ChatGPT

PhysKit Project How-To Guide for New Contributors

Welcome to **PhysKit**, a 3D physics library for C++. This guide is designed to help new contributors work effectively in the repository. Follow these instructions closely to keep the codebase healthy and to avoid issues later in development.

1. Project Layout and Purpose

The repository uses a **modern C++** toolchain (C++26) and is built with **CMake**. The main library is compiled from sources listed in the `PHYSKIT_SOURCES` variable in the top-level `CMakeLists.txt` file [1]. The library exposes a namespaced API under `physkit`, with public headers in `include/physkit/` and implementation files in `src/`. Tests and examples live under the `tests/` directory; CMake adds them through `add_subdirectory(tests)` [2]. Documentation is generated with **Doxygen**, and the build system creates a `docs` target when Doxygen is found [3].

Directory Summary

Directory	Purpose
`include/`	Public headers. Each header corresponds to a logical module.
`src/`	Implementation files. Functions declared in headers must be defined here.
`tests/`	Test and example programs; organised in sub-directories (e.g., `demo`).
`docs/`	Contains the Doxygen configuration template (`Doxyfile.in`).
`.clang-tidy` & `.clang-format`	Configuration for static analysis and formatting.

Keep test code separate from library code. Public interfaces live in `include/`, while implementation details stay in `src/`. When adding files, update the relevant CMake variables and directories, as explained below.

2. Documenting Code with Doxygen

PhysKit uses **Doxygen** to extract API documentation from source files. The configuration template sets the input to the `include` directory and processes both header (`*.h`) and implementation (`*.cpp`) files recursively [4]. The generator creates HTML output and warns when functions lack documentation [5]. It also extracts all public symbols but not private or static members [6].

Writing Doxygen Comments

Use Doxygen tags consistently to produce clear documentation. Place documentation in the header file near the declaration. For example, the existing demo function is documented like this:

```
/// @brief A demo function that prints the PhysKit version.
/// @note This is a placeholder. Remove in future versions.
void demo();
```

Documentation should include:

- `@brief` to provide a one-sentence summary of the function's purpose.
- `@param` descriptions for each parameter (including units or valid ranges where appropriate).
- `@return` to describe the return value (omit for `void` functions).
- `@throws` when the function can throw exceptions.
- Additional tags such as `@note`, `@see`, or `@tparam` for templates as needed.

Use triple slashes (`///`) for one-line documentation or multiline `/** … */` blocks for longer descriptions. Avoid repeating implementation details; focus on *what* the function does and any pre/post-conditions. Run the docs target to generate HTML documentation (discussed below) and verify that your comments render correctly.

Generating API Documentation

Doxygen integration is optional but enabled by default. When you configure the project, CMake checks for Doxygen and, if found, creates a `docs` target [3]. After building the project, you can generate the documentation from the build directory:

```
cmake --build <build-dir> --target docs
```

The generated HTML files live in `<build-dir>/docs/html/index.html`. Open that file in a browser to view the documentation. If Doxygen is not installed, install it using your package manager (`brew install doxygen` on macOS, `sudo apt install doxygen` on Ubuntu, etc.).

3. Setting Up the Development Environment

PhysKit relies on the **Clang** toolchain for code completion, static analysis and formatting. Using the right tools early will save you from many stylistic and functional errors.

3.1 Clangd (Language Server)

Clangd provides intelligent code completion, jump-to-definition, refactoring hints and integrates `clang-tidy` checks. To set it up:

1. **Install clangd**. On Ubuntu: `sudo apt install clangd`; on macOS: `brew install llvm && brew link --force clangd`. Use version 15 or later.
2. **Generate a compile database**. Clangd needs a `compile_commands.json` file to understand your build options. Run CMake with the option to export this database:

```
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

This produces `build/compile_commands.json`. Either symlink it to the project root or configure your editor to use the one in the build directory.

1. **Install the clangd VS Code extension**. In VS Code, open the Extensions panel and search for *clangd*. Install it and configure its **clangd: Path** setting to point to your installed `clangd` binary if necessary (e.g., `/usr/bin/clangd` or `/opt/homebrew/opt/llvm/bin/clangd`).
2. **Enable the CMake Tools extension** (optional but recommended). This extension configures the build directory and generates the compile database automatically. After installing, open the command palette and run *CMake: Configure*. CMake Tools will ask for a build directory (e.g., `build`) and then generate the compile commands file for clangd.
3. Once configured, VS Code will provide real-time diagnostics from both the compiler and clang-tidy.

3.2 clang-tidy (Static Analysis)

`clang-tidy` runs static analysis checks that catch common bugs, performance issues and style violations. PhysKit uses a `.clang-tidy` configuration file at the repository root. It enables a broad set of checks (bug-prone, performance, modernize, readability and clang-analyzer) and disables only a few specific ones [7]. Identifier naming is enforced and treated as an error [8]; all identifiers must be lower case, while macro names are uppercase [9].

To run `clang-tidy` manually, use the compile database:

```
clang-tidy src/your_source.cpp -- -Iinclude
```

In VS Code with clangd, these checks appear as diagnostics in the editor. **Do not disable checks on your own.** If a particular rule seems inappropriate or noisy for our codebase, bring it up with the project lead (JC). Most checks are enabled deliberately to enforce good practices, and changes to this configuration should be discussed as a team.

3.3 clang-format (Automatic Formatting)

PhysKit enforces consistent formatting using `clang-format`. The formatting style is defined in `.clang-format` and is based on the LLVM style with several customisations: four-space indentation, Allman braces, a column width of 100 characters and sorted includes [10]. To format code:

```
clang-format -i include/physkit/new_file.h
clang-format -i src/new_file.cpp
```

In VS Code, install the *Clang-Format* extension or configure the *Editor: Default Formatter* setting to use clangd's built-in formatter. Enable **Format On Save** to automatically format your file whenever you save. Running `clang-tidy` after formatting can help ensure that style and naming rules are both satisfied.

4. Building the Project with CMake

4.1 Building from the Terminal

1. Create a build directory out of source:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

The `-S` flag specifies the source directory; `-B` specifies the build directory. The `compile_commands.json` file is generated here for clangd.

1. Build the library and tests:

```
cmake --build build
```

This compiles the `physkit` library from the files listed in `PHYSKIT_SOURCES` and builds any test executables added as subdirectories [1]. To build specific targets, use `--target physkit` or `--target demo`.

1. Run tests (once added) from the build directory. For example:

```
./build/tests/demo/demo
```

1. Generate documentation (if Doxygen is installed):

```
cmake --build build --target docs
```

The output will be placed in `build/docs/html` [3].

4.2 Using VS Code as an IDE

While the command line works fine, using **VS Code** with the **CMake Tools** extension simplifies many tasks. After installing the extension:

1. Open the project folder in VS Code.
2. Choose a compiler kit via *CMake: Select a Kit* (e.g., GCC or Clang).
3. Run *CMake: Configure* to generate the build system and compile commands. The extension automatically manages `build/` and runs configuration tasks whenever `CMakeLists.txt` changes.
4. Use *CMake: Build* to compile the project. Build results and errors appear in the Problems panel.
5. The extension also provides *CMake: Run Tests* and *CMake: Debug* for test executables.

An IDE reduces friction for novices by providing auto-completion, integrated terminals, and easy configuration. However, being comfortable with the terminal is still important when automating builds or working on continuous integration.

5. Adding New Library Files

When implementing new features in PhysKit, follow this process:

1. **Create header and source files** under the appropriate directories. For example, to implement a `vector3` utility, create `include/physkit/vector3.h` and `src/vector3.cpp`. Name files and identifiers in lower case to comply with the naming rules [8].

2. **Add the new source file to the build system.** Open `CMakeLists.txt` and append your source file to the `PHYSKIT_SOURCES` list [1]. For example:

```
set(PHYSKIT_SOURCES
    src/physkit.cpp
    src/vector3.cpp   # NEW
)
```

You do not need to list header files explicitly; they are included via `target_include_directories` [11].

1. **Document the new API** using Doxygen comments in the header. Write clear `@brief`, `@param`, and `@return` sections and include a usage example where appropriate.

2. **Format and analyse** your code. Run `clang-format -i` on both the header and source, then run `clang-tidy` or check for diagnostics in VS Code. Address all warnings or discuss them with the team.

3. **Write tests** for your feature (see below). Never merge new functionality without test coverage.

4. **Commit your changes** following the repository's commit message guidelines (if any) and open a pull request. CI checks will run `clang-tidy` and `clang-format` again.

6. Adding Tests

Tests reside under the `tests` directory. Each test or demo is placed in its own sub-directory with its own `CMakeLists.txt`. The top-level `tests/CMakeLists.txt` simply includes sub-directories with `add_subdirectory` [2]. To add a new test:

1. Create a sub-directory under `tests/`, e.g., `tests/vector3_test/`.

2. Inside that directory, create `CMakeLists.txt` and a test source file. A simple example:

```
add_executable(vector3_test
    main.cpp
)
target_link_libraries(vector3_test PRIVATE ${PHYSKIT_LIBRARIES})
```

This tells CMake to build the executable from `main.cpp` and link it against the PhysKit library.

1. Write your test in `main.cpp` or split it into multiple files. Use your preferred testing framework (e.g., Google Test) if available; otherwise simple assertions will do. Include necessary headers from PhysKit and exercise the API.

2. Go up one directory to `tests/CMakeLists.txt` and add your sub-directory:

```
add_subdirectory(vector3_test)
```

This ensures the test builds when you run `cmake --build`.

1. Configure and build the project again. Your test executable will be placed in `build/tests/vector3_test/`. Run it to verify that all assertions pass.

Organising tests this way keeps them modular and makes it easy to enable or disable individual tests. Always keep test code separate from library code.

7. Working with clang-tidy Issues

The `.clang-tidy` file defines the set of static analysis checks that run automatically [7]. These rules are in place to catch common pitfalls and enforce uniform naming conventions [12]. **Do not ignore or suppress warnings** unless the team lead approves. If a particular check seems incorrect, noisy or conflicts with the project's design, raise it with the experienced developer (JC). We can adjust the configuration or add justifications where necessary.

Fixing Common clang-tidy Violations

- **Naming conventions**: All function, method, variable, namespace and type names must be lower case [13]; macros use upper case [14]. Use underscores to separate words (e.g., `vector3_length`).
- **Performance issues**: Warnings such as `performance-unnecessary-value-param` suggest passing large objects by const reference instead of by value. Refactor accordingly.
- **Modernize suggestions**: The modernize module points out areas where newer C++ features could improve the code (e.g., `std::make_unique`, range-based for loops). Embrace these suggestions where they make the code clearer and safer, but discuss significant changes with the team.

When in doubt, ask for clarification. A short discussion can save hours of debugging later.

8. Summary

Working on PhysKit requires attention to detail and adherence to the development workflow. Use CMake and an IDE like VS Code to configure and build the project, rely on clangd for intelligent code navigation, let clang-tidy and clang-format enforce style and correctness, and generate documentation with Doxygen to help everyone understand the codebase. By following the guidelines in this document and communicating with your team, you'll contribute meaningful, maintainable code to PhysKit.

---

[1] [2] [3] [11] CMakeLists.txt
https://github.com/jcsq6/PhysKit/blob/main/CMakeLists.txt

[4] [5] [6] Doxyfile.in
https://github.com/jcsq6/PhysKit/blob/main/docs/Doxyfile.in

[7] [8] [9] [12] [13] [14] .clang-tidy
https://github.com/jcsq6/PhysKit/blob/main/.clang-tidy

[10] .clang-format
https://github.com/jcsq6/PhysKit/blob/main/.clang-format