty:
Reproduction steps:
```

Work through identity lookup, reward selection, object creation and UI display separately. Stop each investigation with a reproducible result before combining hooks. This gives a future guide update evidence stronger than a guessed class name or a successful compile.

Stage A Write a tiny design specification

Describe the trigger, effect, limits and visible feedback. For example: a hit schedules one follow-up hit at 30 percent of the original damage, with no recursive triggering. Decide whether the follow-up counts as a ball hit for other effects. Start with one controlled target and a visible placeholder.

Stage B Attach the experiment to existing content

Find the relevant spawn and hit paths. Gate your behavior to one selected existing ball and your own configuration flag. Do not apply a global damage change simply because it is easier to find. Prove the effect in a short run before adding menus or permanent unlocks.

Track ownership and per-shot state. If the game reuses objects through pooling, reset the state whenever an object is reused; this must be investigated rather than assumed. Remove delayed callbacks when a run ends so an old effect cannot fire in a new run.

Stage C Make the feedback readable

Use a distinctive tint, ring or short pulse. Keep the original collision shape during the first visual experiment. A larger sprite should not silently increase the area that deals damage. Test overlapping balls and busy scenes; the effect should communicate the mechanic without obscuring enemies.

Stage D Investigate an independent entry

Before declaring a true new ball, identify how to integrate:

- A stable identity and its data lookup.
- Display name, description and icon.
- Reward eligibility and any exclusions.
- Equip and removal behavior.
- Level progression and stat calculation.
- Evolution or fusion relationships, if your design uses them.
- Encyclopedia and other discovery displays, where relevant.
- Save and load behavior, including missing-mod handling.

Do not assume that adding a larger integer to an enum creates a valid entry. A fixed-size array or switch elsewhere may still reject it. If the game is extensible through data, extending the discovered data path may work; if it has many fixed assumptions, additional patches will be necessary.

If registration is not yet solved, release or keep the experiment as an explicitly named replacement/prototype. The mechanic can still be useful while the content integration remains unfinished.

Stage E Balance and verify

Test one ball, many balls, repeated hits, bosses, pause, death and restart. Set a hard limit on generated follow-ups and verify that one effect cannot trigger itself indefinitely. Compare actual behavior with the description and with your baseline recordings.

8 Develop a custom character in stages

Current status on 29 September 2026: independent character registration is also unverified here. Proposed workflow follows. A character includes more than a portrait: selection identity, presentation, starting loadout, ability ownership and potentially progression or unlock rules. The exact implementation depends on the installed game's structures.

Begin by giving one existing character a temporary experimental ability. A suitable concept is “The Echo Keeper,” whose qualifying shots interact with the Echo Ball mechanic. Keep the art as a placeholder until the gameplay is worth keeping.

Your investigation should answer these questions:

- Where does character selection become a runtime player configuration?
- How are initial stats, starting balls and upgrades applied?
- Which method can run your ability once per intended event?
- Does the ability belong to a character, a player, a projectile or a global manager?
- What clears the ability on a new run or a character switch?
- Which animations, directions, portraits and menu images are actually required?
- How are a new character's identity and unlock state saved?

Test one character first, then character combinations and any other supported ownership paths your mod touches. A community local-co-op author reports extensive assumptions around a single player and fragile interactions with unusual character mechanics. That is evidence to investigate ownership carefully, not proof that custom characters are impossible. Local Coop author notes

For a separate roster entry, verify that both mouse and controller navigation can reach it, that its preview matches its gameplay identity, and that switching away removes its effects. Avoid reusing a vanilla ID for a separate permanent unlock.

If you introduce a custom component into IL2CPP, ordinary C# class creation is not enough. Il2CppInterop documents registration and construction rules for injected types, along with lifetime limitations. Follow the pattern supported by your loader version, and do not use `new` to construct a Unity component as if it were an ordinary data class. This is an advanced step, not necessary for the first logging plugin. Class injection documentation

9 Make art without drawing by hand

Recommended approach: prove the mechanic with simple shapes, then improve only the art the game actually needs. A strong silhouette and clear color are more useful in a crowded fight than tiny detail.

Option A Build a ball from shapes

Use a circle tool for the silhouette, add a dark outline, a smaller highlight and one recognizable mark such as a ring, crack or diamond. Choose a small palette. For an Echo Ball, two offset rings can communicate repetition without illustration skills.

Start at the dimensions measured from a comparable in-game asset. A 32 by 32 canvas can be a practice exercise, but it is not a verified BALL x PIT requirement. Test the sprite at actual gameplay size rather than judging only an enlarged preview.

Option B Use an asset pack

Choose a pack with compatible perspective and proportions. Kenney states that assets on its asset pages use CC0; check the particular asset page and retain its license information. Free availability on a different site does not automatically mean the same terms. Kenney asset terms

A premade character may still need new animation frames and alignment. Pick assets by the animations required, not just by the portrait. Save time by matching an existing frame structure instead of redesigning it during the first mod.

Option C Use generated concepts

An image generator can help produce a design concept or an isolated icon. Expect cleanup for hard edges, transparent pixels and consistent animation. Generating a convincing sprite sheet in one image does not ensure consistent frames or usable slicing.

Example concept prompt:

```
One small fantasy projectile icon: a violet orb with two pale
offset rings, strong dark silhouette, limited palette, centered,
transparent background, no text, no scenery, readable at tiny size.
```

For characters, establish a single approved reference before making additional poses. Keep the same canvas, palette, scale, lighting and viewpoint. A single static character image cannot stand in for every animation the game expects.

Option D Recolor for a local prototype

Where technically feasible, use runtime tinting to distinguish a test object. Keep the experiment local while you learn asset assignments. For a distributed mod, package your own or appropriately licensed artwork and load original game assets from the user's installation where possible.

Piskel is a browser-based sprite editor with animation preview and sprite-sheet export. It is enough for small shape-based sprites and frame alignment. Start with one editor rather than learning an entire art toolchain.

10 Turn an image into a usable game asset

General Unity technique; game integration still requires investigation. A PNG sitting beside your DLL will do nothing until code loads it and assigns it to the correct renderer or UI element. Unity documents loading image bytes into a texture and creating a sprite from a texture rectangle. Image loading · Sprite creation

For the first asset, use a clearly recognizable placeholder and validate the whole path:

1. Resolve the asset path relative to the mod, not the process working directory.
2. Load the image once at an appropriate point on the Unity thread.
3. Create or obtain the needed sprite with the correct rectangle, pivot and pixels-per-unit.
4. Assign it to the discovered gameplay renderer or menu image.
5. Keep its texture and sprite alive while in use, and release your resources when appropriate.
6. Check whether animation or object reuse overwrites that assignment later.

Match the original asset's visual contract: canvas dimensions, transparent padding, pivot, scale, frame count, frame order and direction. Record these measurements before commissioning or generating final artwork. A centered icon and a foot-aligned character sprite may need different pivots.

For pixel art, test point filtering and crisp scaling. For animated characters, preview the entire sequence against a stationary reference so jumping pivots are obvious. An attractive sheet with inconsistent feet placement will look unstable in motion.

More complex content may need an AssetBundle or other asset-loading integration. Investigate the actual game's loading system and match Unity version and target platform before choosing that route. Asset extraction tools do not, by themselves, establish how to inject replacement data back into the running game.

11 Test the full lifecycle

Proposed test plan: keep a short log for each build. Record the exact setup, expected result, observed result and relevant error. Test with your mod alone before testing combinations.

- **Startup:** plugin loads once; missing assets produce a clear message; no duplicate registration.
- **Selection:** entry and text are correct; navigation works; changing selection clears old state.
- **Combat:** effect occurs at the intended rate and only for its owner; no infinite trigger chains.
- **Scaling:** upgrades and other modifiers behave consistently; damage calculations do not compound accidentally.
- **Lifecycle:** pause, death, victory, base return, restart and consecutive runs leave no stale effects.
- **Load:** test a copied save with the mod installed, updated and absent. Define the absent-mod behavior before relying on persistent new IDs.
- **Performance:** observe crowded scenes; avoid per-hit file reads, repeated texture creation, scene-wide searches every frame and unbounded logging.
- **Compatibility:** introduce other mods one at a time, especially those affecting the same selection, hit or stat paths.

For a prototype, prefer temporary run state until you have mapped persistence. For a mature content mod, specify how you handle unknown or removed content IDs. Silently mapping every unknown ID to a vanilla item can also lose user data; make the behavior deliberate and testable.

12 Troubleshoot by the earliest failing layer

The game fails before the title screen

Start with the loader alone. Check the package architecture, loader build and earliest error. If the unmodified game works but the loader alone does not, changing your gameplay code will not resolve that layer. Do not install a second loader on top as a diagnostic shortcut.

The game starts but your message is missing

Check the output DLL location, plugin metadata, framework compatibility and log. Confirm you copied the latest build and removed duplicate copies. A MelonLoader mod placed in BepInEx's plugin directory will not become a BepInEx plugin.

The message appears but your patch does not run

Check that patches are applied, that the method signature and overload match, and that gameplay reaches that method. Add a one-time observation log. A plausible name is not proof of the right hook.

A game update breaks previously working code

Keep the failure log. Revisit generated references and loader compatibility, compare changed method signatures, then rebuild. Follow the loader's supported regeneration procedure rather than deleting arbitrary folders. Record the new game version before declaring compatibility.

An image is invisible or appears briefly

Check file location, transparency, sprite rectangle, renderer assignment, scale and ordering. If it appears briefly, inspect animations or pooled-object initialization that might restore the old sprite. If other objects also change, you may be modifying shared asset state.

The effect gets stronger every frame or survives a restart

Look for repeated multiplication, duplicate patch registration, multiple event subscriptions, persistent static data or callbacks from an earlier run. Add clear initialization and cleanup points. Disable just the affected feature while investigating.

It works alone but conflicts with another mod

Identify the smallest pair that reproduces the problem. Check shared patch targets and ownership of values. Avoid a prefix that skips the entire original method unless it is essential and understood. Document the conflict and the versions tested.

13 Package something another person can install

Proposed release layout for a BepInEx mod:

```
BepInEx/
  plugins/
    EchoBall/
      EchoBall.dll
      Assets/
        echo-ball.png
README.txt
CHANGELOG.txt
LICENSE.txt
```

This is a packaging suggestion, not a built-in content folder recognized by the game. Your mod must load its assets explicitly. Include only dependencies that are necessary and distributable; do not bundle the game's assemblies or your entire development output directory.

The README should state the tested game build and store, loader build, install steps, configuration, what the mod changes, whether it replaces content or adds a separate entry, uninstall behavior and how to report a bug. If persistent content requires the mod to remain installed, explain that alongside installation instructions.

Keep your plugin identifier stable. Give each release a version and keep the source revision associated with it. Include credits for code and artwork you are permitted to reuse. Publishing source online does not automatically grant permission to redistribute it under any terms; inspect the repository's license before copying substantial code.

Ask bug reporters for the game version, loader version, mod version, reproduction steps and relevant log. Encourage removal of personal paths or other unrelated private information before public posting.

14 A realistic first project

Use these milestones instead of trying to build an entire character pack at once:

1. **Loader:** game starts and a compatible community mod works.
2. **Your DLL:** your own plugin writes one message.

3. **Observation:** identify and log one relevant gameplay event.
4. **One change:** alter one reversible behavior on existing content.
5. **One image:** show a simple original icon at the correct size.
6. **One ability:** implement the Echo Ball prototype with clear limits and cleanup.
7. **Content integration:** investigate and verify a separate identity, selection and persistence.
8. **Character integration:** add the character prototype, then a separate roster entry if supported by your discovered hooks.
9. **Release:** pass the lifecycle checks and package a small versioned build.

Do not let final artwork delay milestones 1–4. Conversely, do not mistake a pretty icon for solved gameplay integration. A successful first release can be one well-tested mechanic with very simple visuals.

What to collect for the next development session

Prepare your store and game version, installed loader build, a startup log, the generated assembly names, your desired mechanic in one paragraph and any discovered target signatures. For artwork, add the dimensions and frame requirements measured from a comparable asset. Those concrete inputs make it possible to replace the proposed parts of this guide with tested game-specific instructions.

15 Source directory

Initial sources were reviewed on 26 September 2026; the update sources and mod listings above were checked on 29 September 2026. Publication dates, listed versions and author reports are identified separately from local testing. Branch links and documentation can change; pin a commit or release when you use them for development. Public source inspection is not runtime compatibility testing.

- BALL x PIT official website — official game reference; this guide does not assume an official mod SDK.
- BALL x PIT community source repository — MelonLoader examples, shared helpers and project configuration.
- Auto Pickup installation and source — loader setup and an existing gameplay mod.
- Movement Speed source — concrete Player patch example.
- More Speed, Unlimited Balls and Passives and Local Coop — BepInEx requirements and author-reported implementation issues.
- MelonLoader — official loader documentation and release links.
- BALL x PIT Vortex Extension and extension review submission — optional mod-manager setup and author-stated supported stores.

- Unlimited Resources — resource-changing example with a loaded-save requirement; not custom-content registration.
- BepInEx IL2CPP installation, environment setup and plugin creation — primary tool instructions.
- BepInEx runtime patching, Harmony and HarmonyX — patch concepts and implementation references.
- BepInEx configuration tutorial — configuration binding and reading settings.
- Il2CppInterop class injection — injected type setup and limitations.
- dnSpyEx, UnityExplorer, AssetRipper and Cpp2IL — inspection and analysis tools, not verified BALL x PIT content frameworks.
- Unity image loading and sprite creation — general engine APIs, not game-specific integration points.
- Save-format research — community save documentation.
- Kenney and Piskel — accessible starting points for artwork.