# Contributing to Fruit

This file contains various information and documentation for Fruit contributors.

If you only want to use Fruit, see the [wiki](https://github.com/google/fruit/wiki);

you can find instructions for building Fruit manually

[here](https://github.com/google/fruit/wiki/install#building-fruit-manually).

If you actually want to change Fruit itself, that's great! Read on.

### Basics

#### Build systems

Fruit supports two build systems: CMake (configured in `CMakeLists.txt` files) and

[Bazel](https://www.bazel.io) (configured in `BUILD` files).

This means that when you build/test Fruit code you have a choice of what build system you want to use,

but also that for larger changes (typically, if you add new files) you might need changes in both

`CMakeLists.txt` and `BUILD` files, to make sure that Fruit keeps building (and passing its tests) under both build

systems.

Both build systems are tested in Travis CI (see below).

Example commands to build a development version of Fruit using CMake (with all assertions enabled) and run the tests:

```bash

cd $PATH\_TO\_FRUIT

mkdir build-debug

cd build-debug

cmake .. -DCMAKE\_BUILD\_TYPE=Debug -DFRUIT\_ENABLE\_CLANG\_TIDY=TRUE -DCMAKE\_CXX\_FLAGS="-Werror -DFRUIT\_DEBUG=1 -DFRUIT\_EXTRA\_DEBUG=1 -D\_GLIBCXX\_DEBUG=1"

make -j 16

cd tests

py.test-3 -n auto

```

### Continuous Integration (CI)

Fruit uses Travis CI for continuous integration. You can see the latest CI runs in Travis CI

[here](https://travis-ci.org/google/fruit/builds). The CI configuration is defined in

`extras/scripts/travis\_yml\_generator.py`, that generates a `.travis.yml` file (which must also be checked in, due to the

way Travis CI is configured).

When editing the `travis\_yml\_generator.py` script you should also update the `.travis.yml` file (in the same commit)

by running:

```bash

cd $PATH\_TO\_FRUIT

extras/scripts/travis\_yml\_generator.py >.travis.yml

```

Fruit tests run in Travis CI in various configurations/environments, notably:

\* In Linux or OS X

\* In various Ubuntu versions

\* Using GCC or Clang

\* Optionally running under Valgrind

\* Optionally running with ASan/UBSan

\* Using CMake or Bazel

These tests run after every commit in master and for every pull request (as soon as the pull request is sent).

Linux tests run in Docker, using a set of images built for this purpose

([list of images](https://hub.docker.com/r/polettimarco/fruit-basesystem/tags/)).

If a test fails in Travis CI in some configuration, look at the beginning of the Travis CI Job log for a line such as:

```bash

export OS=linux; export COMPILER='clang-3.9'; export STL='libstdc++'; export UBUNTU='16.04'; extras/scripts/postsubmit.sh DebugValgrind

```

You can then run the same command locally (from your fruit directory) to reproduce the issue. Running this

`postsubmit.sh` script will run the tests under Docker to ensure repeatability of the results.

For example, even if the failure only happens with an old Ubuntu/GCC version you don't have installed, it will download

a Docker image containing that old Ubuntu/GCC and then run the tests inside a VM started from that image.

Once `postsubmit.sh` completes, if you want you can attach to the stopped VM used to run the tests by running:

```bash

docker attach fruit

```

This is often very useful to e.g. re-run a compilation manually with additional debug flags.

When running `postsubmit.sh` manually in this way, it will run using the latest changes in your fruit directory, even if

they aren't staged/committed yet. This allows to do a quicker edit/test cycle.

To speed up the execution of `postsubmit.sh` you can also set the `NJOBS` variable, e.g.:

```bash

export NJOBS=16; export OS=linux; export COMPILER='clang-3.9'; export STL='libstdc++'; export UBUNTU='16.04'; extras/scripts/postsubmit.sh DebugValgrind

```

The default number of jobs (used in Travis CI) is 2.

### How to run Fruit tests on Windows

You can import Fruit in Visual Studio (2017 and later) as a CMake project. You need to set the relevant CMake flags in

the `CMakeSettings.json` file that Visual Studio will create.

For example, if you installed Boost in `C:\boost\boost\_1\_62\_0`, you can put this configuration in your

`CMakeSettings.json`:

{

// See https://go.microsoft.com//fwlink//?linkid=834763 for more information about this file.

"configurations": [

{

"name": "x64-Debug",

"generator": "Visual Studio 16 2019 Win64",

"configurationType": "Debug",

"buildRoot": "${projectDir}\\out\\build\\${name}",

"cmakeCommandArgs": "-DBoost\_INCLUDE\_DIR=C:\\boost\\boost\_1\_62\_0 -DCMAKE\_BUILD\_TYPE=Debug -DFRUIT\_ADDITIONAL\_CXX\_FLAGS=/Z7",

"buildCommandArgs": "-m -v:minimal",

"intelliSenseMode": "windows-msvc-x64"

},

{

"name": "x64-Debug-noboost",

"generator": "Visual Studio 16 2019 Win64",

"configurationType": "Debug",

"buildRoot": "${projectDir}\\out\\build\\${name}",

"cmakeCommandArgs": "-DFRUIT\_USES\_BOOST=False -DCMAKE\_BUILD\_TYPE=Debug -DFRUIT\_ADDITIONAL\_CXX\_FLAGS=/Z7",

"buildCommandArgs": "-m -v:minimal",

"intelliSenseMode": "windows-msvc-x64"

}

]

}

The `/Z7` flag instructs Visual Studio to use the C7 format for debugging information, which allows Fruit's tests to run in parallel without interfering with each other.

If you don't want to use Boost, you can replace the `-DBoost\_INCLUDE\_DIR=...` flags above with `-DFRUIT\_USES\_BOOST=False`.

You can now run CMake within Visual Studio (from the menu: CMake -> Cache -> Generate -> CMakeLists.txt) and build Fruit (from the menu: CMake -> Build All).

You can also run tests, but \*only\* from the command-line (after building Fruit from Visual Studio), running tests from Visual Studio doesn't work.

To do that, you'll need python3 installed (you can download it [here](https://www.python.org/downloads/)).

You'll also some Python packages. You can install them with:

pip install absl-py

pip install pytest

pip install pytest-xdist

To do so:

\* Open the Start menu

\* From there, open the "Native Tools Command Prompt for VS 2017" shell for the chosen architecture. For example, "x64 Native Tools Command Prompt for VS 2017".

\* In Visual Studio, open the Output view (from the menu: View -> Output) and select "CMake" in the "Show output from:" dropdown menu.

\* Scroll to the beginning of that view. You should see two lines starting with "Command line" and "Working directory" respectively.

\* Cd to that working directory in the shell. For example, if the path in the "Working directory" line is `C:\Users\Marco\AppData\Local\CMakeBuild\fa17dda0-4eec-6438-a358-e1253b7e86ff\build\x64-Debug`, you can run `cd "C:\Users\Marco\AppData\Local\CMakeBuild\fa17dda0-4eec-6438-a358-e1253b7e86ff\build\x64-Debug"`.

\* Cd to the "tests" subdirectory ("cd tests").

\* Then run pytest, e.g. `py.test -n auto`.

### Sending pull requests

If you send a pull request, you should make sure that these CI tests are passing. They will run automatically on your

pull request as soon as you send it.

As an exception, if the current master also failed the last CI run feel free to send the pull request anyway (you can go

[here](https://travis-ci.org/google/fruit) to check if that's the case).

If a test fails, see the CI section above for informations on how to reproduce.

You should also make sure that your code:

\* Is formatted correctly ([more details here](#code-style))

\* Has appropriate tests (if your change is user-visible, or if you're introducing new branches that should be tested)

### What to install in order to develop Fruit code

In addition to

[the compiler you need to install to build Fruit](https://github.com/google/fruit/wiki/install#dependencies),

when developing Fruit code you might need some of the following software. Note that depending on your change you may or

may not need all of these; you might want to go ahead without these and then only install additional things if you get

an error about a missing tool.

\* CMake

\* Bazel ([installation instructions](https://www.bazel.io/docs/install.html))

\* Valgrind

\* Docker

## Useful command for fast edit/rebuild/retest cycles

This command uses Bazel to run the tests (so you need to have it installed in order to use this).

Bazel has a much more fine-grained picture of what tests depend on what source files, so it will often avoid running

tests that have passed before when it knows that they will pass (unlike py.test that runs the entire test suite every

time). This is especially relevant for incremental builds when only test sources have changed (e.g. after adjusting an

expectation in a test or fixing a bug in the test); there is little difference when changing `src/` or `include/`

because all tests will be re-run anyway.

```bash

cd $PATH\_TO\_FRUIT/extras/bazel\_root

bazel test --test\_output=errors \

--test\_summary=terse \

//third\_party/fruit/...

```

## Checking test coverage

Fruit's test suite supports collecting test coverage (only when building with GCC on Linux using CMake).

Example commands:

```bash

cd $PATH\_TO\_FRUIT

mkdir build-coverage

cd build-coverage

CXX=g++-6 cmake .. -DCMAKE\_BUILD\_TYPE=RelWithDebInfo -DFRUIT\_ENABLE\_COVERAGE=ON

make -j 10

(cd tests; py.test-3 -n auto)

lcov --rc lcov\_branch\_coverage=1 --capture --directory . --output-file coverage.info

lcov --rc lcov\_branch\_coverage=1 --remove coverage.info '/usr/include/\*' '/tmp/\*' -o coverage-filtered.info

genhtml --no-function-coverage --rc lcov\_branch\_coverage=1 --rc genhtml\_hi\_limit=100 coverage-filtered.info --output-directory html

google-chrome html/index.html

```

The important figures for each file are:

\* Percentage of lines covered

\* Percentage of branches covered

Ideally, they should both be 100%. The `LCOV\_EXCL\_LINE` and `LCOV\_EXCL\_BR\_LINE` markers can be used to mark lines and

branches (respectively) that can't be covered and therefore should be excluded.

Note that the "percentage of \*\*functions\*\* covered" metric is not meaningful for Fruit, since it considers each

instantiation of a template function/method as separate (even if they share the same source lines).

## Code style

C++ code in Fruit should be indented using clang-format (a `.clang-format` file is provided in the Fruit root

directory). You can re-indent all code using this command:

```bash

$ clang-format -i $(git ls-files | egrep '\.cpp|\.h' )

```

## Reporting vulnerabilities

In case of a security vulnerability in Fruit, please contact [poletti.marco@gmail.com](mailto:poletti.marco@gmail.com) directly instead of using the public issue tracker.