MMBN: Godot Style Guide

Maintained by: GameDevone

Table of Contents

MMBN: Godot Style Guide

This style guide lists conventions to write elegant GDScript. The goal is to encourage writing clean, readable code and promote consistency across the project.

All scripts will contain documentation that becomes a mixture of the Godot Style Guide and Doxygen Style for Python.

Formatting

Encoding and Special Characters

- Use line feed (LF) characters to break lines, not CRLF or CR. (editor default)
- Use one line feed character at the end of each file. (editor default)
- Use UTF-8 encoding without a byte order mark. (editor default)
- Use Tabs instead of spaces for indentation. (editor default)

Indentation

Each indent level should be one greater than the block containing it.

Good:

```
for i in range(10):
    print("hello")
```

Bad:

```
for i in range(10):
  print("hello")

for i in range(10):
        print("hello")
```

Use 2 indent levels to distinguish continuation lines from regular code blocks.

Good:

```
effect.interpolate_property(sprite, "transform/scale",
            sprite.get_scale(), Vector2(2.0, 2.0), 0.3,
            Tween.TRANS_QUAD, Tween.EASE_OUT)
```

Bad:

```
effect.interpolate_property(sprite, "transform/scale",
    sprite.get_scale(), Vector2(2.0, 2.0), 0.3,
```

```
    Tween.TRANS_QUAD, Tween.EASE_OUT)
```

Exceptions to this rule are arrays, dictionaries, and enums. Use a single indentation level to distinguish continuation lines:

Good:

```
var party = [
    "Godot",
    "Godette",
    "Steve",
]

var character_dir = {
    "Name": "Bob",
    "Age": 27,
    "Job": "Mechanic",
}

enum Tiles {
    TILE_BRICK,
    TILE_FLOOR,
    TILE_SPIKE,
    TILE_TELEPORT,
}
```

Bad:

```
var party = [
        "Godot",
        "Godette",
        "Steve",
]

var character_dir = {
        "Name": "Bob",
        "Age": 27,
        "Job": "Mechanic",
}

enum Tiles {
        TILE_BRICK,
        TILE_FLOOR,
        TILE_SPIKE,
        TILE_TELEPORT,
}
```

Trailing comma

Use a trailing comma on the last line in arrays, dictionaries, and enums. This results in easier refactoring and better diffs in version control as the last line doesn't need to be modified when adding new elements.

Good:

```
enum Tiles {
    TILE_BRICK,
    TILE_FLOOR,
    TILE_SPIKE,
    TILE_TELEPORT,
}
```

Bad:

```
enum Tiles {
    TILE_BRICK,
    TILE_FLOOR,
    TILE_SPIKE,
    TILE_TELEPORT
}
```

Trailing commas are unnecessary in single-line lists, so don't add them in this case.

Good:

```
enum Tiles {TILE_BRICK, TILE_FLOOR, TILE_SPIKE, TILE_TELEPORT}
```

Bad:

```
enum Tiles {TILE_BRICK, TILE_FLOOR, TILE_SPIKE, TILE_TELEPORT,}
```

Blank lines

Surround functions and class definitions with two blank lines:

```
func heal(amount):
    health += amount
    health = min(health, max_health)
    emit_signal("health_changed", health)


func take_damage(amount, effect=null):
    health -= amount
    health = max(0, health)
    emit_signal("health_changed", health)
```

Use one blank line inside functions to separate logical sections.

Line length

If you can, try to keep lines under 80 characters. This helps to read the code on small displays and with two scripts opened side-by-side in an external text editor. For example, when looking at a differential revision.

One statement per line

Never combine multiple statements on a single line. No, C programmers, not even with a single line conditional statement.

Good:

```
if position.x > width:
    position.x = 0

if flag:
    print("flagged")
```

Bad:

```
if position.x > width: position.x = 0

if flag: print("flagged")
```

The only exception to that rule is the ternary operator:

```
next_state = "fall" if not is_on_floor() else "idle"
```

Avoid unnecessary parentheses

Avoid parentheses in expressions and conditional statements. Unless necessary for order of operations, they only reduce readability.

Good:

```
if is_colliding():
    queue_free()
```

Bad:

```
if (is_colliding()):
    queue_free()
```

Boolean operators

Prefer the plain English versions of boolean operators, as they are the most accessible:

- Use `and` instead of `&&`.
- Use `or` instead of `||`.

You may also use parentheses around boolean operators to clear any ambiguity. This can make long expressions easier to read.

Good:

```
if (foo and bar) or baz:
    print("condition is true")
```

Bad:

```
if foo && bar || baz:
    print("condition is true")
```

Comment spacing

Regular comments should start with a space, but not code that you comment out. This helps differentiate text comments from disabled code.

Good:

```
# This is a comment.
#print("This is disabled code")
```

Bad:

```
#This is a comment.
# print("This is disabled code")
```

Whitespace

Always use one space around operators and after commas. Also, avoid extra spaces in dictionary references and function calls.

Good:

```
position.x = 5
position.y = target_position.y + 10
dict["key"] = 5
my_array = [4, 5, 6]
print("foo")
```

Bad:

```
position.x=5
position.y = mpos.y+10
dict ["key"] = 5
myarray = [4,5,6]
print ("foo")
```

Don't use spaces to align expressions vertically:

```
x        = 100
y        = 100
```

```
velocity = 500
```

Quotes

Use double quotes unless single quotes make it possible to escape fewer characters in a given string. See the examples below:

```
# Normal string.
print("hello world")

# Use double quotes as usual to avoid escapes.
print("hello 'world'")

# Use single quotes as an exception to the rule to avoid escapes.
print('hello "world"')

# Both quote styles would require 2 escapes; prefer double quotes if it's a tie.
print("'hello' \"world\"")
```

Numbers

Don't omit the leading or trailing zero in floating-point numbers. Otherwise, this makes them less readable and harder to distinguish from integers at a glance.

Good:

```
var float_number = 0.234
var other_float_number = 13.0
```

Bad:

```
var float_number = .234
var other_float_number = 13.
```

Use lowercase for letters in hexadecimal numbers, as their lower height makes the number easier to read.

Good:

```
var hex_number = 0xfb8c0b
```

Bad:

```
var hex_number = 0xFB8C0B
```

Take advantage of GDScript's underscores in literals to make large numbers more readable.

Good:

```
var large_number = 1_234_567_890
var large_hex_number = 0xffff_f8f8_0000
var large_bin_number = 0b1101_0010_1010
# Numbers lower than 1000000 generally don't need separators.
var small_number = 12345
```

Bad:

```
var large_number = 1234567890
var large_hex_number = 0xfffff8f80000
var large_bin_number = 0b110100101010
# Numbers lower than 1000000 generally don't need separators.
var small_number = 12_345
```

Naming conventions

These naming conventions follow the Godot Engine style. Breaking these will make your code clash with the built-in naming conventions, leading to inconsistent code.

File names

Use snake_case for file names. For named classes, convert the PascalCase class name to snake_case:

```
# This file should be saved as `weapon.gd`.
extends Node
class_name Weapon
# This file should be saved as `yaml_parser.gd`.
extends Object
class_name YAMLParser
```

This is consistent with how C++ files are named in Godot's source code. This also avoids case sensitivity issues that can crop up when exporting a project from Windows to other platforms.

Classes and Nodes

Use PascalCase for class and node names:

```
extends KinematicBody
```

Also use PascalCase when loading a class into a constant or a variable:

```
const Weapon = preload("res://weapon.gd")
```

Functions and Variables

Use snake_case to name functions and variables:

```
var particle_effect
func load_level():
```

Prepend a single underscore (_) to virtual methods functions the user must override, private functions, and private variables:

```
var _counter = 0
func _recalculate_path():
```

Signals

Use the past tense to name signals:

```
signal door_opened
signal score_changed
```

Constants and Enums

Write constants with CONSTANT_CASE, that is to say in all caps with an underscore (_) to separate words:

```
const MAX_SPEED = 200
```

Use PascalCase for enum names and CONSTANT_CASE for their members, as they are constants:

```
enum Element {
    EARTH,
    WATER,
    AIR,
    FIRE,
}
```

Code Order

Organize GDScript code this way.

```
01. tool
02. class_name
03. extends
04. # docstring

05. signals
06. enums
07. constants
08. exported variables
09. public variables
10. private variables
11. onready variables

12. optional built-in virtual _init method
13. built-in virtual _ready method
14. remaining built-in virtual methods
15. public methods
16. private methods
```

This code order follows four rules of thumb:

1. Properties and signals come first, followed by methods.
2. Public comes before private.
3. Virtual callbacks come before the class's interface.
4. The object's construction and initialization functions, `_init` and `_ready`, come before functions that modify the object at runtime.

Class Declaration

If the code is meant to run in the editor, place the tool keyword on the first line of the script.

Follow with the class_name if necessary. You can turn a GDScript file into a global type in your project using this feature. For more information, see GDScript basics.

Then, add the extends keyword if the class extends a built-in type.

DocString, File Header and Function Header

Following that, you should have the class's docstring as a file header with comments as shown below. This docstring is a derivative of Python comments used by Doxygen in the case the project documentation is generated automatically.

```
class_name MyNode
extends Node
"""
# file       my_node.gd
# author(s)  Devone Reynolds
# par        email: gamedevone1@gmail.com
# date       31 DEC 2020
# copyright  Copyright (c) 2021 GameDevone
# brief      Base class for all Nodes need for personal use.
# note       To use this Node you will have to attach it to a Node in the Scene
"""
```

Using this format the file header will display this information:

- The name of the file
- Who made contributions to the file as authors
- Email of those contributors
- Date of creation,
- Copyright information
- Short description of the purpose of that file
- Additional notes on how to use the script if it is not obvious

Functions will use a similar header that will display:

- A short description of what the function does
- All parameters and how they serve to perform the function
- What the function return if any value at all

```
"""
# brief                  Returns the direction of the camera movement from the
                         player
# note                   Static functions do not get access to 'self'
# param velocity_current
                         Current speed of the player
# param move_direction
                         Direction that the player is moving
# param delta
                         Time that has passed since the last call to _process or
                         _physics_process
# return                 Estimated speed over delta time as a Vector 3
"""

func calculate_velocity(
            velocity_current: Vector3,
```

```
            move_direction: Vector3,
            delta: float
      ) -> Vector3:
            # Calculate the new velocity and ensure it does not exceed max speed
            var velocity_new := move_direction * move_speed
            if velocity_new.length() > max_speed:
                  velocity_new = velocity_new.normalized() * max_speed
            velocity_new.y = velocity_current.y + gravity * delta

            # Return newly calculated velocity
            return velocity_new
```

Signals and properties

Write signal declarations, followed by properties, that is to say, member variables, after the docstring.

Enums should come after signals, as you can use them as export hints for other properties.

Then, write constants, exported variables, public, private, and onready variables, in that order.

Each variable will have a comment above it describing its use and be separated by a single empty line. The only exception to this are the signals which should be written in a self explanatory way.

```
signal spawn_player(position)

# Possible types of jobs for the player
enum Jobs {KNIGHT, WIZARD, ROGUE, HEALER, SHAMAN}

# Number of lives the player starts with
const MAX_LIVES = 3

# Current job the player uses
export(Jobs) var job = Jobs.KNIGHT

# Amount of health the player starts with
export var max_health = 50

# Amount of damage player deals
export var attack = 5

# Current health the player has
var health = max_health setget set_health

# How fast the player moves
var _speed = 300.0
```

```
# Reference to the Sword Node
onready var sword = get_node("Sword")
```

Member variables

Don't declare member variables if they are only used locally in a method, as it makes the code more difficult to follow. Instead, declare them as local variables in the method's body.

Local variables

Declare local variables as close as possible to their first use. This makes it easier to follow the code, without having to scroll too much to find where the variable was declared.

Methods and static functions

After the class's properties come the methods.

Start with the `_init()` callback method, that the engine will call upon creating the object in memory. Follow with the `_ready()` callback, that Godot calls when it adds a node to the scene tree.

These functions should come first because they show how the object is initialized.

Other built-in virtual callbacks, like `_unhandled_input()` and `_physics_process`, should come next. These control the object's main loop and interactions with the game engine.

The rest of the class's interface, public and private methods, come after.

```
"""
# brief      Called when the State Machine is initialized.
"""
func _init():
    add_to_group("state_machine")


"""
# brief      Called when the node is 'ready', i.e. when both the node and
             its children have entered the scene tree.
"""
func _ready():
    connect("state_changed", self, "_on_state_changed")
```

```
    _state.enter()


"""
# brief             Called when an InputEvent hasn't been consumed by _input() or
                    any GUI.
# param event
                    Input event that occured.
"""
func _unhandled_input(event):
    _state.unhandled_input(event)

"""
# brief                   Change the current state to the requested state.
# param target_state_path
                          Scene Tree path to the requested state.
# param msg
                          Any variables to be passed along to the requested state.
"""
func transition_to(target_state_path, msg={}):
    if not has_node(target_state_path):
        return

    var target_state = get_node(target_state_path)
    assert(target_state.is_composite == false)

    _state.exit()
    self._state = target_state
    _state.enter(msg)
    Events.emit_signal("player_state_changed", _state.name)
```

Static typing

Since Godot 3.1, GDScript supports optional static typing.

Declared types

To declare a variable's type, use `<variable>: <type>`:

```
var health: int = 0
```

To declare the return type of a function, use `-> <type>`:

```
func heal(amount: int) -> void:
```

Inferred types

In most cases you can let the compiler infer the type, using `:=`:

```
var health := 0 # The compiler will use the int type.
```

However, in a few cases when context is missing, the compiler falls back to the function's return type. For example, `get_node()` cannot infer a type unless the scene or file of the node is loaded in memory. In this case, you should set the type explicitly.

Good:

```
onready var health_bar: ProgressBar = get_node("UI/LifeBar")
```

Bad:

```
# The compiler can't infer the exact type and will use Node
# instead of ProgressBar.
onready var health_bar := get_node("UI/LifeBar")
```