Project Misfits Tech Bible

Have questions or comments about anything? Feel free to ping me and I'll follow up. - Ben

Purpose

To provide a thorough, organized reference to important technical requirements, guidelines, and options for the project. This includes naming & formatting conventions, the file hierarchy, and the source control pipeline.

Goals

- Document the intentionality behind every technical decision
- Standardize methods to enable easy cross-collaboration
- Identify key technical opportunities or constraints which could be utilized or worked around

Table of Contents

Game Engine, Language, Technical Specs

This project uses Godot 4.5.1 and GDScript. Everyone MUST use Godot 4.5.1 and code using GDScript.

The game will be run at 60 frames per second, a 1080p resolution, and in a 16:9 screen aspect ratio.

Engine Plugins

- LimboAI - Provides behavior tree & state machine functionality
- Godot Dialogue Manager - Provides dialogue and cutscene features with a built-in editor and runtime
- Phantom Camera - provides common camera features, such as changing the view, changing camera parameters dynamically, camera follow modes, and camera triggers

Project Settings Setup

In the Godot project, some project settings must be changed to enforce some guidelines described in this document. This can be done within the Project Settings window.

For Static Typing:
Advanced Settings ON > Debug > GDScript
Set "Untyped Declaration" and "Inferred Declaration" to *Warn*.
Finally, go to the Editor tab > Editor Settings > Text Editor > Completion > Add Type Hints ON.

For Script Templates:
Editor > Script > Templates Search Path
Set to the path of the "script_templates" folder.

File Hierarchy

- References: Organizing files in Godot - rp.wtf, Game Project Architecture and Organization Advice for Godot 4.0+ - abmarnie & Godot 2D Platformer Demo - SlayHorizon
- Our file hierarchy largely follows the principle of locality of behavior, with folders designated for specific entities or groups which contain all relevant scripts, scenes, and other files.
- There are three top-level folders: "assets", "src", and "addons"
    - The "assets" folder contains all non-code files such as art and audio.
    - The "src" folder contains all code-related files like scripts, scenes, and entity database files.

    - The “addons” folder contains all files for Godot addons. These files may be read or copied but never modified.
- The folders for “assets” and “src” will be mostly the same, except for asset or code-exclusive files which need their own folders. This is by design to let asset creators import their files into the game safely without worrying about game code.
- Search-based navigation: If an asset or source code file is exclusively used by a single entity or entity group, prefix the file’s name with that entity/entity group (e.g. “player_jump.ogg” or “hud_heart.svg”). This makes it easy to search for exclusive files related to an entity.
- Inherited Scene Folders: Nest folders for inherited scenes within their base scene’s folder.
- Locally Shared Resources: Store shared resources for a specific "scene type" in a central folder named after that "scene type", with subfolders for each owning scene. Try to avoid excessive nesting.
- Globally Shared Resources: Place general resources used by many different "scene types" in a sibling folder named after that resource's data type. For example, put all globally used .shader files in a shaders/ folder.

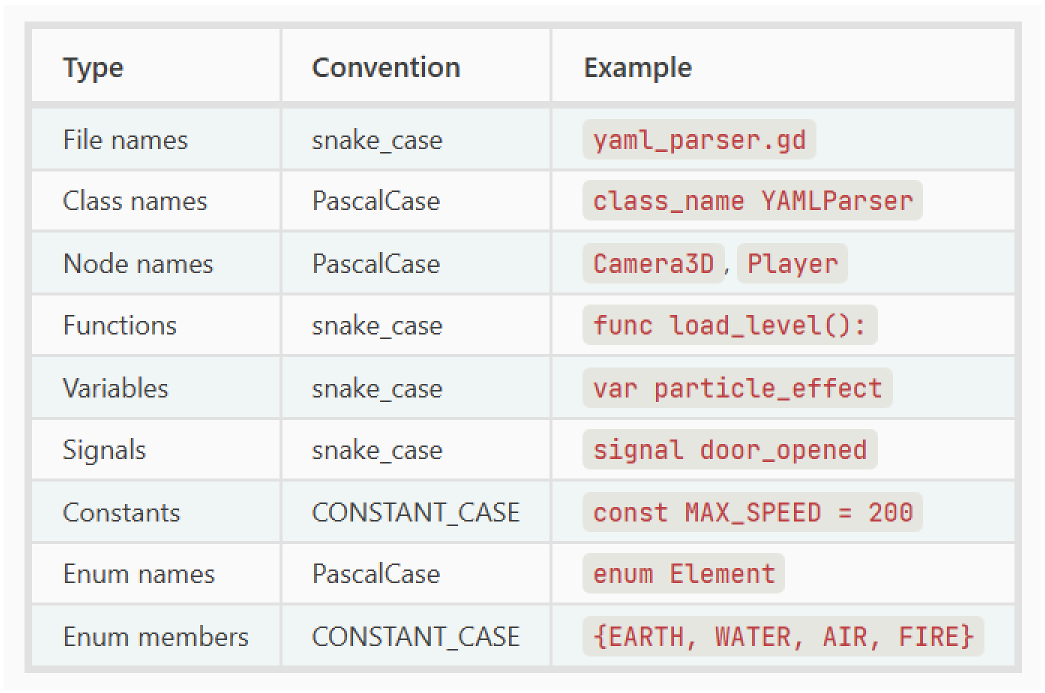
Directory Structure:

res://	general project files
● addons/	contains all addon files
● assets/	contains all asset files
○ audio/	audio files
○ dialogue/	dialogue assets
○ entities/	entity asset files
■ actors/	character assets
■ decor/	decorative assets
■ hazards/	hazard assets
■ objects/	other object assets
○ rooms/	room assets
○ ui/	menu assets
○ vfx/	visual effect and shader assets
● src/	contains all source files
○ audio/	audio resources
○ autoload/	autoload scenes and scripts
○ components/	component scripts
○ entities/	entity source files
■ actors/	character scenes and scripts
■ decor/	decorative scenes
■ hazards/	hazard scenes
■ objects/	other object scenes and scripts

■ triggers/	trigger scenes and scripts
○ gameplay/	main gameplay scene and script
○ rooms/	room scenes and scripts
○ ui/	menu scenes and scripts
○ vfx/	visual effect and shader scripts

Naming & Formatting Conventions

- All naming and formatting should conform to GDScript's style guide. Below are some important takeaways.
- Naming conventions should follow the table below:



Type	Convention	Example
File names	snake_case	`yaml_parser.gd`
Class names	PascalCase	`class_name YAMLParser`
Node names	PascalCase	`Camera3D`, `Player`
Functions	snake_case	`func load_level():`
Variables	snake_case	`var particle_effect`
Signals	snake_case	`signal door_opened`
Constants	CONSTANT_CASE	`const MAX_SPEED = 200`
Enum names	PascalCase	`enum Element`
Enum members	CONSTANT_CASE	`{EARTH, WATER, AIR, FIRE}`

Additional notes:

- Singletons will also use CONSTANT_CASE.
- all signal names should begin with the "on" prefix for readability (e.g. on_damage_taken)
- All scenes and code resource files should be in snake_case. This includes script files (including shader scripts), text files, spreadsheet files, JSON files, etc.
  - An exception is **map/room files**, which should follow the naming conventions outlined in the "Room Pipeline" section of the Design Bible
- Use tabs rather than spaces.
- File extensions should be in lowercase.

Name Abbreviations

- All naming abbreviations must be approved and listed here before it may be used in the codebase.

- to suggest a naming abbreviation, contact **Benjamin Levy**. If he approves the naming abbreviation, he will let you know and add it to this section.
    - Avoid naming abbreviations which may be confused with other names (e.g. abbreviating something called I## N## T## as I.N.T. will be confused with the primitive int).

Importing Guidelines

- All art, audio, and writing files should be imported into their respective folders within the file hierarchy to the most specific folder possible.
    - You may create additional folders if it supports and enhances the existing file hierarchy.
- Art & audio assets should be named according to the Art Bible and Audio Bible, respectively. Other files should follow the naming conventions outlined above (typically snake_case).

Coding Guidelines

- Use static typing when creating variables.
    - Static typing is when each variable is assigned a type when created (int, float, Vector2, etc.)
    - This makes code more readable, is more efficient at runtime, and avoids “dirty” or “hack-y” code.
- Connect signals in the editor when possible. This makes code more modular by reducing its direct reliance on other code, thus enforcing data-driven design. There may be exceptions, so use this as a general rule-of-thumb.
- When you need to connect a scene to an external dependency, implement dependency injection (ideally from an ancestor) in one of the following ways:
    - Emit a signal/event for external procedures to run in response to.
    - Create a public (non-underscore-prefixed) method for externals to directly call.
    - Create a publically settable (non-underscore-prefixed) field/property for externals to directly inject a reference or value into. Using @export is also acceptable.
- Make use of Godot’s “Access as Unique Name” feature when working with scenes that have a unique and important child. For example, the Player will only ever have one Camera child, so accessing it as a unique node makes it easier to access and manage via code. Additionally, accessing a node with a unique name means that if/when you move the node somewhere else in the scene tree, the unique name will automatically find and update its path, eliminating the need to go through your code and manually update the path to that child.
- Be generous with the number of comments you create. The more, the merrier! Similarly, focus on producing intuitive and readable code over efficient code. Other people will be

looking at your code, and code efficiency is unlikely to become an issue considering our project scope.

  - Create comments in accordance with Godot's recommendations: GDScript documentation comments - Godot Docs

Iterative Polish

- Reference: Rules of the Game: Five Techniques from Quite Inventive Designers
- Game polish is typically saved for the end of production so all core features are fully implemented before spending time with (comparatively less important) polish.
- However, polish is very important to a game's feel. A feature that may look bland in its bare-bones state becomes much more fun or interesting just by adding polish!

- When adding a new feature, try to include some *pizazz*, defined as some form of "juice" which typically emphasizes user feedback. Programmer art or scrappy *pizazz* is fine!
- Revisit *pizazz* as needed for updates.
- This methodology allows us to enhance the feel of the feature and polish the game iteratively rather than waiting until the very end of the project.

Tool Scripts (@tool)

- Reference: Running code in the editor - Godot Docs
- Tool classes should be used to create development tools for use in Godot's editor.
  - Tools should NOT be used for testing in-game functionality or any in-game features.
- Tool scripts must be put in the "tools" folder and include @tool at the top of the script.

Singletons/Autoloads

- Singletons should be used for managers and systems which are ALWAYS active.
  - Do NOT use singletons for managers which are only required for some segments or scenes in the game. For example, a minigame manager would not be a singleton because minigames appear and disappear for specific segments of gameplay.
- Singleton scripts must be put in the "autoload" folder and set as autoloads in the project settings.

Script Templates

- Reference: Creating script templates - Godot Docs

- Script templates should be used if you find yourself reusing the same script files as a base for other scripts, such as an NPC script. Script templates allow you to easily copy the code template through the editor.
- Script templates are created inside of the “script_templates” folder. Each sub-folder represents a node type (e.g. CharacterBody2D, Texture). These sub-folder names are case-sensitive and must be written in PascalCase just like the node names.
    - The “script_templates” folder does not appear in the Godot editor and must be accessed through your file explorer. Instead of using the Godot Editor, create script template files directly and edit them in your preferred external text editor.
- Script Templates should be named like any other script in Godot, with a focus on the abstract purpose of the script.
- At the start of every script template, include a meta header like the following:

  ```
  # meta-name: Platformer Movement
  # meta-description: Predefined movement for classical platformers
  ```

- This header determines the name and description which appears when browsing for a script template in-editor.
- The meta-name should be the same as the script template’s file name, but with each word Capitalized and underscores (_) being replaced with spaces ( ).
- The meta-description should be a one or two-sentence description of the script template’s purpose. Make it simple, clear, and concise.

Named Classes

- Reference: Registering Named Classes - Godot Docs
- Each “entity scene” (a scene representing an actor, interactable object, or which requires a primary controller script) should include a class_name (and optionally an icon) to make it clear what the scene’s purpose is. This also helps to keep the Node hierarchy clear and intuitive.
- At the start of every entity script, include the following header:

  class_name [insert class name]
  @icon(“[icon file path]”)

- the class_name should be the name of the entity or a name which clearly describes the scene’s purpose. This should be in PascalCase, just like Godot’s Node type names.
- @icon can be used to create a custom icon for the scene rather than Godot’s base icons.

Abstract Classes

- Reference: Abstract classes and methods - Godot Docs

- When creating a node or script which is likely to be reused, make it an abstract class or script template, respectively.

Source Control Rules

- Our repository's Git Moderator is **Jeremy King**. He can help you if you have questions about the repository or encounter any merge conflicts.
    - The Git Moderator is responsible for answering questions about the repository, helping to resolve any merge conflicts, and merging completed iterations on the "dev" branch into the "prod" branch.
- The Git repository has two core branches: "dev" and "prod".
    - The "dev" branch acts as the base development branch that feature branches must be created from. All other branches must be PR'd into "dev" for testing and QA before a functional version gets merged into "prod".
    - The "prod" branch contains the last fully-functional version of the project, with no critical bugs or unfinished features.
- When to Create a Branch
    - Create a new branch to begin working on a new feature, bugfix, or anything else which requires changes to the repository. Think of the branch as a container for all changes specific to adding a specific feature or fixing a specific bug.
    - You may not create branches off of "prod"; only branch off of "dev" or other branches.
- Naming branches
    - For features: feature/[feature name]
    - For small changes: tweak/[tweak name]
    - For bugfixes: bugfix/[bug name]
    - Use snake_case for naming (e.g. "feature/player_object")
- Naming commits
    - Use active voice. For example, "Implemented player movement" rather than "Player movement implemented"
    - Include any notes about the commit in the description. Describe what needed to be changed, any bugs encountered, or remaining tweaks to be made.
- Naming Pull Requests
    - Pull requests should be prepended with "Feature:", "Tweak:", or "Bugfix:" depending on the purpose of the branch. The rest of the name should correspond to the specific change being made (e.g. "Feature: Player Movement" or "Bugfix: Invincible Enemies")
    - In the description, describe the general changes made, note any known bugs or tweaks that should be made.
- When to Pull Request
    - Create a Pull Request when the deliverable for your branch is in a complete state.

    - Before creating a Pull Request, merge the most recent version of “dev” into your branch to ensure your changes do not conflict.
    - Once you create a Pull Request, send a ping in our Discord server asking for someone to review and approve your Pull Request.
    - If your reviewer writes a comment on the Pull Request that is marked as “<reviewer> requested changes”, you will need to re-request a review. After new commits have been pushed to your branch, the GitHub Pull Request page will show a new little refresh icon that you can click on to re-request review.
    - When reviewing a Pull Request, make sure the deliverable is functional and the code is clean and well-documented.
    - When reviewing a Pull Request, if further changes are required, make sure to write a comment specifying what changes are necessary and click the “Request Changes” button before submitting your review.
    - After approving a Pull Request, delete the associated branch to keep the repository’s branch list clean and straightforward.
- Handling merge conflicts
    - In the event you encounter a merge conflict, contact the Git Moderator. Together, you will both review the conflicts and determine which changes must be modified or overwritten to resolve the conflict.
    - If no changes can be modified or overwritten, the branch must be rolled back to before the conflicting changes were made.
    - Making changes on existing code in “dev” to accommodate changes from another branch should be a last resort.
- Commit habits
    - Commit early and commit often. Ideally, each change you make in code should be a separate commit. This makes it easier to rollback problematic changes and encourages creating simple, bite-sized code.

Code Reviews & Pipeline

- Before beginning a task, review its requirements with the owner of the task’s design domain and other relevant people. Make sure you understand the exact task specifications, how the task should be implemented, and what existing systems have to be worked around.
- Once a task is completed, create a Pull Request as per the Source Control Rules and send a ping in our Discord server so another programmer may review it.

Building Guidelines

- Reference: Exporting Projects - Godot Docs
- Builds should be created every week before designated playtesting times.

- Builds must be approved by the Git Moderator before they may be used externally.
- Before creating a build, make sure to download the proper export templates for Godot. This can be done through the editor or on Godot's website.
- To make a build of the game:
    1. If needed, add a new platform preset by clicking the "Add…" button on the top of the export menu.
    2. Resolve any complaints (colored red) in the export menu.
    3. Click the "Export Project…" button. This will export a Godot executable and all project data for the selected platform preset.
    4. Put the project's contents into a folder.
    5. Package the folder as a ZIP and upload it to the GitHub project's "Releases" page.
- To make a build for MacOS specifically:
    1. Set a valid and unique *bundle identifier* in the *Application* section of the export options.
    2. Set *Code Signing* > *Codesign* to *Built-in (ad-hoc only)*.
    3. Set *Notarization* > *Notarization* to *Disabled*.
    4. If using Windows, make sure to export the project **as a .zip file**.
    5. If MacOS users have trouble running the application, refer to Running Godot apps on macOS - Godot Docs
- Put exported builds within the project's "builds" folder and create the proper sub-folder according to the purpose of the build.

Version Numbering

- Reference: Semantic Versioning 2.0.0
- Start with 1.0.0, representing MAJOR.MINOR.PATCH.
- Increment the:
    1. MAJOR version when you make incompatible API changes (we will likely never do this)
    2. MINOR version when you add functionality in a backward compatible manner (likely most major features)
    3. PATCH version when you make backward compatible bug fixes

Debugging & Testing

- Debugging utilities should be created as soon as possible. Ideally, create an in-game debug menu which can spawn entities and teleport the Player to different locations. This will make recreating specific scenarios easier.

Unit Testing

- Unit testing is more difficult to implement for games compared to regular software, as the game's state can be hard to quantify and measure in simple numbers or booleans. However, it still serves a purpose for more complex systems or calculations.
- You may decide when a unit test is or is not required. There is no strict requirement on what should or should not have a unit test. However, make sure to think about it for every feature you implement. Just ask yourself, "is it easy or important to make a unit test for this system?"
- Use `assert()` statements for unit testing.

Logging

- Reference: [Grug on Logging - The Grug Brained Developer]
- Logging is very important for debugging and testing large-scale systems. It may not be as useful for this smaller-scale project, but a good rule-of-thumb is to include some logging functionality with all of your code.
- Rule of thumb: **log all major logical branches within code (like if/for statements).**
- For each script which contains logging code, make a boolean to enable/disable logging (with the default being "disabled") so as to not clog the output log.
- Remember to use different printing/logging functions in different circumstances:

print() -	Logs a regular message
push_warning() -	Logs a warning (in yellow)
push_error() -	Logs an error (in red)

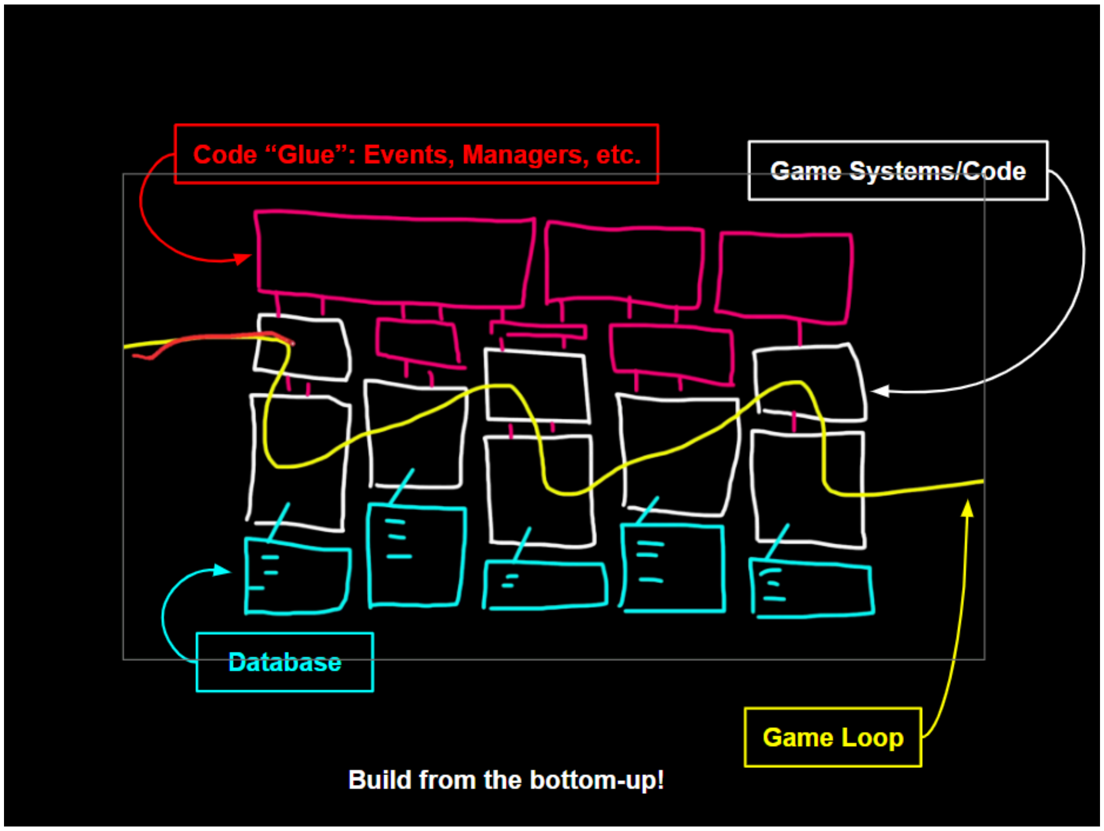
Audio Integration

- Audio integration will be done using Godot's built-in audio tools. They are capable of vertical remixing & horizontal resequencing, and can layer on various audio effects as well.
- An example of the audio tools' utility can be found here: [Godot 4.3 Adaptive Music Demonstration - TheLotanos]
- Additional information about audio integration and pipeline can be found here: Audio Bible

Data-driven Design

- Reference: [Best Code Architecture For Indie Games - Jonas Tyroller]
- Separate your codebase into 3 distinct layers: the database, game systems, and "glue" code
    - The Database is where your immutable data should live. These should live in text files, CSVs, or other file formats which are easy to read from.

        - Examples: player/NPC parameters, dialogue, string text
        - Keep relevant data in one place; if it is *game-designed* together, store it together
        - Godot can also store data as Resources to make it easier to access in-engine.
        - Consider designing the database first, then building the game systems on top of it. This keeps your codebase data-first.
    - Game Systems should not contain any data. Instead, they should read from the Database at runtime to configure themselves and set up the game world.
        - e.g. the Player class accesses a text file with parameters for movement speed, jump height, health, etc.
    - The "Glue" Code connects the game systems and database together indirectly. "Glue" code can be events, game managers, connector scripts, or in-editor connections. It prevents data and game systems from being directly reliant on each other, thus letting the codebase stay *modular* and *scalable*.
        - However, avoid "universal glue": for example, if you find your game manager becoming bloated with the functionality of multiple systems, you should separate the game manager into multiple smaller managers that connect with each other.
- There is one system that will always be one bit of "spaghetti code" you cannot change: time, specifically the game loop that is always calling from other game systems.
    - Manage it by tightly controlling the execution flow; make sure everything runs in the order you want it to.
- If you find two systems are difficult to keep entirely separate (e.g. health and attack), consider combining them into one shared system!


Code "Glue": Events, Managers, etc.
Game Systems/Code
Database
Game Loop
Build from the bottom-up!

Triggers

Triggers are our solution to triggering game events. They function as a way to start cutscenes, signal when the player interacts with a door, manipulate the camera, and much much more.

Triggers are modeled after Entity Component Systems (ECS). ECS prioritizes composition over inheritance, meaning objects and their functionality is a "has a" relationship instead of an "is a" relationship. For example, a player would be an entity and its health system would be a component attached to it. This works well for Godot's node tree system and makes up for inheritance features Godot lacks.

All trigger scripts must define and override an "_on_trigger" function that defines what happens when the trigger is activated. Triggers can activate under different circumstances; some triggers activate when the player enters them and some triggers activate when the player presses the interact button.

Triggers also have an enabled variable that tracks whether they can trigger in the first place. All triggers use the Event Flag system to determine whether they are enabled. Event Flags are described in a later section of this document. Triggers can be given specific event flags that need to be true or false for the trigger to be enabled. This lets us dynamically enable or disable certain triggers (for example, disabling cutscenes) whenever event flags change to reflect the latest game state. This can be extended to any entity in the game using the ProcessingComponent script, a component that can be attached to any entity to enable or disable it according to event flags. This lets us further push dynamic game objects, allowing NPCs to appear and disappear in rooms or walls to melt as the game progresses.

Groups & Layers

When creating a new Node, make sure it is added to the proper groups and layers.

Named Physics/Collision Layers

Before adding Nodes to a new physics layer, contact **Benjamin Levy** to confirm the need for a new layer. Then, make sure to name the layer in accordance with the type of entity which uses the layer. Name all layers using snake_case.

Layer #	**Name**	**Purpose**
1	player	The Player's collision layer.
2	surfaces	Collision layer for all walls, floors, and ceilings.
3	platforms	Collision layer for all platform objects.
4	hazards	Collision layer for static objects which may damage the Player.
5	interact	Collision layer for interactable objects.
6	grab	Collision layer for grabbable objects.
7	enemies	Collision layer for the Ice Scream enemy.
8	leaf_mode_passable	Collision layer for platforms which the Player may pass through while Leaf Dashing.
9	npc	Collision layer for all NPCs, including Az and Winston.

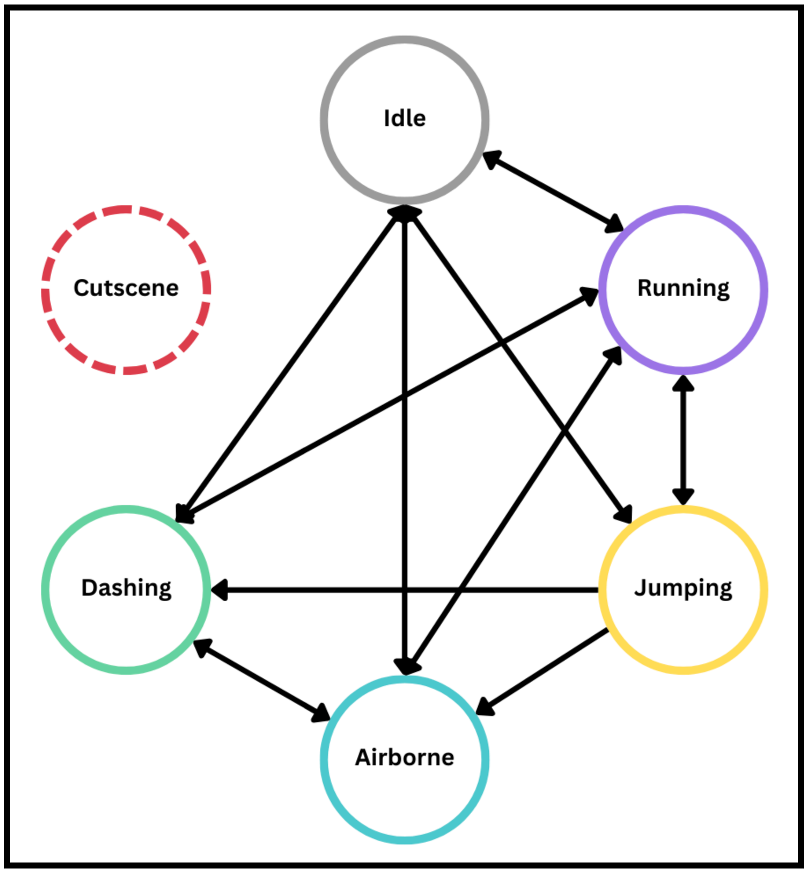
10	fenn_passable	Collision layer for platforms which the Player may pass through while NOT Leaf Dashing.
11	dash_zone	Collision layer for Wind Zones set to the infinite dash mode.
12	no_dash_zone	Collision layer for Wind Zones set to the no-dash mode.

Event Flags

- Event flags are used to signify whether one-time events have been triggered or not.
- Every event flag is stored in the JSON resource `event_flags_db.tres.`
  - Each value is a boolean, either `false` for untriggered or `true` for triggered.
  - Each key's name follows the naming convention SUBJECT_VERB (e.g. "heater_one_activated")
  - For conversation event flags, make the subject the character and context of the conversation. Make the verb "triggered". (e.g. "az_meet_great_hall_triggered")
- At compile time, the `event_flags` autoload script copies the JSON information into a dictionary which can be read & set by any other script.
- Please get event flags using the `get_flag()` function and set event flags using the `set_flag()` function.
  - These functions perform error checking and update other event flag-relevant variables.

Player State Machine

The Player has 6 states, and their transition relationships are modeled in the diagram below with arrows:



Saving/Loading

- The autoload "SaveManager" is used to save event flags and the Player's current room to disk at runtime.
- Whenever an event flag is modified, the SaveManager is called to overwrite the event flags database file with the new information.
- Similarly, whenever the Player enters a new room, the SaveManager is called to store the Player's new room in a separate database file.
- When the game executable is run, the Main Menu checks to see if modified save data exists. If so, the menu adds a new UI option called "Continue" which lets the user load back into the last saved room with the modified event flags loaded.
    - If the game does not detect any save data, the "Continue" option will not appear.
- If the user selects the "New Game" option, the game will begin with all event flags set to "false". The next time an event flag is set, the event flag save data will be overwritten with the new event flags.

Database

Player

Reference: Platformer Toolkit - Game Maker's Toolkit

Category	Parameter	Data Type	Purpose
Base Data	terminal_velocity	float	The Player's maximum positive Y-velocity.
Run	run_max_speed	float	The Player's maximum X-velocity when running. The Player's actual X-velocity may exceed this value if forces beside running momentum are applied.
Run	ground_acceleration	float	The Player's X-velocity gain per second while running.
Run	ground_deceleration	float	The Player's X-velocity loss per second while not running & on the ground.
Run	ground_turn_speed	float	The Player's X-velocity gain per second while turning to run in the opposite direction on the ground.
Run	ground_friction	float	The Player's X-velocity loss per second while over their max run speed.
Run	air_acceleration	float	The Player's X-velocity gain per second while moving in the air.
Run	air_deceleration	float	The Player's X-velocity loss per second while in the air and not moving.
Run	air_turn_speed	float	The Player's X-velocity gain per second while turning to move in the opposite direction in the air.

Jump	jump_height	float	How high the peak of the Player's jump reaches in world units.
Jump	jump_time_to_peak	float	How long (in seconds) it takes for the Player to reach the peak of their jump.
Jump	fall_gravity_multiplier	float	Multiplier for gravitational pull which applies if the Player is jumping and releases the jump button. This enables variable jump height.
Jump	jump_coyote_time	float	How long (in seconds) after starting to fall where the Player may still initiate a jump.
Jump	jump_buffer_time	float	How long (in seconds) before landing on the ground where the Player may "queue" a jump to trigger as soon as they land.
~~Jump~~	~~jump_corner_rounding_distance~~	~~float~~	~~UNUSED: How close (in game units) the Player must be to a ledge before they slide onto it at the peak of their jump.~~
Leaf Meter	meter_dash_zone_buildup_rate	float	The amount of wind per second that the Player gains while in a Dash Zone.
Leaf Meter	meter_dash_drain_rate	float	The amount of wind per second that the Player loses while Leaf Dashing.
Leaf Meter	meter_dash_end_drain	float	The amount of wind drained after ending a Leaf Dash.
~~Leaf Meter~~	~~meter_pile_drain_rate~~	~~float~~	~~UNUSED: The amount of wind per second that the Player loses while in Leaf Pile mode.~~
Leaf Meter	meter_cooldown_time	float	Time in seconds after landing on the ground before the Player may Leaf Dash again.
Dash	dash_min_speed	float	The Player's minimum speed

			while Leaf Dashing.
Dash	dash_max_speed	float	The Player's maximum speed while Leaf Dashing.
Dash	dash_speed_friction	float	The Player's velocity loss per second while over their max dash speed.
Dash	dash_acceleration	float	The Player's velocity gain per second while Leaf Dashing.
Dash	dash_angular_turn_speed	float	The Player's turn speed (in degrees) while Leaf Dashing. Not scaled by delta time.
~~Dash~~	~~dash_deceleration~~	~~float~~	~~UNUSED: The Player's speed loss per second while Leaf Dashing with very little wind left.~~
~~Dash~~	~~dash_angular_turn_speed_deceleration~~	~~float~~	~~UNUSED: The Player's turn speed loss per second while Leaf Dashing with very little wind left.~~
~~Dash~~	~~meter_dash_deceleration_start~~	~~float~~	~~UNUSED: If the Player is Leaf Dashing with this amount of wind or less in their Leaf Meter, they begin slowing down.~~
Dash	dash_end_velocity_multiplier	float	When ending a Leaf Dash, multiply velocity by this value to "fling" the Player.
Dash	dash_end_max_velocity	float	When ending a Leaf Dash, this is the max "fling" velocity the Player can have.
Post-dash	post_dash_gravity	float	Gravity applied to Player during the post-dash mode.
Post-dash	post_dash_fast_fall_gravity_multiplier	float	Multiplier for Player gravity while pressing the move_down action during the post-dash mode.
Misc.	hit_recoil_velocity	float	How far the Player is launched

			after being hit.
Misc.	hit_recoil_direction	Vector2	The direction the Player is launched after being hit.
Misc.	hit_invincibility_time	float	How long after being hit that the Player is invincible for.

Ice Scream

Note: This database uses StringNames (strings with a prepended “&”) as keys.

Parameter	**Data Type**	**Purpose**
end_range	int	
gravity	int	
max_speed	float	
normal_acceleration	float	
normal_deceleration	float	
turn_speed	float	