# Contribution guidelines for OpenTK

## Preface

First of all, thank you for considering contributing to the OpenTK

project! It's a large codebase with a lot of twists and turns, and a

helping hand is always welcome.

There are multiple ways to contribute to the project - creating bug

reports, opening pull requests, as well as commenting on and engaging in

discussions about other contributions, to name a few. This document is

intended as a set of guidelines to help your contribution get accepted

faster, maintain a high standard, and to help us (the maintainers) set a

few ground rules for working with us.

If you have any questions about the contents of this document, the code,

or how to contribute, please drop us a line on [Gitter][1] or

[Discord][2]. We'll be happy to answer as best we can.

#### Table of Contents

1. [Things to keep in mind](#things-to-keep-in-mind)

2. [Setting Up](#setting-up)

3. [Pull Requests](#pull-requests)

1. [Bug Fixes](#bug-fixes)

2. [New Features](#new-features)

3. [Cosmetic & Stylistic Changes](#cosmetic-&-stylistic-changes)

4. [Breaking Changes](#breaking-changes)

4. [Bug Reports](#bug-reports)

5. [Discussions & Suggestions](#discussions-&-suggestions)

## Things to keep in mind

Like a lot of other modern projects, OpenTK is written for multiple

platforms and operating systems. Therefore, it's important to keep this

in mind when contributing to the project - otherwise, it may make

accepting your contribution much more difficult. You'll want to consider

that the bug you're experiencing might not be present on other platforms

or system configurations, or that your pull request doesn't take all

platforms into account. Sometimes this important, sometimes it's not.

OpenTK is also (as previously mentioned) a very large codebase which has

seen a lot of people and a lot of styles over the years. This is

reflected in the deeper, darker parts of the codebase where mixed

styles, weird naming, bizarre code and eldritch sorcery abound. What may

seem like a small change on the surface could lead you down on a path of

unraveling one thread after another, and what started off as a simple

bug fix could transform into a lot of headscratching.

To make this at least somewhat easier, here's a few concrete general

tips which you should stick to:

\* Always consider cross-platform gotchas.

\* Always work in small, iterative chunks which you can easily describe.

\* It's a good idea to open your PR early, so that you can get quick

feedback. Tag it with "WIP" in the title.

\* Avoid cosmetic or visual changes, unless your contribution is strictly

focused on that.

\* Don't be afraid to ask, especially before diving in. There might be

someone else working on the very same thing already!

\* Consider how your contribution might affect other contributions.

Sometimes one change will break another if you're not careful.

In terms of these guidelines, the terminology is as follows:

\* Must: If your contribution does not follow this rule, it will not be

accepted.

\* Should: If your contribution does not follow this rule, it has a lower

chance of being accepted.

\* May: If your contribution does not follow this rule, it's probably not

going to matter that much. It'd be a nice touch, though.

With that in mind, check the following sections for more concrete and

direct guidelines.

## Setting Up

For first-time contributors, there are a few steps that you'll need to

go through in order to start contributing.

#### 1. Get a copy of the code

First, fork OpenTK to your own profile and clone a local copy.

```bash

$ git clone git@github.com:username/opentk.git

$ cd opentk

$ git remote add upstream https://github.com/opentk/opentk.git

```

#### 2. Create a working branch

Development is done against the `develop` branch - this is where all the

magic happens. Your changes should always be based on this branch, so in

order to start working, create a new branch with an appropriate name and

base it on `develop`.

```bash

$ git checkout -b my-branch -t origin/develop

```

#### 3. Let git know who you are

In order to better track changes and who does what, it's a good practice

to give git some information about yourself.

```bash

$ git config --global user.name "John Doe"

$ git config --global user.email "john.doe@example.com"

```

Optionally, you can also add your public GPG key and sign your commits -

that way, there is no question that it's definitely you that's created

the commit. GitHub has some excellent information on how to do this and

why it's a good idea - [Signing Commits With GPG][3].

```bash

$ git config --global user.signingkey QF3G6A39

$ git config --global commit.gpgsign true

```

#### 4. Commit changes

Once you've finished up a change, it's time to commit it. In doing so,

you'll be writing some sort of commit message, and there are some

guidelines for how this should be formatted. Primarily,

\* Keep the first line of the commit message 50 characters or less

\* Always keep the second line blank.

\* If you need a longer description, keep all subsequent more descriptive

lines at 72 characters or less.

The first line is what will be visible on the commit lists on GitHub, so

make sure it's as descriptive as you can make it.

#### 5. Synchronizing your changes

Sometimes, pull requests and code changes take time, and other

contributions are accepted in the meantime. When this happens, you'll

need to synchronize your changes with what's in the main repository.

This should be done using `rebase`, not `merge`, to keep the commit

history from being cluttered with merge commits.

If you've not pushed your changes anywhere yet, it's sufficient to

simply run (when on your branch)

```bash

$ git fetch upstream

$ git rebase upstream/develop

```

to fetch the latest code and replay your work on it. However, if you've

already pushed it, you might run into some issues when pushing to your

fork after rebasing. To get around this, you'll have to forcibly push

your changes to overwrite what's in your repository.

```bash

$ git fetch --all

$ git rebase upstream/develop

$ git push --force-with-lease origin my-branch

```

#### 6. Opening a pull request

When you feel that you're all done and you've pushed your changes to

git, it's time to open a pull request and have your changes reviewed.

Before doing so, run a final test by executing the build script in the

base directory of the codebase.

```bash

$ ./build.sh

```

If it executes without any problems, you're good to go and ready to move

on to creating your [Pull Request][4].

## Pull Requests

Pull requests are without a doubt one of the more involved contribution

types. Primarily, in order for a pull request to be accepted, it must

maintain a high quality, be well tested, and not have any major breaking

changes (unless absolutely necessary). There's going to be a lot of

stuff dumped on you in the next few paragraphs, but keep in mind that

most are \*guidelines\*, not hard rules. Stick to them as best you can,

and when in doubt - just ask.

All pull requests must have or do the following:

\* A clear, concise and descriptive title. As a rule of thumb, don't make

it longer than twelve words or 72 characters.

\* A clear and detailed description of what the pull request has changed.

This includes how the behaviour of the library will change if the pull

request is accepted - a maintainer should be able to read your

description and fully understand what accepting it would mean without

having to dive into the code.

\* Be based on the `develop` branch of the main OpenTK repository.

All pull requests should have the following:

\* If applicable, a compilable example which demonstrates the changes. A

git repository is preferred, and your changed branch should be included

as a submodule.

\* A short explanation of why you think these changes are necessary, if

it is not readily apparent from the rest of the pull request.

All code changes must follow these rules:

\* The style should be adhered to religiously. In general, this is the

same as following the MSDN and CoreFX guidelines with some changes. A

complete style guide is in the works.

\* All new methods, fields, properties, events, classes, structures and

enumerations must have appropriate XML documentation comments wherein

their behaviour is explained. These comments will be visible to the end

user, and should (in combination with the naming of the element) be

sufficient to fully understand what the element does.

\* XML comments on methods must describe each parameter (if any).

\* Changes to existing access modifiers should be avoided if at all

possible.

Furthermore, your pull request should:

\* Be tested on all applicable platforms. If you do not have access to a

platform (not owning a Windows license, not having a Mac on hand, not

having Linux installed, etc), ask for help testing your fix in Gitter or

in your pull request.

### Bug Fixes

Bug fixes should resolve a single reported issue, or a collection of

issues which fall under a single common meta-issue.

Your bug fix must:

\* Fix the issue on all supported platforms, or, if not applicable (such

as a platform-specific or platform-agnostic bug), make it clear that the

other platforms will not have the same issue.

\* Refer to the issue number using github's pound syntax - for instance,

"This PR resolves issue #1".

### New Features

New features should introduce a single feature, capability, or

functionality to the library which was not previously present. No more

than one feature may be introduced in any one pull request.

Your feature addition must:

\* Implement the feature on all supported platforms. If the feature

cannot be implemented on one platform for some reason, this must be

clearly explained in the pull request and documented in the source code.

Your feature addition should, if applicable and possible:

\* Implement a set of unit tests which test the entirety of the added

public API surface. These tests must pass on the CI service (Travis).

Furthermore, if your new feature replaces or makes an existing feature

obsolete, this must be clearly stated. This may prevent your pull

request from being accepted in the current development cycle, or it may

fast-track it, depending on the changes.

### Cosmetic & Stylistic Changes

Cosmetic and stylistic changes are those changes which do not affect

executing code - that is, the library operates exactly the same way

before and after change, but the code might look nicer or follow the

style better.

A cosmetic pull request must:

\* Not break any outstanding pull request, or, if both would modify the

same code, be prepared to wait until the other contribution is accepted

or rejected before being considered.

\* Change an affected file in its entirety to match the style guide

standard that the contribution is using. Mixed styles are not permitted.

As an example, if the pull request adds an XML comment to a method, it

should also comment all other code elements which do not have XML

documentation in that file.

Cosmetic contributions are not required to change everything in a file.

Single atomic cosmetic changes (such as applying a single rule from the

style guide) are permitted.

### Breaking Changes

A pull request is considered to have introduced a breaking change if it

does or wants to do one of the following:

\* Removes one or more public method, field, property, event, class,

structure or enumeration.

\* Renames one or more public method, field, property, event, class,

structure or enumeration.

\* Alters the public behaviour of an existing method or property without

fixing a bug or correcting the behaviour to an expected result.

\* Changes the accessibility of a previously public API to a more

restrictive accessibility.

\* Changes the method signature of a public method (renaming a parameter

does not constitute a breaking change, and is a cosmetic change).

These types of pull requests are difficult to handle, and are only

accepted as part of an active development cycle. Their contributions

will not make it into regular point releases, but can be merged into the

next major release.

In general, if your pull request introduces a breaking change, you

should follow this rule:

\* No public-facing API should be deleted or made inaccessible. Instead,

you should introduce an alternate method, field or property and mark the

previous one with an `[Obsolete("Use XXX instead.")]` attribute. Code

marked obsolete in the previous development cycle is deleted when a new

cycle begins.

## Bug Reports

Every bug report must follow the [Issue Template][5]. Reports which do

not follow this template will be closed.

If you can include a compilable example which demonstrates the issue

you're having, the chances that the bug will be fixed increase

substantially. It's a lot faster to work with a problem if you have

something that quickly shows you what's going wrong. As with pull

requests, a git repository is preferred. The OpenTK version that

exhibits the issue should be included as a package reference, either

from NuGet or MyGet.

One important thing - make sure that the problem is actually an issue

with OpenTK before opening a bug. It may be a driver issue if it's

graphical, or a library problem if it's input-related. It may be a

problem with your code, or it might be an issue in a library you use. As

with most things, asking for help on Gitter, Discord, or other related

forums will help you solve your problem faster and prevent invalid bug

reports from being opened.

## Discussions & Suggestions

We're always open to suggestions and discussions about current and

future features and goals of the library. Most of these discussions take

place on Gitter, but for larger projects and goals it might be a good

idea to create a github project tracker together with the maintainers.

If you think the discussion warrants a more permanent forum, talk to us

:)

[1]: https://gitter.im/opentk/opentk

[2]: https://discord.gg/GZTYR4s

[3]: https://help.github.com/articles/signing-commits-with-gpg/

[4]: #pull-requests

[5]: https://github.com/opentk/opentk/blob/develop/.github/ISSUE\_TEMPLATE.md