Code Modification

Going beyond the basics and modifying the game's code.

Custom Actors

Sometimes the base game doesn't offer exactly what you need for your level. That's when you should turn to custom actors!

> This guide provides a basic overview of how to create custom actors. A decent understanding of C++ along with some prior NSMB code modding knowledge are recommended for this tutorial.

Section 1 - The Boiler Plate

This is code you will be writing basically every time you go to create a custom actor. It is standard practice to split your actor into two files `myActor.cpp` and `myActor.hpp`

`myActor.cpp`

```
#include "nsmb.hpp"
#include "myActor.hpp"

ncp_over(0x020c560c, 0) const ObjectInfo objectInfo = MyActor::objectInfo; //Stage Object ID 44 (use this in the editor)
ncp_over(0x02039a34) static constexpr const ActorProfile* profile = &MyActor::profile; //objectID 46

s32 MyActor::onCreate(){
	return true;
}

bool MyActor::loadResources() {
    return true;
}

s32 MyActor::onDestroy(){
	return true;
}

s32 MyActor::onRender(){
	return true;
```

```
}

bool MyActor::updateMain(){
	return true;
}
```

This file is where you program your actor.

At the beginning of the file is where you define:

- Where in the object bank table your actor's object info is stored. The array index is what you place in the editor to "place" your object. In the sample above, it is replacing ID 44. The beginning of this array is located at `0x020c529c`
- Where in the main process table your actor's profile is stored. The beginning of this array is located at `0x0203997c`

Here's an overview of what these functions are usually used for:

Function	Purpose
`onCreate()`	This function is ran when your actor is first created. This function is used for initalizing variables and anything that should be done once.
`loadResources()`	This function is called when the game is setting up your actor. [Todo]: Find the normal use case for this function.
`onDestroy()`	This function is called when your Actor is being removed from the stage. This function is used for cleaning up after the actor (i.e. freeing resources).
`onRender()`	This function is called every game tick while your actor is in view of the camera. This is typically used for graphics related code (for example, changing what texture an NSBTX is currently drawing).
`updateMain()`	This function is called every game tick while your actor is loaded. This function is used for the main functionality of your code (i.e. calculating positions, updating variables, etc).

> TODO: Expand this table with more functions you can override in a `StageEntity`

`myActor.hpp`

```
#pragma once

#include "nsmb.hpp"
```

```
class MyActor: public StageEntity {
public:
    virtual s32 onCreate() override;

    static bool loadResources();

    virtual bool updateMain() override;

    virtual s32 onRender() override;

    virtual s32 onDestroy() override;

    static constexpr u16 objectID = 46;

    static constexpr ObjectInfo objectInfo = {
        0, 0,
        0, 0,
        0, 0,
        0, 0,
        CollisionSwitch::None,
    };

    static constexpr u16 updatePriority = objectID;
    static constexpr u16 renderPriority = objectID;
    static constexpr ActorProfile profile = {&constructObject<MyActor>, updatePriority,
renderPriority, loadResources};
};
```

This file is where you store your instance variables and declare your functions.

Section 2 - State Machines

While not all Actors *need* to have a state machine, it can often times greatly improve the readability and reliability of your code (state machines are also a strategy Nintendo used when writing actors for the game).

Defining the State Machine

In your `.hpp`, you need to add the following:

```
class MyActor: public StageEntity {
    /* ... */

    // Functions for the state machine. Add more as needed.
    void exampleState();
    void anotherExampleState();

    void (*updateFunc)(MyActor*);
s8 updateStep;

    void switchState(void (MyActor::*updateFunc)());

    /* ... */
}
```

In your `.cpp`, here's what you need:

```
s32 MyActor::onCreate(){
  /* ... */

  // Initalize the current state
  switchState(&MyActor::exampleState);

  /* ... */
}

bool MyActor::updateMain(){
/* ... */

    // Make sure to call the update function every tick
updateFunc(this);

/* ... */
}

bool MyActor::exampleState(){
    if (updateStep == Func::Init) {
        updateStep++;
        return;
    }
```

```
    if (updateStep == Func::Exit) {
        return;
    }
}

bool MyActor::anotherExampleState(){
    if (updateStep == Func::Init) {
        updateStep++;
        return;
    }

    if (updateStep == Func::Exit) {
        return;
    }
}

void MyActor::switchState(void (MyActor::*updateFunc)()) {
    auto updateFuncRaw = ptmf_cast(updateFunc);

    if (this->updateFunc != updateFuncRaw) {
        if (this->updateFunc) {
            this->updateStep = Func::Exit;
            this->updateFunc(this);
        }

        this->updateFunc = updateFuncRaw;

        this->updateStep = Func::Init;
        this->updateFunc(this);
    }
}
```

Using the State Machine

While the code may look intimidating, state machines are very intuitive once you start working with them.

Function/Variable	Purpose

`updateStep`	This variable keeps track of what update step the current state is in. 1 (or `Func::Init`) is used for when a state function is entered for the first time. Useful for setting variables related to the current state. -1 (or `Func::Exit`) is used for when a state function is being exited. Used for cleaning up any state specific code before the code changes to the next state. All other update step values can be used inside a step to create "sub steps" via conditional code.
`switchState()`	This function will swap the current state to the function passed as a parameter. Call this function when you want to change what state you are in.

> TODO: Expand this page with more actor components (for example, colliders)

Porting old patch syntax to NCPatcher

This page will help you understand how you can port any patching syntax to NCPatcher's.

You are searching through NSMBHD or NSMB Central and you find a shiny code patch, you rush and put it in the `source` folder of your project only to find that a code patch is not compatible with your code! That sucks.

So what can we do? For this example NSMB E3 Recreation's `PlayerAnims.cpp` code patch will be used.

Step 1 - Investigation

Let's start by taking a look at `PlayerAnims.cpp`.

```
#include <nsmb.hpp>
#include <nsmb/extra/fixedpoint.hpp>

#define NAKED __attribute__((naked))

// Slow down rotation speed
NAKED void repl_02114DFC_ov_0A() { asm("MOV R5, #0xC00\nBX LR"); }

// Walking transition delay
void repl_0211667C_ov_0A() {}

void repl_02116698_ov_0A(Player* player, int id, bool doBlend, Player::FrameMode frameMode,
fx32 speed, u16 frame) {
    // 3.75fx (0x3C00) is the max walk animation speed
    if (speed > (3.75fx / 2)) {
        speed = (3.75fx / 2);
    }

    if (player->animID == 2) {
        player->setBodyAnimationSpeed(speed);
```

```
    } else {
        if (player->animID == 1) {
            fx32 xvel = Math::abs(player->velocity.x);
            if (xvel >= 1.5fx) {
                player->setAnimation(2, doBlend, frameMode, speed, frame);
            } else {
                player->setBodyAnimationSpeed(speed);
            }
        } else {
            player->setAnimation(1, doBlend, frameMode, speed, frame);
        }
    }
}

// Force jump on anim 1
NAKED void nsub_02116A14_ov_0A() { asm("CMP R0, #1\nB 0x02116A18"); }

// Use anim 1
NAKED void repl_02116A2C_ov_0A() { asm("MOV R1, #1\nBX LR"); }
```

We must now wonder, what kind of a patch is this? Is this an `NSMBe` type patch or a `Fireflower` type patch?

By comparing common traits that each patcher uses we can guess what kind of patch type we are dealing with.

`NSMBe` type patches:

- Does not use attributes to declare patches, uses the function name. `void hook_x() {}`
- Patches always follow the format `<PATCH TYPE>_<ADDRESS HEX>_ov_<OVERLAY HEX>` or `<PATCH TYPE>_<ADDRESS HEX>` if you don't need to specify an overlay.
- `PATCH TYPE` can only be `hook`, `repl` or `nsub`.

`Fireflower` type patches:

- Uses attributes to declare patches. `hook(X) void func() {}`
- Patches always follow the format `<PATCH TYPE>(0x<ADDRESS HEX>, 0x<OVERLAY HEX>)` or `<PATCH TYPE>(0x<ADDRESS HEX>)` if you don't need to specify an overlay.
- `PATCH TYPE` can only be `hook`, `rlnk`, `safe` or `over`.

Did you guess correctly what kind of patch we are working with?

Click here to reveal the answer

NSMBe

Step 2 - Porting

This is a fairly simple process. Here is a list that shows the different patch syntax between the patchers:

NSMBe	Fireflower	NCPatcher
hook	safe	ncp_hook
repl	rlnk	ncp_call
nsub	hook	ncp_jump
	over	ncp_over
		ncp_repl

And here is an example comparing some of them:

```
// NSMBe
void hook_02000000() {} // doSomethingPatch
void repl_0200A000() {} // doUnspecifiedPatch
void repl_02010000_ov_0A() {} // doWhateverOverlayPatch
// over does not exist in NSMBe

// Fireflower
safe(0x02000000) void doSomethingPatch() {}
rlnk(0x0200A000) void doUnspecifiedPatch() {}
rlnk(0x02010000, 10) void doWhateverOverlayPatch() {}
over(0x02159348, 52) static int stupidVar = 0x0215CA6C;

// NCPatcher
ncp_hook(0x02000000) void doSomethingPatch() {}
ncp_call(0x0200A000) void doUnspecifiedPatch() {}
ncp_call(0x02010000, 10) void doWhateverOverlayPatch() {}
ncp_over(0x02159348, 52) static int stupidVar = 0x0215CA6C;
```

These addresses are ficticious and purely for demonstration!

An important thing to remember is that all values in `NSMBe` patches are always written in hexadecimal without `0x` prepended to them. In NCPatcher if you want to specify an hexadecimal value you need to prepend `0x`, otherwise the value will be interpreted as a decimal value!

Let's go back to `PlayerAnims.cpp` and try to apply these changes.

```
#include <nsmb.hpp>
#include <nsmb/extra/fixedpoint.hpp>

#define NAKED __attribute__((naked))

NAKED ncp_call(0x02114DFC, 10)
void slowDownRotationSpeed() { asm("MOV R5, #0xC00\nBX LR"); }

// Walking transition delay
ncp_call(0x0211667C, 10) void doNotJumpOnAnim2() {}

ncp_call(0x02116698, 10)
void customPlayerAnimator(Player* player, int id, bool doBlend, Player::FrameMode frameMode,
fx32 speed, u16 frame) {
    // 3.75fx (0x3C00) is the max walk animation speed
    if (speed > (3.75fx / 2)) {
        speed = (3.75fx / 2);
    }

    if (player->animID == 2) {
        player->setBodyAnimationSpeed(speed);
    } else {
        if (player->animID == 1) {
            fx32 xvel = Math::abs(player->velocity.x);
            if (xvel >= 1.5fx) {
                player->setAnimation(2, doBlend, frameMode, speed, frame);
            } else {
                player->setBodyAnimationSpeed(speed);
            }
        } else {
            player->setAnimation(1, doBlend, frameMode, speed, frame);
        }
    }
}
```

```
NAKED ncp_jump(0x02116A14, 10)
void forceJumpOnAnim1() { asm("CMP R0, #1\nB 0x02116A18"); }

NAKED ncp_call(0x02116A2C, 10)
void useAnim1() { asm("MOV R1, #1\nBX LR"); }
```

The code should now compile!

If your code still doesn't work because it complains about some functions not being defined or not existing then you might want to check this out as well: Porting old patches to the NSMB Code Reference

What if the patch was an assembly `.s` file instead of C `.c` or C++ `.cpp`? The process is the same.

```
hook_....:
    BX LR
```

Becomes

```
ncp_hook(...)
    BX LR
```

Step 4 - Tidying up

Even though the code should now be able to execute, it is still not in its optimal state. This part is slightly more complicated because it requires understanding the code.

NCPatcher includes its own definition of `__attribute__((naked))` which is `ncp_asmfunc` so we remove that macro definition and use `ncp_asmfunc` instead.

```
#include <nsmb.hpp>
#include <nsmb/extra/fixedpoint.hpp>

ncp_asmfunc ncp_call(0x02114DFC, 10)
void slowDownRotationSpeed() { asm("MOV R5, #0xC00\nBX LR"); }

// Walking transition delay
ncp_call(0x0211667C, 10) void doNotJumpOnAnim2() {}

ncp_call(0x02116698, 10)
```

```
void customPlayerAnimator(Player* player, int id, bool doBlend, Player::FrameMode frameMode,
fx32 speed, u16 frame) {
    // 3.75fx (0x3C00) is the max walk animation speed
    if (speed > (3.75fx / 2)) {
        speed = (3.75fx / 2);
    }

    if (player->animID == 2) {
        player->setBodyAnimationSpeed(speed);
    } else {
        if (player->animID == 1) {
            fx32 xvel = Math::abs(player->velocity.x);
            if (xvel >= 1.5fx) {
                player->setAnimation(2, doBlend, frameMode, speed, frame);
            } else {
                player->setBodyAnimationSpeed(speed);
            }
        } else {
            player->setAnimation(1, doBlend, frameMode, speed, frame);
        }
    }
}

ncp_asmfunc ncp_jump(0x02116A14, 10)
void forceJumpOnAnim1() { asm("CMP R0, #1\nB 0x02116A18"); }

ncp_asmfunc ncp_call(0x02116A2C, 10)
void useAnim1() { asm("MOV R1, #1\nBX LR"); }
```

Now, take a look at the original purpose of `repl_0211667C_ov_0A` (now named `doNotJumpOnAnim2`) and the code it targeted.

```
ov10:02116678   CMP R0, #2
ov10:0211667C   BEQ 0x021166A0
ov10:02116680   MOV R0, R5
```

We can see that what we are doing is the following:

```
ov10:02116678    CMP R0, #2
ov10:0211667C    BL  repl_0211667C_ov_0A
```

```
ov10:02116680    MOV R0, R5
//...
repl_0211667C_ov_0A:
    BX LR // return generated by the compiler
```

Essentially we are just making it so `BEQ 0x021166A0` will never jump to `0x021166A0`, but we are not doing this efficiently because we jump from `0x0211667C` to `repl_0211667C_ov_0A` and then back to `0x02116680` instead of just continuing. This wastes memory and CPU cycles, but it was the only way of doing so in `NSMBe`. Instead we can write it like `ncp_repl(0x0211667C, 10, "NOP")` in NCPatcher, making the instruction do nothing and just skip to the next one without using any more memory.

```
ov10:02116678    CMP R0, #2
ov10:0211667C    NOP        // Skips to the next instruction
ov10:02116680    MOV R0, R5
```

After evaluating all theses different cases, our optimal code should look like this:

```
#include <nsmb.hpp>
#include <nsmb/extra/fixedpoint.hpp>

// Slow down rotation speed
ncp_repl(0x02114DFC, 10, "MOV R5, #0xC00")

// Walking transition delay
ncp_repl(0x0211667C, 10, "NOP")

ncp_call(0x02116698, 10)
void customPlayerAnimator(Player* player, int id, bool doBlend, Player::FrameMode frameMode,
fx32 speed, u16 frame) {
□// 3.75fx (0x3C00) is the max walk animation speed
□if (speed > (3.75fx / 2)) {
□□speed = (3.75fx / 2);
□}

□if (player->animID == 2) {
□□player->setBodyAnimationSpeed(speed);
□} else {
□□if (player->animID == 1) {
□□□fx32 xvel = Math::abs(player->velocity.x);
□□□if (xvel >= 1.5fx) {
□□□□player->setAnimation(2, doBlend, frameMode, speed, frame);
```

```
□□□} else {
□□□□player->setBodyAnimationSpeed(speed);
□□□}
□□} else {
□□□player->setAnimation(1, doBlend, frameMode, speed, frame);
□□}
□}
}

// Force jump on anim 1
ncp_repl(0x02116A14, 10, "CMP R0, #1")

// Use anim 1
ncp_repl(0x02116A2C, 10, "MOV R1, #1")
```

Porting old patches to the NSMB Code Reference

Setting Up Code Modifications

So, you're ready to dive into the code of the game? Let's get started!

> In this tutorial, you will learn how to:

- Set up the NSMB DS Code Template
- Set up ARM GCC
- Set up NCPatcher
- Extract/Build your ROM

> This guide will cover the "NCPatcher Standalone" method described in the code template as the steps are more synchronized between all operating systems.

Setting Up the Code Template

1. Head over to the code template's GitHub
2. Click on **Code -> Download ZIP**
3. Now extract this zip and rename the folder to what you want to call your project (this will be referred to as your **project root**)
4. > Don't put any spaces in your folder name!

> You have now set up the code template

Setting up ARM GCC

1. Head to the Arm GNU Toolchain Download Page
2. Now search (using CTRL/CMD + F) for **AArch32 bare-metal target (arm-none-eabi)** and download the correct installer for your operating system.
3. Open the installer and install the toolchain.
4. > Pick a location without spaces to install the toolchain!

> You have now set up ARM GCC

Setting Up NCPatcher

1. Head over to the NCPatcher GitHub releases page

2. Download the latest release for your operating system
3. Extract NCPatcher

Now, NCPatcher depends on **ncpatcher.json**, so lets make it!

> If you'd like to learn more about this file, head over to the NCPatcher GitHub!

In your **project root**, create the following files:

- **ncpatcher.json**
  - Copy/Paste the following JSON into the file

```
{
  "$arm_flags": "-masm-syntax-unified -mno-unaligned-access -mfloat-abi=soft -mabi=aapcs",
  "$c_flags": "-Os -fomit-frame-pointer -ffast-math -fno-builtin -nostdlib -nodefaultlibs -
nostartfiles -DSDK_GCC -DSDK_FINALROM",
  "$cpp_flags": "-fno-rtti -fno-exceptions -std=c++20",
  "$asm_flags": "-Os -x assembler-with-cpp -fomit-frame-pointer",
  "$ld_flags": "-lgcc -lc -lstdc++ --use-blx",

  "backup": "backup",
  "filesystem": "nsmb",
  "toolchain": "arm-none-eabi-",

  "arm7": {},
  "arm9": {
  "target": "arm9.json",
  "build": "build"
  },

  "pre-build": [],
  "post-build": [],

  "thread-count": 0
}
```

> You have now set up NCPatcher

Extracting, Building, and Repackaging Your ROM

If you're on Windows

1. Download fireflower.zip and extract it.
2. Move **nds-build.exe** and **nds-extract.exe** out from the folder

If you're on macOS/Linux

1. Download nds-extract.zip
2. Download nds-build.zip
3. Extract both ZIPs

Now, you need to build the tools.

For NDS Extract:

1. Open a new **Terminal** window in the folder of the code
2. Run this command: `g++ nds-extract.cpp -o nds-extract -std=c++20`

For NDS Build:

1. Open a new **Terminal** window in the folder of the code
2. Run this command: `g++ nds-build.cpp -o nds-build -std=c++20`

> From here on, the instructions will work for all operating systems. If you are on Windows 10, you can use **Command Prompt** instead of **Terminal**

Extracting Your ROM

1. Open a **Terminal** window in your **project root**
2. Run this command: `/path/to/nds-extract rom.nds nsmb`

> Replace /path/to/ with the actual file path to nds-extract. Also replace "rom" with the actual name of your .nds file

This will extract the contents of your ROM into a folder named nsmb

> You have extracted your ROM

Building Your ROM

This step will compile and patch your ROM with any code files found in the **source** directory in your **project root**. The Code Template comes with a few examples included in the **source** directory.

1. Open a **Terminal** window in your **project root**
2. Run this command: `/path/to/ncpatcher`

> Replace /path/to/ with the actual file path to ncpatcher

> You have built your ROM

Repackaging Your ROM

nds-extract depends on **buildrules.txt**, so let's create it!

- **buildrules.txt**
  - Copy/Paste the following text into the file:

```
rom_header nsmb/header.bin
arm9_entry KEEP
arm9_load KEEP
arm7_entry KEEP
arm7_load KEEP
fnt nsmb/fnt.bin
file_mode ADJUST
arm9 nsmb/arm9.bin
arm7 nsmb/arm7.bin
arm9ovt nsmb/arm9ovt.bin
arm7ovt nsmb/arm7ovt.bin
icon nsmb/banner.bin
rsa_sig nsmb/rsasig.bin
data nsmb/root
ovt_repl_flag 0xFF
ov9 nsmb/overlay9
ov7 nsmb/overlay7
```

This step will take the files form the **nsmb** folder and repackage them into a .nds file

1. Open a **Terminal** window in your **project root**
2. Run this command: `/path/to/nds-build buildrules.txt NSMB.nds`

> Replace /path/to/ with the actual file path to nds-build.

> You have repackaged your ROM

Using GDB with Ghidra and melonDS

What you'll need:

- The latest version of Ghidra
- A build of melonDS that has the GDB enabled
- > The easiest way to get this is to grab a GitHub action build of melonDS. You can find that here. (Note: you'll need to be signed into a GitHub account to download these builds)
- The GNU ARM Embedded Toolchain installed on your system
- A Ghidra database of NSMB DS
- > Eventually, NSMB Central will host a shared Ghidra project so we have one centralized project anyone can contribute to. For now, you can generate a Ghidra project using this tool. If you need help, please ask in our Discord!

Configuring melonDS

To enable the GDB, you need to do the following:

1. Click on the **Config** menu at the top of the emulator, then click on **Emu Settings**
2. Click on the **Devtools** tab
3. Check Enable **GDB stub**
4. > If you do not see the Devtools tab, then you have not built melonDS with GDB enabled. Please check the link at the start of the guide to find a download with GDB enabled or build it yourself enabling GDB in CMake

> melonDS is ready to go!

Setting up Ghidra

To begin, open your Ghidra project in the code viewer as you normally would.

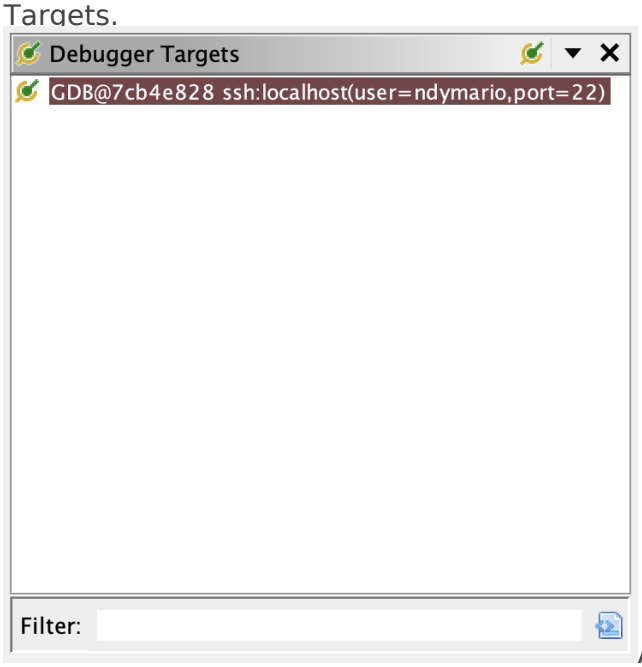
1. Click on File -> Configure, which should open a list of tools
2. Check the "Debugger" box

This should cause windows to appear in your current project, likely making the following steps redundant. If you are unable to find a window, the following steps will either open the window, or present it to you in the project.

Creating a Debugger Target

> This method has been tested on Linux and macOS. You should be able to follow these steps using WSL on Windows. Follow this guide if you need help setting up WSL.

To begin, open the Debugger Targets window by navigating to Windows -> Debugger -> Debugger Targets.



As you'll notice, there is an active connection in the screenshot but nothing on your end...let's fix that!

Click on the  button to open the **connect** window.

If you are on Linux:

- Choose **gdb** in the dropdown
- Set **arm-none-eabi-gdb -ex "set arch armv5t"** as the **GDB Launch Command**
- > If you have not added arm-none-eabi-gdb to your PATH, you'll need to provide the absolute path
- Click 

If you are on macOS

- Choose **gdb via SSH** in the dropdown
- Set **arm-none-eabi-gdb -ex "set arch armv5t"** as the **GDB Launch Command**
- > If you have not added arm-none-eabi-gdb to your PATH, you'll need to provide the absolute path
- Set **SSH hostname** to **localhost**
- Set **SSH username** to your username
- > You can use the command **whoami** in the terminal to get your username if you don't know it
- Click 

If you are on Windows

- > This still needs to be tested on Windows. This guide will be updated when steps have been made

> You have now created a Debugger Target

Connecting to melonDS

The **gdb interpreter** should have opened for you when you connected to the **debugging target.**

- > If you have lost the interpreter window, open the objects window (Window -> Debugger -> Objects) and click on  to bring the menu back
- In melonDS, open your ROM. (You can either boot directly to the game or launch the firmware)

Now, in the interpreter menu, run the command **target remote localhost:[ARM9 Port]** (Where [ARM9 Port] is the ARM9 Port set in the Devtools tab.)

- > By default, it should be 3333. The command would be **target remote localhost:3333**

If melonDS pauses after running this command, GDB is now talking to melonDS

- > If the connection immediately closes after running the command: change the ARM9 port to something else, restart melonDS, and close the current GDB connection.

>

> You have now connected Ghidra to melonDS

Using Breakpoints

If you would like to set breakpoints, you'll need to use the **Dynamic PC**

1. Open the **Dynamic PC** window by clicking **Window -> Listing -> Dynamic - Auto PC,** [...]
    1. > If you do not see this option, you can alternatively open it via **Window -> Debugger -> New Dynamic Listing**
2. Next, open the **Modules** window by clicking **Window -> Debugger -> Modules**
3. Lastly, click on  in the **Modules** window.

Now, setting a breakpoint in your code view should set a breakpoint in the **Dynamic PC**

- > Breakpoints will only update if the emulator has hit a breakpoint or has been paused by pressing 

> You have now set up Ghidra to debug melonDS. Happy coding!

Public Code Mods

This is a collection of locations where you can find public code mods for your own NSMB mod. The header of each section links to the relevant server/board if you want to look around for anything scattered around those posts. If you find another location, feel free to add it to this list. All code here is for use with NCPatcher and MammaMia Team's code template unless otherwise specified.

NSMB Central:

- #nsmb-requests
- #public-resources

GitHub Repositories:

- https://github.com/NSMBC/Code-Mods
- https://github.com/pete420griff/nsmb-stuff
- https://github.com/illythecat/New-PlusPlus-Super-Mario-Bros
- https://github.com/mariomadproductions/nsmb-e3-rec (may be unpolished)
- https://github.com/Newer-Team/NewerSMBDS (NSMBe patcher)

NSMBHD:

- https://nsmbhd.net/thread/2569-misc-patches-thread/ (NSMBe patcher; Dirbaio's template)

NSMB Co-op Development Guide

Introduction

This guide explains how to write code that works correctly with the NSMB Co-op hack. The co-op system allows two players (Mario and Luigi) to play simultaneously on separate consoles connected via local wireless.

The biggest challenge in co-op development is preventing **desyncs** - situations where the two consoles have different game states. This document covers common desync patterns and how to avoid them, along with co-op-specific systems like player spectating.

> **Key Principle:** Any code that affects gameplay state must produce identical results on both consoles, regardless of which console is running it.

Table of Contents

Core Concepts

What is a Desync?

A desync occurs when the two consoles have different game states. For example:

- Console 1 thinks Mario has 3 lives, Console 2 thinks Mario has 2 lives
- Console 1 shows an enemy as alive, Console 2 shows it as defeated
- Console 1 has a different random number sequence than Console 2

Safe vs. Unsafe Operations

- **Safe**: Operations that only affect local display/audio (sounds, UI elements, screen shaking)
- **Unsafe**: Operations that modify gameplay state (player health, enemy behavior, item spawning, particle effects)

The Golden Rule

When writing gameplay logic, always consider: "What happens if both consoles run this code at the same time?"

Common Anti-Patterns to Avoid

Before diving into specific solutions, here are the most common mistakes that cause desyncs:

? DON'T: Use Game::localPlayerID for gameplay logic

```
// This will desync!
if (shouldTriggerEvent()) {
    Player* player = Game::getPlayer(Game::localPlayerID);
    player->giveReward();
}
```

? DO: Loop through all players or use linkedPlayerID

```
// This stays in sync!
if (shouldTriggerEvent()) {
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        Player* player = Game::getPlayer(playerID);
        if (playerMeetsCondition(player)) {
```

```
            player->giveReward();
        }
    }
}
```

? DON'T: Use ViewShaker without playerID parameter

```
// This will desync!
ViewShaker::start(type, viewID);
```

? DO: Specify which player should feel the shake

```
// This stays in sync!
ViewShaker::start(type, viewID, playerID, false);
```

? DON'T: Use Game::getRandom() for gameplay logic

```
// This will desync!
if ((Game::getRandom() & 0xFF) == 0) {
    spawnEnemy();
}
```

? DO: Use Net::getRandom() for synchronized randomness

```
// This stays in sync!
if ((Net::getRandom() & 0xFF) == 0) {
    spawnEnemy();
}
```

Understanding Desyncs: A Detailed Example

Let's examine how a typical desync occurs using a Goomba collision example:

Let's pretend this is how a Goomba is coded to hurt a player:

```
void Goomba::hurtPlayer() {
    // Game::localPlayerID is the ID of the player for *our* console
    s32 playerID = Game::localPlayerID;

    // Game::getPlayer(id) gives us a pointer to a Player object
    // id = 0 → Mario
    // id = 1 → Luigi
    Player* player = Game::getPlayer(playerID);

    // The local player gets hurt
    player->getHurt();

    // Problem:
    // On Console 0 → local_player_id = 0 → only Mario gets hurt
    // On Console 1 → local_player_id = 1 → only Luigi gets hurt
    //
    // Bad result:
    // Console 0 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
    //
    // Console 1 sees:
    //   Mario = NOT HURT
    //   Luigi = HURT
}
```

The Solution

The fix is to use the collision information that's already available:

Usually when a collision with an enemy occurs, the actor is informed of which player collided with it. This information is stored in `this->linkedPlayerID`.

```
void Goomba::hurtPlayer() {
    // this->linkedPlayerID is the ID of the player that collided with the Goomba
    // Let's assume it was Mario (0)
    s32 playerID = this->linkedPlayerID;

    // Game::getPlayer(id) gives us a pointer to a Player object
    // id = 0 → Mario
    // id = 1 → Luigi
    Player* player = Game::getPlayer(playerID);

    // The player that collided with the Goomba gets hurt
    player->getHurt();

    // Good result:
    // Console 0 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
    //
    // Console 1 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
}
```

Alternative: Player Loop Pattern

When `this->linkedPlayerID` isn't available, use the player loop pattern:

```
bool Goomba::shouldPlayerGetHurt(Player* player) {
    // ... do any checks to decide if Player should get hurt

    // In this example we assume Mario (0) is in love with the Goomba
    return player->isInLoveWithGoomba(this);
}

void Goomba::hurtPlayer() {
    // Update the logic for all players
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        // Game::getPlayer(id) gives us a pointer to a Player object
        // id = 0 → Mario
```

```
        // id = 1 → Luigi
        Player* player = Game::getPlayer(playerID);

        if (shouldPlayerGetHurt(player)) {
            // The player that collided with the Goomba gets hurt
            player->getHurt();
        }
    }

    // Good result:
    // Console 0 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
    //
    // Console 1 sees:
    //   Mario = HURT
    //   Luigi = NOT HURT
}
```

Co-op-Safe Patterns and Solutions

Player Targeting: Finding the Right Player

Use `ActorFixes_getClosestPlayer(this)` instead of `Game::getLocalPlayer()` or `Game::getPlayer(Game::localPlayerID)`.

```
void Volcano::spawnMeteor() {
    // BAD: Always targets the local player
    Player* target = Game::getLocalPlayer();

    // GOOD: Finds the closest player to the volcano
    Player* target = ActorFixes_getClosestPlayer(this);

    // Spawn meteor at target's position
    Vec3 meteorPos = target->position;
    Actor::spawnActor(METEOR_ID, 0, &meteorPos, nullptr, nullptr, nullptr);
}
```

For zone-specific targeting:

```
void SpikeBass::attack() {
    // Find the closest player in a specific zone
    Player* target = ActorFixes_getClosestPlayerInZone(this, zoneID);
    if (target == nullptr) {
        // Fallback to any closest player
        target = ActorFixes_getClosestPlayer(this);
    }

    // Attack the target
    fireProjectileAt(target->position);
}
```

Audio: Console-Specific Sound Effects

> This is one of the few cases where it's safe to use `Game::localPlayerID`, as audio is local to each console - the other console doesn't receive or process your sound effects.

```
void MyHack::updatePlayerFlyState() {
    // Update the logic for all players
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        Player* player = Game::getPlayer(playerID);

        // ... logic to update the fly state

        // Play flight finished jingle
        if (player->finishedFlying) {
            // Only the player that finished flying will hear the jingle
            if (playerID == Game::localPlayerID) {
                SND::playSFX(FLIGHT_FINISHED_SFX, &player->position);
            }
        }
    }
}
```

Another common pattern is to play sound effects when items are collected or power-ups are switched:

```
void RedRing::spawnReward() {
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        Player* player = Game::getPlayer(playerID);

        // Determine reward based on power-up
        PowerupState reward = calculateReward(player->currentPowerup);

        // Play sound only for the local player when they get Fire Flower
        if (reward == PowerupState::Fire && playerID == Game::localPlayerID) {
            SND::playSFX(0x17E, &this->position);
        }

        // Spawn the item for this player
        spawnItemForPlayer(reward, playerID);
    }
}
```

Safe Uses of Game::localPlayerID

There are specific cases where using `Game::localPlayerID` is not only safe, but necessary:

1. **Sound Effects**: Audio is local to each console
2. **Visual UI Elements**: Screen-specific UI components like menus and HUD
3. **Liquid Position**: Due to co-op forcing shared areas, liquid levels are stored per-console
4. **File Loading**: Loading graphics or UI resources that are console-specific

```
// □ SAFE: Sound effects
if (playerID == Game::localPlayerID) {
    SND::playSFX(soundID, &position);
}

// □ SAFE: Liquid collision (special case - see liquid section)
if (player->position.y < Stage::liquidPosition[Game::localPlayerID]) {
    // Handle liquid damage
}

// □ SAFE: UI updates
if (playerID == Game::localPlayerID) {
    FS::loadFileLZ77(spectateTextFileID, (u16*)HW_OBJ_VRAM);
}
```

View Shaking: Per-Player Screen Effects

If you want to shake the screen for a specific player, never use the basic `ViewShaker::start` overloads with conditional logic:

```
// BAD: This causes instant desync
if (canShakePlayer(Game::localPlayerID)) {
    ViewShaker::start(type, viewID);
}
```

This causes an immediate desync. Instead, use the 4-argument overload without the conditional check:

```
void SledgeBro::doGroundPound() {
    for (s32 playerID = 0; playerID < Game::getPlayerCount(); playerID++) {
        Player* player = Game::getPlayer(playerID);

        // Check if this specific player should be affected
        if (ActorFixes_isPlayerInShakeRange(player)) {
            ViewShaker::start(3, this->viewID, playerID, false);

            // Play sound only for the local player
            if (playerID == Game::localPlayerID) {
                SND::playSFX(138, &this->position);
            }

            // Apply gameplay effects to this specific player
            if (!Game::getPlayerDead(playerID)) {
                player->takeDamage();
            }
        }
    }
}
```

Camera and Visibility Checks

Never use `Game::isOutsideCamera(..., Game::localPlayerID)` for gameplay logic.
Use `ActorFixes_isOutsideCamera` or `ActorFixes_isInRangeOfAllPlayers` instead:

```
void Enemy::updateBehavior() {
    // BAD: Only checks against local player's camera
    if (Game::isOutsideCamera(this->position, boundingBox, Game::localPlayerID)) {
        return; // Skip update
    }

    // GOOD: Checks against the closest player's camera
    if (ActorFixes_isOutsideCamera(this, boundingBox)) {
        return; // Skip update
    }

    // Continue with enemy logic...
}
```

For entities that need to stay active when any player can see them:

```
void Enemy::onUpdate() {
    // This ensures the actor only updates if ANY player can see it
    if (!ActorFixes_isInRangeOfAllPlayers(this)) {
        return; // All players are too far away, skip update
    }

    // Continue updating since at least one player can see us
    updateLogic();
}
```

Rendering Optimization

Use `ActorFixes_safeSkipRender` for 3D animated entities that need to update their models but may not render:

```
class HammerBro : public StageEntity3DAnm {
    bool skipRender() override {
        // This will update the model but only render for players who can see it
        return ActorFixes_safeSkipRender(this);
    }
};
```

Random Number Generation

Use `Game::getRandom()` for local code (UI, effects, sounds).
Use `Net::getRandom()` for gameplay logic that affects game state:

```
void Blockhopper::updateJump() {
    // BAD: Different random numbers on each console = desync
    if ((Game::getRandom() & 0xFF) == 0) {
        doJump();
    }

    // GOOD: Synchronized random numbers across consoles
    if ((Net::getRandom() & 0xFF) == 0) {
        doJump();
    }
}
```

Special Cases

Liquid/Lava Damage: The Exception to the Rule

> **Special Case:** Liquids are one of the few exceptions where you **DO** use `Game::localPlayerID`!

This is because the co-op implementation doesn't support per-player liquid levels - if liquid is detected in the level, both players are forced to always be in the same area, so they share the same liquid level. The liquid position is managed per-console, not per-player.

```
void checkLiquidDeath(Player* player) {
    s32 playerID = player->linkedPlayerID;

    // CORRECT: Use localPlayerID for liquid position
    // Both players share the same liquid level since they're in the same area
    if (player->position.y < Stage::liquidPosition[Game::localPlayerID]) {
        player->playSFXUnique(338, &player->position);
        Liquid_doWaves(player->position.x, 1);
        Game::losePlayerLife(playerID);
        Game::setPlayerDead(playerID, true);
    }
}
```

The reason for this exception:

- Liquid positions are stored per-console: `Stage::liquidPosition[Game::localPlayerID]`
- Co-op forces both players to be in the same area at all times
- Each console only tracks one liquid level (the local one)
- Trying to use `Stage::liquidPosition[playerID]` would fail because only index `[Game::localPlayerID]` is valid

StageLayout Data: Incomplete Arrays

> **Critical Issue:** Many parts of the StageLayout have arrays that appear to support both players (`PlayerCount`-sized arrays), but in reality **only the local player's data is populated or valid**.

This means that even though the StageLayout structure contains arrays like:

```
ScreenInfo screenFG[PlayerCount];     // Only [localPlayerID] is valid
ScreenInfo screenBG[PlayerCount];     // Only [localPlayerID] is valid
```

> **You MUST use `Game::localPlayerID` when accessing these arrays**, regardless of which player you're working with:

```
// □ BAD: Will access invalid/empty data for non-local player
u16 getForegroundID(u32 playerID) {
    return Stage_getFgScreenID(playerID); // This will fail for playerID != localPlayerID
}

// □ GOOD: Always use localPlayerID for StageLayout data
u16 getForegroundID(u32 playerID) {
    return Stage_getFgScreenID(Game::localPlayerID); // This works correctly
}
```

Real Examples from the Codebase:

1. **Pipes Background Fix**: The screen foreground data is only available for the local player:

   ```
   // Fixed pipes background - must use localPlayerID instead of playerID
   // Original code caused desyncs by trying to access screenFG[playerID]
   // when only screenFG[localPlayerID] contains valid data
   ```

2. **Volcano Eruption**: Background screen data is local-only:

```
// Check if we're in a volcano level (screen ID 15)
if (Stage_getFgScreenID(Game::localPlayerID) == 15) {
    ActorFixes_updateVolcanoBackground();
}
```

3. **BG1CNT Register Fix**: Background control data is per-console:

```
// Do not set BG1 CNT with other player's data - we don't have it!
if (Game::localPlayerID == playerID) {
    // Apply background changes only for local player
}
```

Why This Happens:

- Nintendo's original game was single-player, so many systems only tracked one set of data
- Background/foreground rendering data is managed per-console since each console renders independently
- Screen effects, camera settings, and background animations are local to each console

When to Use localPlayerID for StageLayout:

- Screen/background data access (`screenFG`, `screenBG`, `screenTS`)
- Camera and view-related information that's console-specific
- Background effects and animations
- Tile rendering and display settings

Rotators: Forced View Synchronization

> **Critical Issue:** Levels with **rotators** (rotating/tilting level mechanics) force additional constraints that override normal co-op behavior.

Rotators are detected by checking if any tileset has `screenID == 0xFF00`:

```
bool Stage_areaHasRotator() {
    u32 tilesetCount = Stage::getBlockElementCount(StageBlockID::Tileset);
    for (u32 i = 0; i < tilesetCount; i++) {
        if (Stage::stageBlocks.tileset[i].screenID == 0xFF00)
            return true;
    }
    return false;
}
```

Rotator Restrictions:

When a level has a rotator, **both players must always be in the same view**.

This limitation exists because each console only maintains its own StageLayout data, making it impossible to reliably detect or synchronize rotator states for the other player. Supporting independent rotators for both players would require a major overhaul of the system, so co-op forces both players to remain in the same view to guarantee that all rotator effects apply to a shared, consistent state.

Consequences of Rotators:

- Players cannot split into different areas/views
- Door and pipe transitions affect both players simultaneously

Examples of Rotator Levels:

- W8 Final Castle

Hardcoded Area Values

> **Important:** The codebase contains many hardcoded area number checks for specific levels and special behaviors. These represent special-case handling that you need to be aware of when modifying co-op behavior. There are plans to remove these hardcoded constraints by using actors that set flags on the levels and then despawn.

Mini-Mushroom Cutscene Areas (180, 181):

```
// Mini-mushroom cutscene areas
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 180 || areaNum == 181) {
    PlayerSpectate::clearSpectators();
    *rcast<u32*>(0x02085ACC) |= 0x20; // toadHouseFlag
    *rcast<u32*>(0x020CA8B4) = 0x1000; // timeLeft

    // Special handling for mini-mushroom collection areas
    if (itemType == 25 && player->currentPowerup == PowerupState::Mini) {
        Stage::exitLevel(1); // Exit to mini world
        return;
    }
}
```

Boss Defeat Cutscenes:

```
// World 2 Boss (42) and World 5 Boss (105)
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 42) { // World 2
    switchToCutsceneArea(0);
} else if (areaNum == 105) { // World 5
    switchToCutsceneArea(1);
}
```

Final Castle Special Handling (173):

```
// Hardcoded: prevent rotators from resetting but still get rid of lava in W8 Final Castle
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 173 && Game::getPlayer(0)->viewID != 0) {
    Stage_forceAreaReload = 2;
}
```

Boss Arena Loading (19, 175):

```
// Skip loading castle models for specific boss areas
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 19 || areaNum == 175) {
    return; // Don't load the model
}
```

Shared Camera Mode (174):

```
// Force shared camera for specific boss fights
u32& areaNum = *rcast<u32*>(0x02085A94);
if (areaNum == 174) {
    PlayerSpectate::sharedCamera = true;
}
```

Common Hardcoded Values:

- **19, 175**: Boss arenas requiring special model handling
- **42**: World 2 Boss (Mummy Pokey)
- **105**: World 5 Boss (Petey Piranha)
- **173**: World 8 Final Castle (rotator + lava)
- **174**: Specific boss fight with shared camera
- **180, 181**: Mini-mushroom cutscene areas

Why These Exist:

- **Memory Management**: Some boss fights need special model loading/unloading
- **Transition Logic**: Areas that exit to special worlds or cutscene areas
- **Camera/View Constraints**: Levels with unique mechanics (rotators, looper)

When Working with New Areas:

1. Check if your area number conflicts with existing hardcoded values
2. Consider whether your area needs special co-op handling
3. Add your own hardcoded checks if needed for area-specific behavior

Debugging and Troubleshooting

Desync Detection System

The codebase includes a `DesyncGuard` system that helps detect when the game state diverges between consoles. Key events that are monitored include:

- Player damage events
- Power-up changes
- Scene transitions
- RNG usage

If you're adding new gameplay systems, consider adding desync check markers at critical points:

```
void myGameplayFunction() {
    // Your gameplay logic here

    // Mark that this function was called to detect desyncs
    DesyncGuard::markDesyncCheck();
}
```

General Debugging Tips

> Remember: When in doubt, loop through all players and apply logic based on each player's individual state rather than assuming anything about the local player!

Key Questions to Ask:

- Does this code behave differently on Console 0 vs Console 1?
- Am I using `Game::localPlayerID` for gameplay logic?

- Are my random numbers synchronized between consoles?
- Will both consoles execute this logic identically?

Advanced Systems

Player Spectate System

The co-op hack includes a spectate system that allows dead players to watch the other player and automatically follow them through level transitions. This system maintains engaging co-op gameplay when one player dies, rather than forcing a restart or breaking the co-op experience.

How Spectating Works

When a player dies in co-op mode, instead of immediately respawning or ending the level, they enter **spectate mode**:

1. **Target Assignment**: The dead player's camera follows the living player

```
// playerID ^ 1 gives us the other player (0 becomes 1, 1 becomes 0)
PlayerSpectate::setTarget(deadPlayerID, deadPlayerID ^ 1);
```

2. **Camera Following**: The spectating player's camera smoothly lerps to follow their target

```
// Camera position updates to match the target player
Player* target = PlayerSpectate::getTargetPlayer(spectatorPlayerID);
target->followCamera(spectatorPlayerID);
```

3. **View Transitions**: When the living player enters doors/pipes, spectators automatically follow

```
// All spectators following transitPlayerID will switch views too
PlayerSpectate::syncSpectatorsOnViewTransition(transitPlayerID);
```

System Components

- **Target Tracking**: Each player has a target they're spectating (`playerTarget[playerID]`)
- **Local Target Cache**: Quick access to who the local player is watching (`localTarget`)
- **Smooth Transitions**: Camera lerping prevents jarring jumps when switching targets
- **Shared Camera Mode**: Special mode for certain levels (like the level looper in W8 Final Castle) where both players share one camera view
- **Entrance Following**: Spectators automatically transition to new areas when their target does

API Reference

```
// Check if a player is spectating someone else
bool PlayerSpectate::isSpectating(u32 playerID);

// Get who a player is currently spectating
Player* PlayerSpectate::getTargetPlayer(u32 playerID);

// Manually set spectate target
PlayerSpectate::setTarget(spectatorID, targetPlayerID);

// Enable smooth camera transitions
PlayerSpectate::setLerping(playerID, true);
```

Spectate Mode Triggers

Entering Spectate Mode:

- **Player Dies (Other Alive)**: Player dies and other player is still alive → Enter spectate mode
- **Level Start (No Lives)**: Player is dead when spawning (e.g., at level start with 0 lives)
- **Warp Cannon**: When using warp cannons, all other players spectate player 0 during the shooting sequence
- **Boss Victory Cutscenes**: Players not directly involved in the victory sequence spectate the player who triggered it
- **Flagpole**: (NOT IMPLEMENTED YET) When one player hits the flagpole, the other player spectates them during the victory sequence

Exiting Spectate Mode:

- **Player Respawns**: The player spawns back into the level → Respawned player returns to normal
- **New Level**: New level starts → Spectate targets reset to self (`PlayerSpectate::clearSpectators()`)
- **Cutscene End**: Boss introduction cutscenes end → Players return to normal

Camera Lerping System

The spectate system includes smooth camera transitions via **lerping** (linear interpolation):

Automatic Lerp Activation:

- **Boss Victory**: When players miss cutscenes or are repositioned during boss victories
- **Flagpole Hit**: (NOT IMPLEMENTED YET) Smooth transition when other player hits the flagpole

Automatic Lerp Deactivation:

The lerping system automatically stops itself when transitions complete:

```
// Position lerping stops when camera reaches close enough to target
if (distanceX < 48fx && distanceY < 48fx) {
    if (distanceX == 0 && distanceY == 0)
        playerLerping[playerID] = false; // Auto-disable
}

// Zoom lerping stops when zoom difference is eliminated
if (distance == 0)
    playerLerpingZoom[playerID] = false; // Auto-disable
```

Key Lerping Properties:

- **Smooth Movement**: Camera doesn't snap instantly to new positions
- **Distance Thresholds**: Lerping ends when within 48 units of target position
- **Separate Zoom**: Zoom lerping is handled independently from position lerping
- **Self-Terminating**: No manual intervention needed - lerping automatically stops when complete

The spectate system ensures that co-op gameplay remains engaging even when one player dies, allowing them to continue following the action and automatically rejoin when appropriate.

Summary

This guide covered the essential principles for writing co-op compatible code in NSMB:

Remember these key points:

- Never use `Game::localPlayerID` for gameplay logic (except liquids and local-only effects)
- Always loop through all players or use `this->linkedPlayerID` for collision-based logic
- Use `Net::getRandom()` for synchronized randomness, `Game::getRandom()` for local effects only
- Specify player IDs explicitly in functions like `ViewShaker::start()`
- Use co-op-safe helper functions like `ActorFixes_getClosestPlayer()`

When in doubt:

- Ask yourself: "Will this code behave identically on both consoles?"
- Test your changes with two consoles to verify synchronization
- Use the debugging tools and desync guards to catch issues early

>

> Following these patterns will help ensure your code works seamlessly in the co-op environment while maintaining the engaging two-player experience.