Code Standards

This document is designed to be fairly comprehensive and sufficiently specific.

That being said: just because it's not covered here doesn't mean it's okay. Common sense people, common sense.

Language

We will be using Godot with C#.

- Godot uses an internal scripting language, GDScript, by default. Due to project requirements, we will instead be using C#.
- Godot currently has strong native C# support. Much of the official documentation comes with equivalent C# example code.

Portion of this guide are adapted from Google's C# style guide.

> While Google's style guide can serve as a good reference for more examples and greater detail in some areas, this document exists as the definitive set of standards for this project. If it is not explicitly covered in this document, it is not considered an official requirement.

Additionally, it is encouraged to consult the official Godot docs for best practices as well.

Naming Conventions

Case Conventions

- Classes, methods, enums, public properties and fields, file names, namespaces: `PascalCase`
- Local variables, parameters: `camelCase`
- Constants: `CAPITAL_SNAKE_CASE`

Other Naming Conventions

Excessive verbosity is preferable to excessive abbreviation.

- We all have code completion, and it's easier to hit the `tab` key than to figure out what a random acronym is supposed to mean.

- As always, excessive verbosity is NOT *encouraged*. It is, however, the preferred extreme to land in.

Use `i`, `j`, and `k` for loop variables, in that order.

- If you need to go beyond k, something is very wrong structurally and we have a bigger problem.

DO NOT use `x` or `y` for anything OTHER than spatial coordinates.

- `x` and `y` are considered reserved for working with graphical coordinates.
- This is universally applicable anywhere in the codebase, *especially* when used as class fields.
- This extends to include the use of `x` and `y` within larger names. (Eg, `velocityX.`)

Whitespace

1 statement per line maximum.

1 assignment per statement.

Indentation should consist of 4 spaces.

- Use of indentation should be 'standard' and structurally appropriate.

Unary operators should NOT have a space. All other operators MUST have a space.

```
!unarySpaced

other % spacing
```

Braces

Opening braces should always be on the same line as the preceding statement, ie with no line break before, and there should always be a single space before the opening brace.

```
LikeThis() {
    statements;
}
```

`else` statements go on the same line as both the preceding and following braces.

```
} else {
```

Braces should always be used, **even when not strictly necessary.** For example, in loops.

```
while ()
    a single line can technically be used its unacceptable;
```

Instead:

```
while () {
    code blocks should always be contained within braces;
}
```

Parentheses

A space always goes between keywords such as `if/for/while` and following parentheses. A space should also follow commas. Methods, however, do not require a space.

```
if (you have a conditional, loop, comma) {
    Put(a space, but methods are different);
}
```

No space after opening parentheses or before closing parentheses.

```
(like so)
```

Parentheses must always be used when a compound expression is used within a larger expression. Never rely on order of operations when the order effects the result. If there is more than one operator, group everything.

Examples

This is acceptable (and preferred) because there is only a single operator.

```
offset + amplitude;
```

This is unacceptable.

```
offset + timeStep * frequency;
```

Assuming the multiplication is intended to take place first, `timeStep` should be grouped with `frequency` in parentheses.

```
offset + (timeStep * frequency);
```

> Note that an outer layer of parentheses is not necessary. Grouping the entire expression is not required, and in fact discouraged.

The following expression is acceptable. This is because changing the order of operation does not matter.

```
damage * damageMultiplier * timeDelta;
```

Code Comments

Excessive and unnecessary commenting is preferable to insufficient commenting.

- While not desireable, it is easier to trim down or simply delete excessive comments than it is to puzzle out what poorly documented code is supposed to do.

Focus on **what** and **why.**

- *What* does the code do?
- *Why* is it needed?

Explaining *how* is not nearly as relevant. The assumption is that we can all understand individual statements, or if not, we can quickly look things up. If the underlying mechanism does need explanation, recursively breaking things down within the body of the code could be helpful.

Comments go above the code they relate to.

Breaking Things Down. And Also Nesting.

Excessive helper functions and code subdivision is preferable to spaghetti.

- Once again, a proper balance is the most desireable.
- However, it is easier to quickly track down and put together well-documented components than it is to untangle masses of spaghetti.
- Rule of thumb: if it's larger than your screen, consider splitting.

Nesting depth will be interpreted in a context-dependent way. That being said, consider a depth of FIVE to be the default maximum.

- Within a larger body of code, the way we think about nesting depth can sometimes be "reset."
- Even so, nesting depth and repeated code tolerance go hand-in-hand. If "absolute" nesting depth starts getting high, it's also a good time to consider breaking things down.

Project Organization

Start with using the Godot defaults and official guidelines here. Most of the project is going to be contained within the scope of a single Godot root directory.

Seriously. Learning Godot basics and following the basic tutorials will do a better job at explaining this than this overall style guide.