Open CASCADE Technology
7.7.0

Guide to installing and using Git for OCCT development

November 2, 2022

Contents

1 Overview

1.1 Purpose

The purpose of this document is to provide a practical introduction to Git to OCCT developers who are not familiar with this tool and to facilitate the use of the official OCCT Git repository for code contribution to OCCT.

It can be useful to learn more about Git concepts and tools from a book a or manual. Many good books on Git can be found at https://git-scm.com/documentation

For the experienced Git users it can be enough to read sections 1 and 3 of this document to start working with the repository.

Familiarize yourselves with the Contribution Workflow document that describes how Git is used for processing contributions to OCCT.

This and related documents are available at the Resources page of the OCCT development portal at https://dev.opencascade.org/index.php?q=home/resources.

1.2 Git URL

URL of the official OCCT source code Git repository (accessed by SSH protocol) is:

```
gitolite@git.dev.opencascade.org:occt
```

or

```
ssh://gitolite@dev.opencascade.org/occt.git
```

1.3 Content

The official repository contains:

- The current certified version of OCCT: the "master" branch. This branch is updated by the Bugmaster only. Official OCCT releases are marked by tags.
- Topic branches created by contributors to submit changes for review / testing or for collaborative development. The topic branches should be named by the pattern "CR12345" where 12345 is the ID of the relevant issue registered in Mantis (without leading zeroes), and "CR" stands for "Change Request". The name can have an additional postfix used if more than one branch was created for the same issue.
- Occasionally topic branches with non-standard names can be created by the Bugmaster for special needs.

1.4 Short rules of use

The name specified in the user.name field in Git configuration should correspond to your login name on the OCCT development portal. This is important to clearly identify the authorship of commits. (The full real name can be used as well; in this case add the login username in parentheses.)

By default, contributors are allowed to push branches only with the names starting with CR (followed by the relevant Mantis issue ID). Possibility to work with other branches can be enabled by the Bugmaster on request.

The branch is created by the developer in his local repository when the development of a contribution starts. The branch for new developments is to be created from the current master. The branch for integration of patches or developments based on an obsolete version is created from a relevant tag or commit. The branch should be pushed to the official repo only when sharing with other people (for collaborative work or review / testing) is needed.

Rebasing the local branch to the current master is encouraged before the first submission to the official repository. If rebasing was needed after the branch is pushed to the official repo, the rebased branch should have a different name (use suffix).

Integration of contributions that have passed certification testing is made exclusively by the Bugmaster. Normally this is made by rebasing the contribution branch on the current master and squashing it into a single commit. This is made to have the master branch history plain and clean, following the general rule “one issue – one commit”. The description of the commit integrated to the master branch is taken from the Mantis issue (ID, 'Summary', followed by the information from 'Documentation' field if present).

In special cases when it is important to save the commits history in the branch (e.g. in case of a long-term development integration) it can be integrated by merge (no fast-forward).

The authorship of the contribution is respected by preserving the Author field of the commit when integrating. Branches are removed from the official repository when integrated to the master. The Bugmaster can also remove branches which have no commits during one-month period.

The Bugmaster may ask the developer (normally the one who produced the contribution) to rebase a branch on the current master, in the case if merge conflicts appear during integration.

1.5 Version of Git

The repository is tested to work with Git 1.7.6 and above. Avoid using versions below 1.7.1 as they are known to cause troubles.

2 Installing Tools for Work with Git

2.1 Windows platform

Installation of Git for Windows (provided by MSysGit project) is required.

In addition, it is recommended to install TortoiseGit to work with Git on Windows. If you do not install TortoiseGit or any other GUI tool, you can use GitGui and Gitk GUI tools delivered with Git and available on all platforms.

2.1.1 Installation of Git for Windows

Download Git for Windows distributive from https://git-for-windows.github.io/ During the installation:

- Check-in "Windows Explorer integration" options:
  - "Git Bash Here";
  - "Git GUI Here".
- To avoid a mess in your PATH, we recommend selecting "Run Git from Windows Prompt" in the environment settings dialog:
- In "Configuring the line ending conversions" dialog, select "Checkout Windows-style, commit Unix style endings".

Note that by default Git user interface is localized to the system default language. If you prefer to work with the English interface, remove or rename .msg localization file in subdirectories *share/git-gui/lib/msgs* and *share/gitk/lib/msgs* of the Git installation directory.

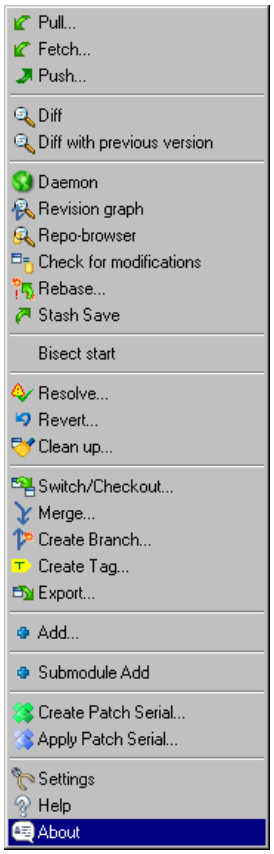
Before the first commit to the OCCT repository, make sure that your User Name in the Git configuration file (file *.gitconfig* in the *$HOME* directory) is equal to your username on the OCCT development portal.

2.1.2 Installation and configuration of TortoiseGit

Download TortoiseGit distributive from https://tortoisegit.org/download/. Launch the installation.

- Select your SSH client. Choose option
  - "OpenSSH, Git default SSH Client" if you prefer to use command-line tools for SSH keys generation, or
  - "TortoisePLink, coming from Putty, integrates with Windows better" if you prefer to use GUI tool (PuttyGen, see 3.2).
- Complete the installation.

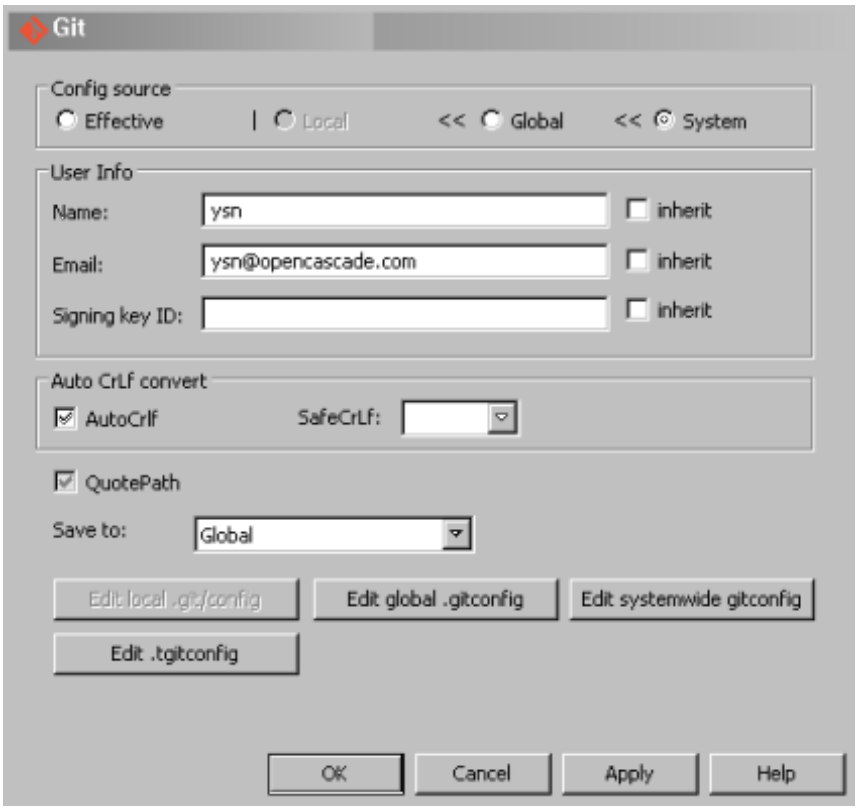
  TortoiseGit integrates into Windows Explorer, thus it is possible to use context menu in Windows Explorer to access its functionality:



Note that if you have installed MSysGit or have Git installed in non-default path, on the first time you use TortoiseGit you may get the message demanding to define path to Git. In such case, click on **Set MSysGit path** button and add the path to git.exe and path to MigGW libraries in the Settings dialog.

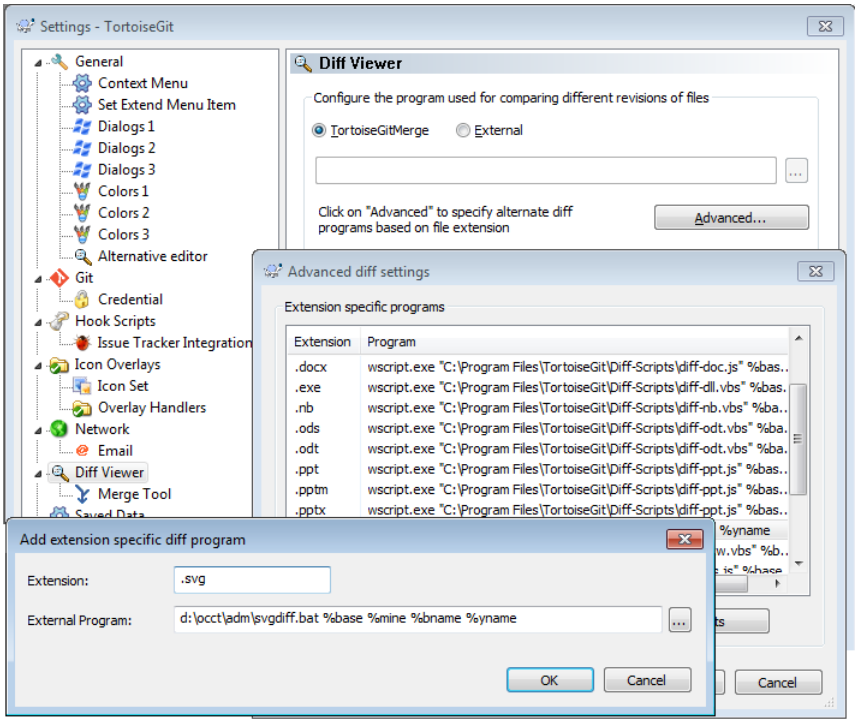
- After the installation select Start -> Programs -> TortoiseGit Settings to configure TortoiseGit.

Select Git->Config to add your user name and Email address to the local *.gitconfig* file



Optionally, you can set up TortoiseGit to use visual diff utility for SVG images used in OCCT documentation. For that, click on item "Diff Viewer" in the Settings dialog, then click button "Advanced..." in the right tab to add a new record with the following parameters:

- Extension: `.svg`
- External program: `<path_to_OCCT>\adm\svgdiff.bat %base %mine %bname %yname`



2.2 Linux platform

We assume that Linux users have Git already installed and available in the *PATH*.

Make sure to configure Git so that the user name is equal to your username on the OCCT development portal, and set SafeCrLf option to true:

```
> git config --global user.name "Your User Name"
> git config --global user.email your@mail.address
> git config --global your@mail.address
```

3 Getting access to the repository

3.1 Prerequisites

Access to the repository is granted to the users who have signed the Contributor License Agreement.

The repository is accessed by SSH protocol, thus you need to register your public SSH key on the development portal to get access to the repository.

SSH keys are used for secure authentication of the user when accessing the Git server. Private key is the one stored on the user workstation (optionally encrypted). Open (or public) key is stored in the user account page on the web site. When Git client accesses the remote repository through SSH, it uses this key pair to identify the user and acquire relevant access rights.

Normally when you have Git installed, you should have also SSH client available. On Unix/Linux it is installed by default in the system. On Windows it is typical to have several SSH clients installed; in particular they are included with Cygwin, Git, TortoiseGit.

It is highly recommended to use the tools that come with the chosen Git client for generation of SSH keys. Using incompatible tools (e.g. *ssh-keygen.exe* from Cygwin for code generation, and TortoiseGit GUI with a default Putty client for connection to server) may lead to authentication problems.

3.2 How to generate a key

3.2.1 Generating key with Putty

Use this option if you have installed TortoiseGit (or other GUI Git client on Windows) and have chosen “TortoiseP↩ Link” (or other Putty client) as SSH client during installation.

To generate the key with this client, run **Puttygen** (e.g. from Start menu -> TortoiseGit -> Puttygen), then click **Generate** and move mouse cursor over the blank area until the key is generated.

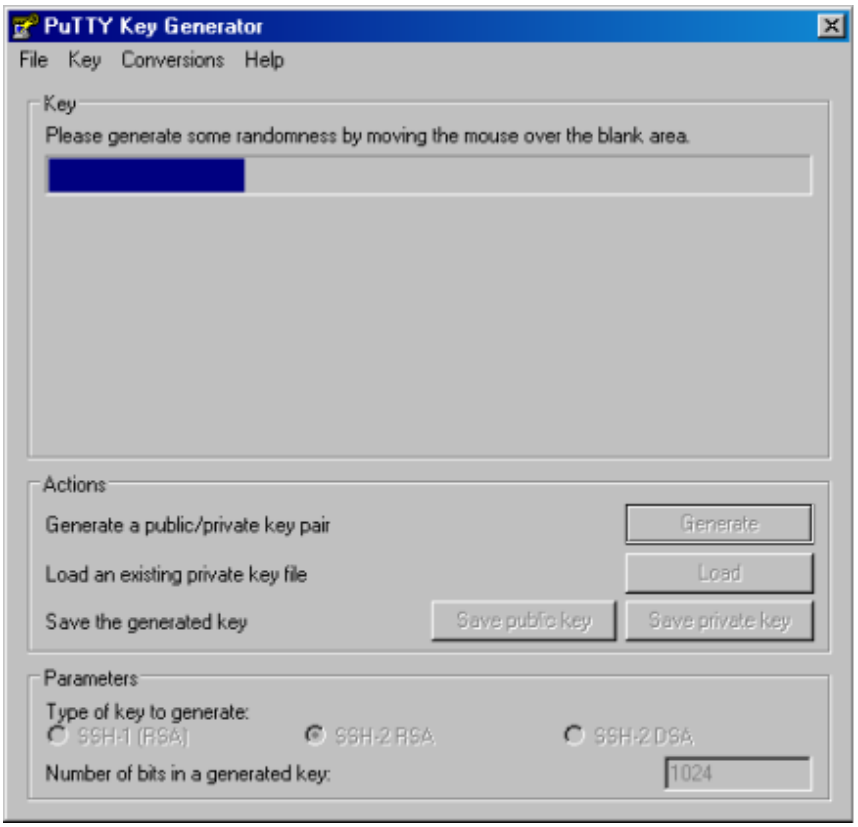


Figure 1: Putty key generator

After the key is generated, you will see GUI controls to define the public key comment and / or specify the password for the private key protection. When done, save both the public and the private key to the files of your choice (make sure to store your private key in a secure place!).

Copy the public key as shown by Puttygen to the clipboard to add it in your account. Do not copy the Putty public key file content – it is formatted in a way not suitable for the web site.

3.2.2 Generating key with command-line tools

Use this option if you work on Linux or if you have chosen “OpenSSH” as SSH client during installation of TortoiseGit (or other Windows tool).

Make sure that you have *ssh* and *ssh-keygen* commands in the path. On Windows, you might need to start **Git Bash** command prompt window.

Use the following command to generate SSH keys:

```
> ssh-keygen -t rsa -C "your@mail.address"
```

The last argument is an optional comment, which can be included with the public key and used to distinguish between different keys (if you have many). The common practice is to put here your mail address or workstation name.

The command will ask you where to store the keys. It is recommended to accept the default path *$HOME/.ssh/id_rsa*. Just press **Enter** for that. You will be warned if a key is already present in the specified file; you can either overwrite it by the new one, or stop generation and use the old key.

If you want to be on the safe side, enter password to encrypt the private key. You will be asked to enter this password each time you use that key (e.g. access a remote Git repository), unless you use the tool that caches the key (like TortoiseGit). If you do not want to bother, enter an empty string.

On Windows, make sure to note the complete path to the generated files (the location of your $HOME might be not obvious). Two key files will be created in the specified location (by default in $HOME/.ssh/):

- *id_rsa* – private key
- *id_rsa.pub* – public key

The content of the public key file (one text line) is the key to be added to the user account on the site (see below).

3.2.3 Generating key with Git GUI

GitGUI (standard GUI interface included with Git) provides the option to either generate the SSH key (if not present yet) or show the existing one. Click Help/Show SSH key and copy the public key content for adding to the user account page (see below).

3.3 Adding public key in your account

Log in on the portal `https://dev.opencascade.org` and click on **My account** link to the right. If you have a Contributor status, you will see **SSH keys** tab to the right.

Click on that tab, then click **Add a public key**, and paste the text of the public key (see above sections on how to generate the key) into the text box.

Click **Save** to input the key to the system.

Note that a user can have several SSH keys. You can distinguish between these keys by the Title field ID; by default it is taken from SSH key comment. It is typical to use your e-mail address or workstation name for this field; no restrictions are set by the portal.

Note that some time (5-10 min) is needed for the system to update the configuration after the new key is added. After that time, you can try accessing Git.

4 Work with repository: developer operations

4.1 General workflow

To start working with OCCT source repository, you need to create its clone in your local system. This cloned repository will manage your working copy of the sources and provide you the means to exchange code between your clone and the origin.

In most cases it is sufficient to have one clone of the repository; your working copy will be updated automatically by Git when you switch branches.

The typical development cycle for an issue is as follows:

- Create a new branch for your development, basing on the selected version of the sources (usually the current master) and switch your working copy to it
- Develop and test your change.
- Do as many commits in your branch as you feel convenient; the general recommendation is to commit every stable state (even incomplete), to record the history of your development.
- Push your branch to the repository when your development is complete or when you need to share it with other people (e.g. for review)
- Before the first push, rebase your local branch on the latest master; consider collapsing the history in one commit unless you think the history of your commits is interesting for others. Make sure to provide a good commit message.
- Do not amend the commits that have been already pushed in the remote repository, If you need to rebase your branch, commit the rebased branch under a different name, and remove the old branch.

You can switch to another branch at any moment (unless you have some uncommitted changes in the working copy) and return back to the branch when necessary (e.g. to take into account review remarks). Note that only the sources that are different between the switched branches will be modified, thus required recompilation should be reasonably small in most cases.

4.2 Cloning official repository

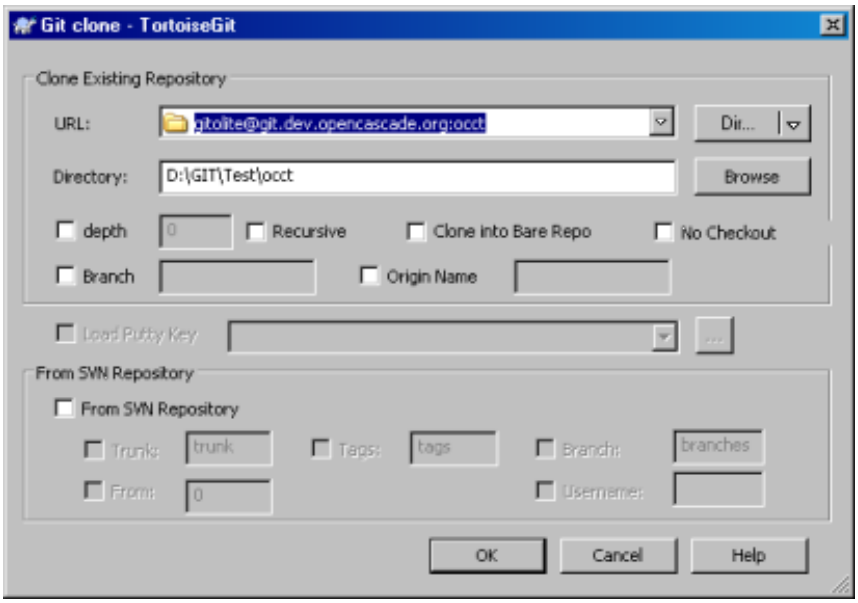
Clone the official OCCT repository in one of following ways:

- From command line by command:

```
> git clone gitolite@git.dev.opencascade.org:occt <path>
```

where <*path*> is the path to the new folder which will be created for the repository.

- In TortoiseGit: create a new folder, open it and right-click in the Explorer window, then choose **Git Clone** in the context menu:



If you have chosen Putty as SSH client during TortoiseGit installation, check the **Load Putty Key** option and specify the location of the private key file saved by PuttyGen (see 3.2.1). This shall be done for the first time only.

Note that on the first connection to the repository server you may be requested to enter a password for your private SSH key; further you can get a message that the authenticity of the host cannot be established and will be asked if you want to continue connecting or not. Choose **Yes** to continue. The host's key will be stored in *$HOME/.ssh/known_hosts* file.

4.3 Branch creation

You need to create a branch when you are going to start development of a new change, apply a patch, etc. It is recommended to fetch updates from the remote repository before this operation, to make sure you work with the up-to-date version.

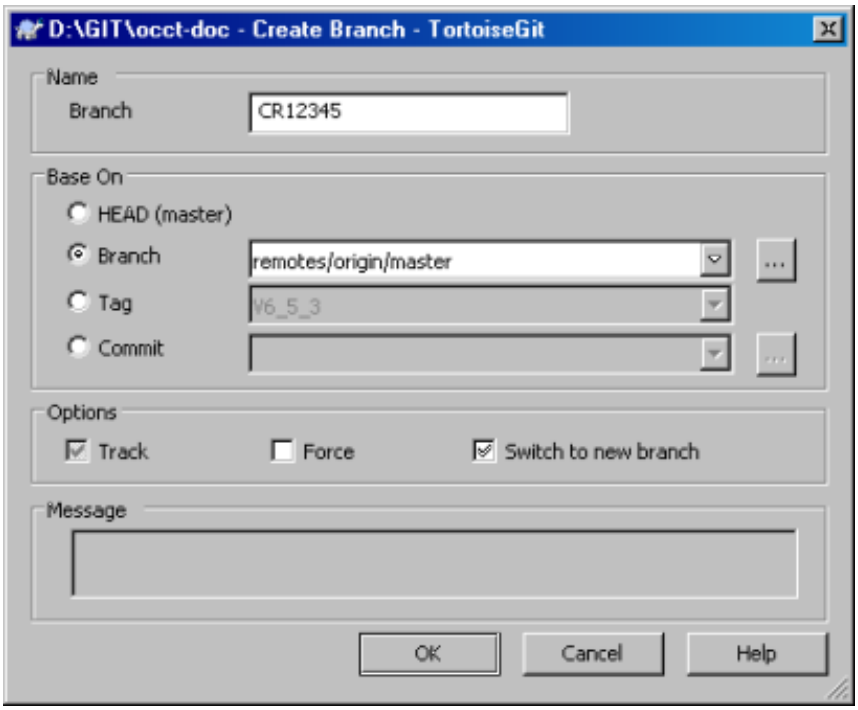
Create a branch from the current master branch unless you need to base your development on a particular version or revision.

In the console:

```
> git checkout -b CR12345 origin/master
```

In TortoiseGit:

- Go to the local copy of the repository.
- Right-click in the Explorer window, then choose **Git Create Branch**.
- Select **Base On** Branch *remotes/origin/master*.



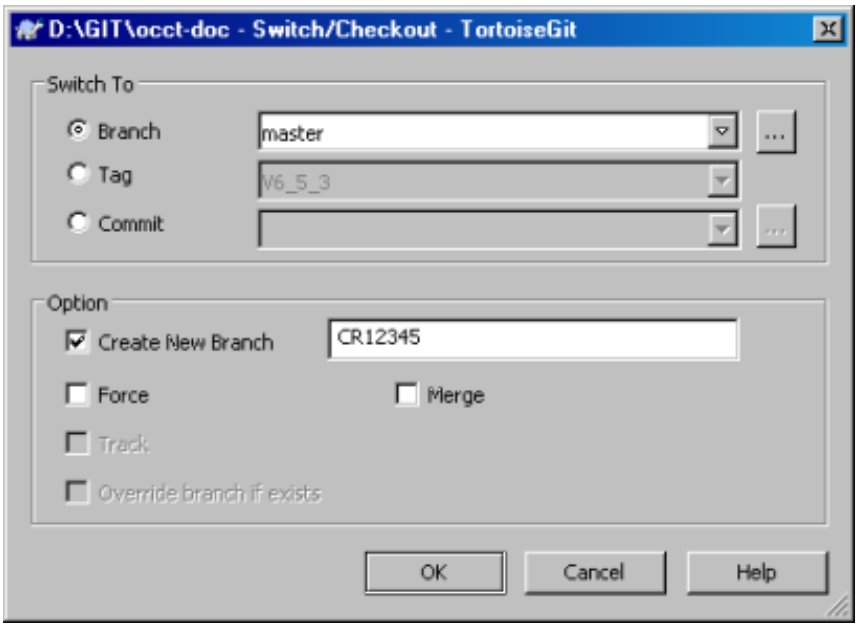
Check option **Switch to new branch** if you are going to start working with the newly created branch immediately.

4.4 Branch switching

If you need to switch to another branch, use Git command checkout for that. In the console:

```
> git checkout CR12345
```

In TortoiseGit: right-click in the explorer window and select in the context menu **TortoiseGit** -> **Switch/Checkout**.



Note that in order to work with the branch locally you need to set option **Create new branch** when you checkout the branch from the remote repository for the first time. Option **Track** stores association between the local branch and the original branch in a remote repository.

4.5 Committing branch changes

Commit your changes locally as soon as a stable status of the work is reached. Make sure to review carefully the committed changes beforehand to avoid unintentional commit of a wrong code.

- In the console:

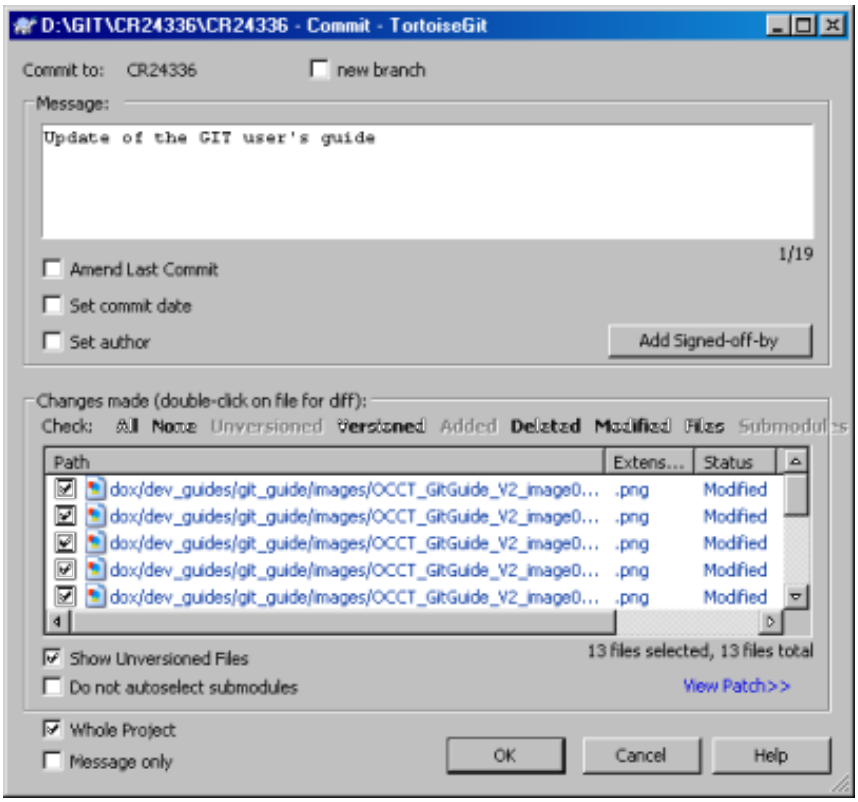
```
> git diff
...
> git commit -a -m "Write meaningful commit message here"
```

Option -a tells the command to automatically include (stage) files that have been modified or deleted, but it will omit the new files that might have been added by you. To commit such new files, you must add (stage) them before commit command.

To find new unstaged files and them to commit, use commands:

```
> git status -s
  ?? file1.hxx
  ?? file2.cxx
> git add file1.hxx file2.cxx
```

- In TortoiseGit: right-click in the explorer window and select in the context menu **Git Commit ->** **CR...**:



Unstaged files will be shown if you check the option 'Show Unversioned Files'. Double-click on each modified file to see the changes to be committed (as a difference vs. the base version).

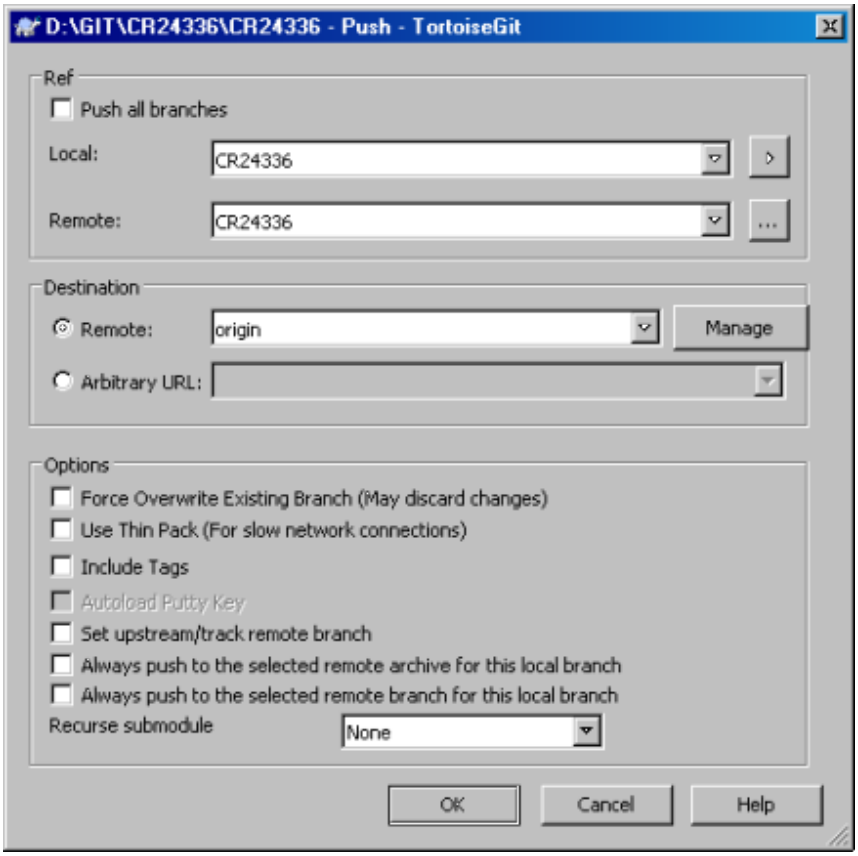
4.6 Pushing branch to the remote repository

When the code developed in your local branch is ready for review, or you need to share it with others, push your local changes to the remote repository.

- In the console:

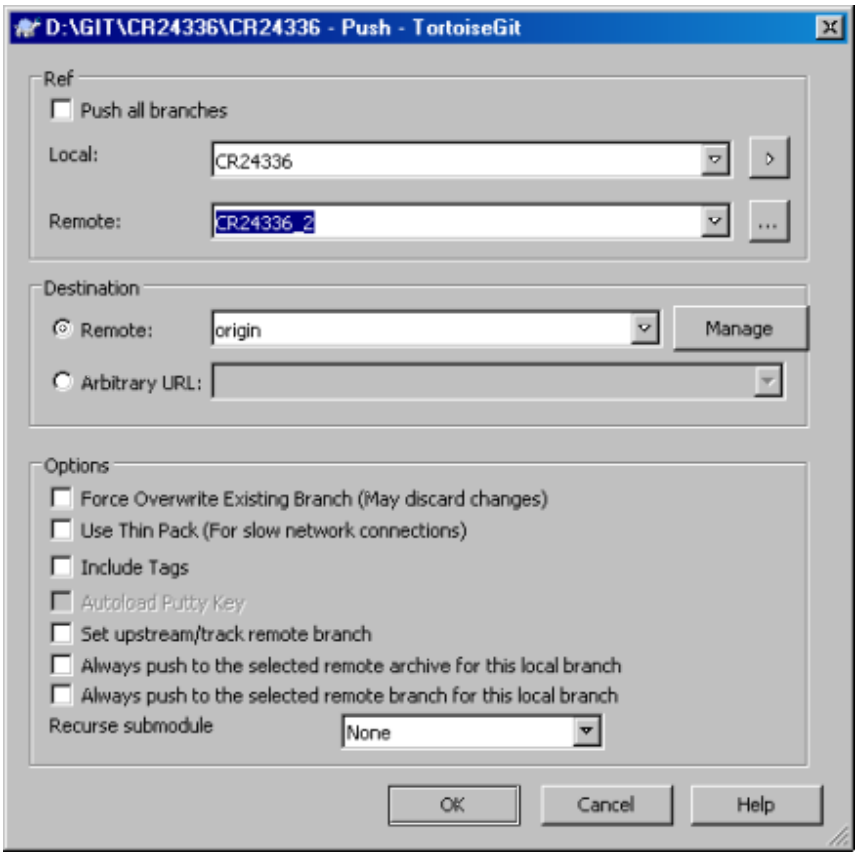
```
> git push "origin" CR12345:CR12345
```

- In TortoiseGit: right-click in the explorer window and select in the context menu, TortoiseGit -> **Push**



Note that Git forbids pushing a branch if the corresponding remote branch already exists and has some changes, which are not in the history of your local branch. This may happen in different situations:

- You have amended the last commit which is already in the remote repository. If you are sure that nobody else uses your branch, push again with **Force** option.
- You have rebased your branch, so that now it is completely different from the branch in the remote repository. In this case, push it under a different name (add a suffix):



Then remove the original remote branch so that other people recognize that it has been replaced by the new one. For that, select TortoiseGit -> **Push** again, select an empty line for your local branch name, and enter the name of the branch to be removed in **Remote** field:

- The other developer has committed some changes in the remote branch. In this case, **Pull** changes from the remote repository to have them merged with your version, and push your branch after it is successfully merged.

4.7 Synchronizing with remote repository

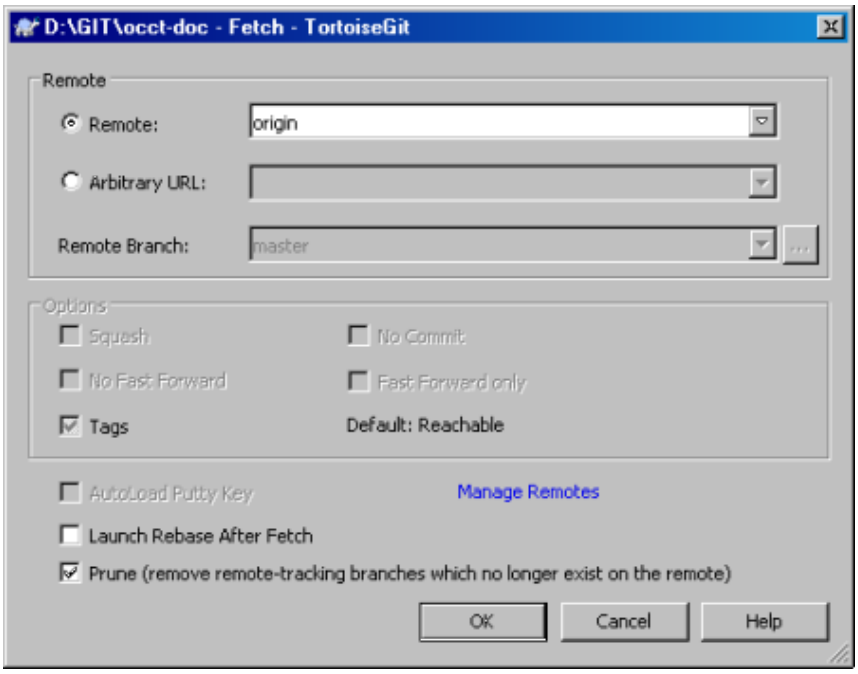
Maintain your repository synchronized with the remote one and clean unnecessary stuff regularly.

Use Git command *fetch* with option *prune* to get the update of all branches from the remote repository and to clean your local repository from the remote branches that have been deleted.

- In the console:

```
> git fetch --prune
```

- In TortoiseGit: right-click in the explorer window and select in the context menu **TortoiseGit** -> **Fetch**. Check in **Prune** check-box.



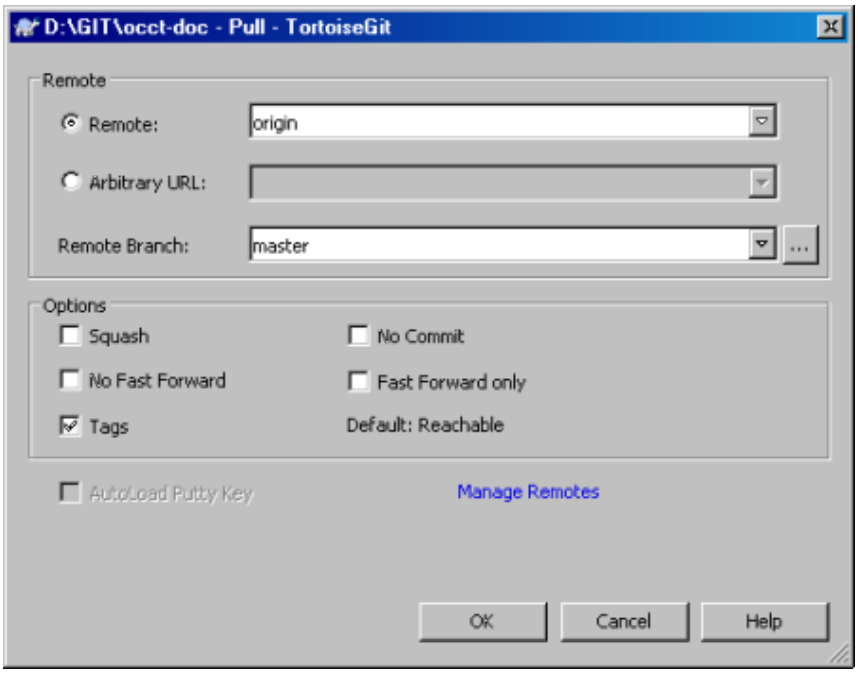
If the branch you are working with has been changed in the remote repository, use Git command *pull* to get the remote changes and merge them with your local branch.

This operation is required in particular to update your local master branch when the remote master changes.

- In console:

```
> git pull
```

- In TortoiseGit: right-click in the explorer window and select in the context menu **TortoiseGit** -> **Pull**.



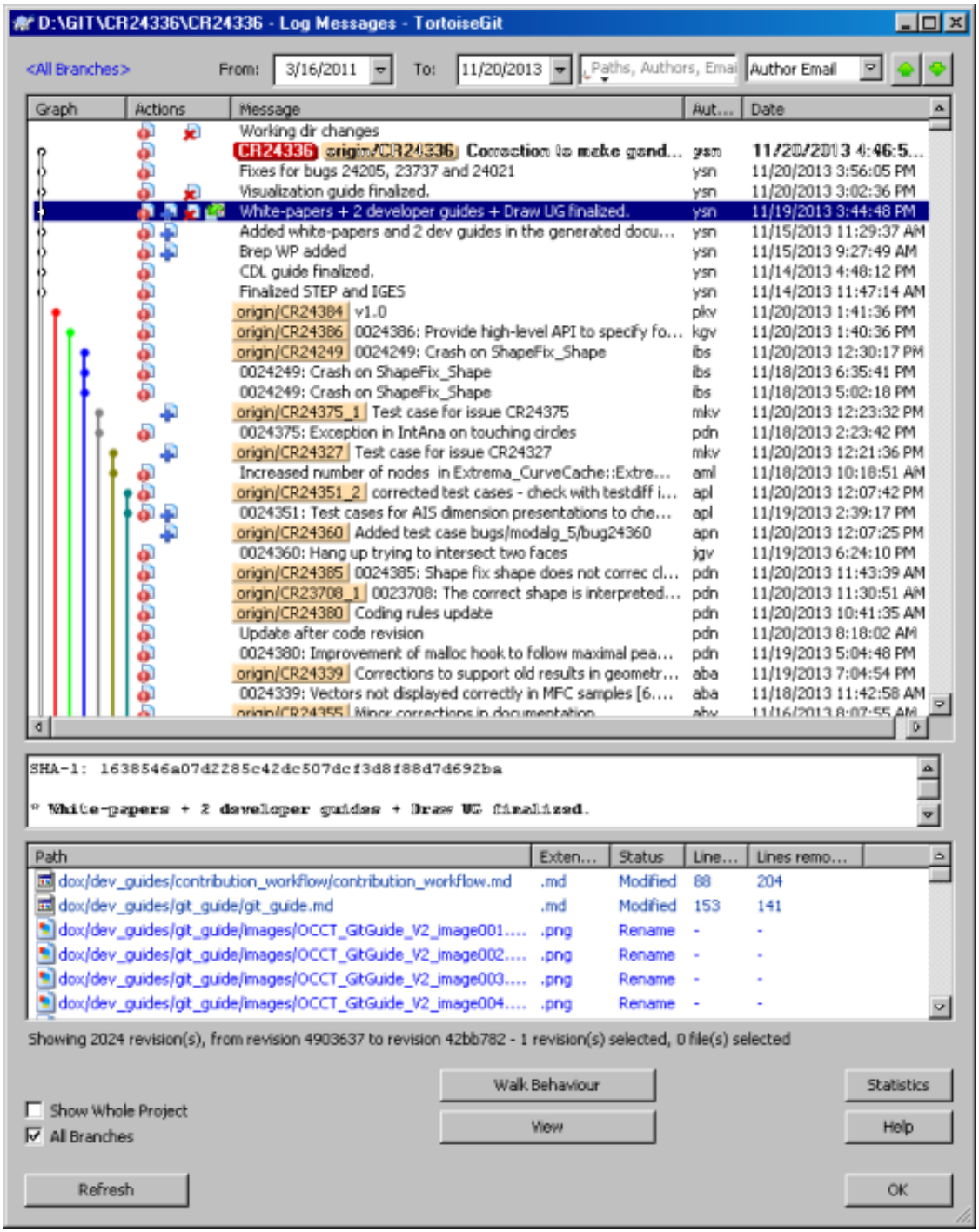
Note that the local branches of your repository are the primary place, where your changes are stored until they get integrated to the official version of OCCT (master branch). The branches submitted to official repository are for collaborative work, review, and integration – that repository should not be used for long-term storage of incomplete changes.

Remove the local branches that you do not need any more. Note that you cannot delete the current branch. It means that you need to switch to another one (e.g. master) if the branch you are going to delete is the current one.

- In the console:

```
> git branch -d CR12345
```

- In TortoiseGit: right-click in the explorer window and select in the context menu **TortoiseGit** -> **Git Show Log**.



Select **All branches** check-box to view all branches. Right-click on the branch you want to delete and select **Delete** item in the context menu.

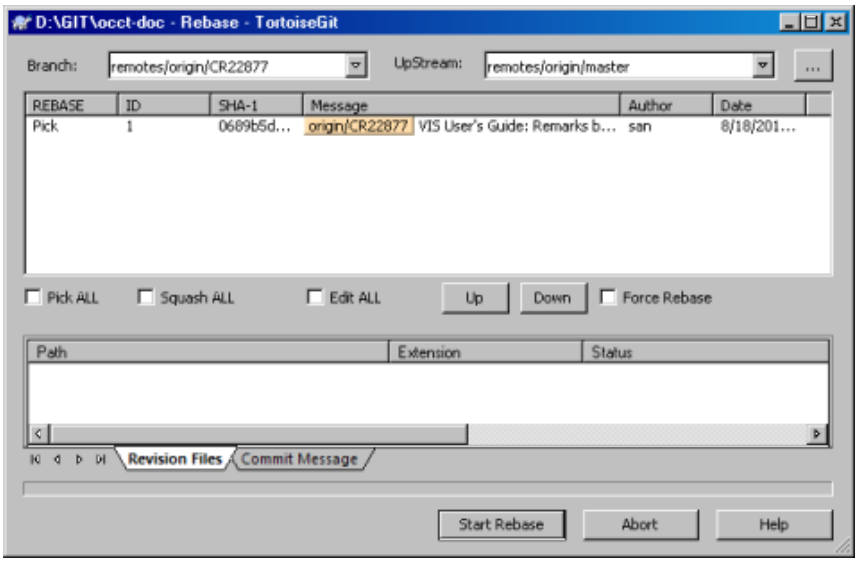
Note that many functions described above can be accessed from the Log View, which is a very convenient tool to visualize and manage branches.

4.8 Applying a fix made on older version of OCCT

If you have a fix made on a previous version of OCCT, perform the following sequence of operations to prepare it for testing and integration to the current development version:

- Identify the version of OCCT on which the fix has been made. In most cases, this will be an OCCT release, e.g. OCCT 6.7.0.
- Find a tag or a commit corresponding to this version in the Git history log of the master branch.
- Create a branch basing on this tag or commit. In TortoiseGit history log: right-click on the base commit, then select **Create branch at this version**.

- Check option **Switch to the new branch** to start working within the new branch immediately, or switch to it separately afterwards.
- Put your fix in the working copy, build and check that it works, then commit to the branch.
- Rebase the branch on the current master. In TortoiseGit: right-click on the working directory, choose **TortoiseGit** -> **Rebase**, select *remotes/origin/master* as UpStream revision, and click **Start**:



Note that you can get some conflicts during rebase. To resolve them, double-click on each conflicted file (highlighted by red in the file list) to open visual merge tool. Switch between conflicting fragments by red arrows, and for each one decide if the code of one or both conflicting versions is to be taken.

4.9 Rebasing with history clean-up

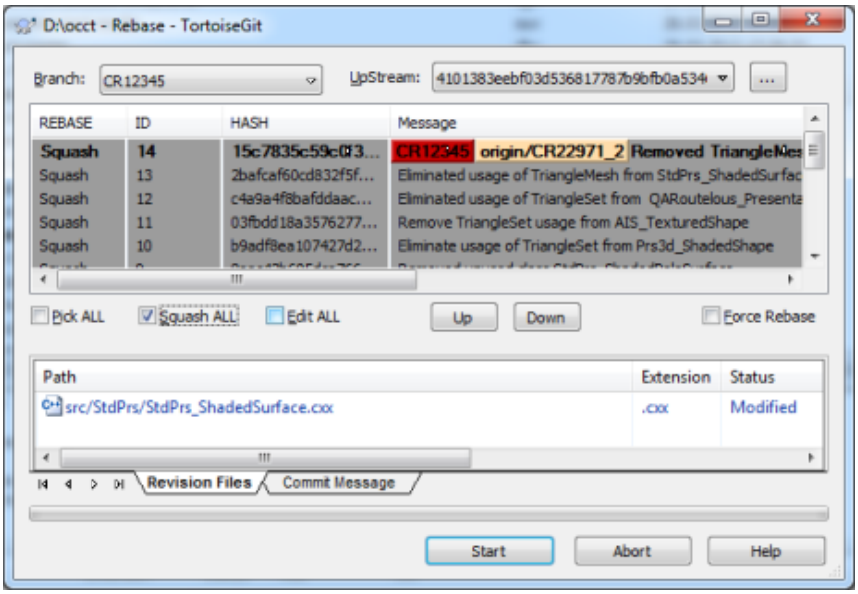
At some moments you might need to rebase your branch on the latest version of the master.

We recommend rebasing before the first submission of the branch for review or when the master has diverged substantially from your branch.

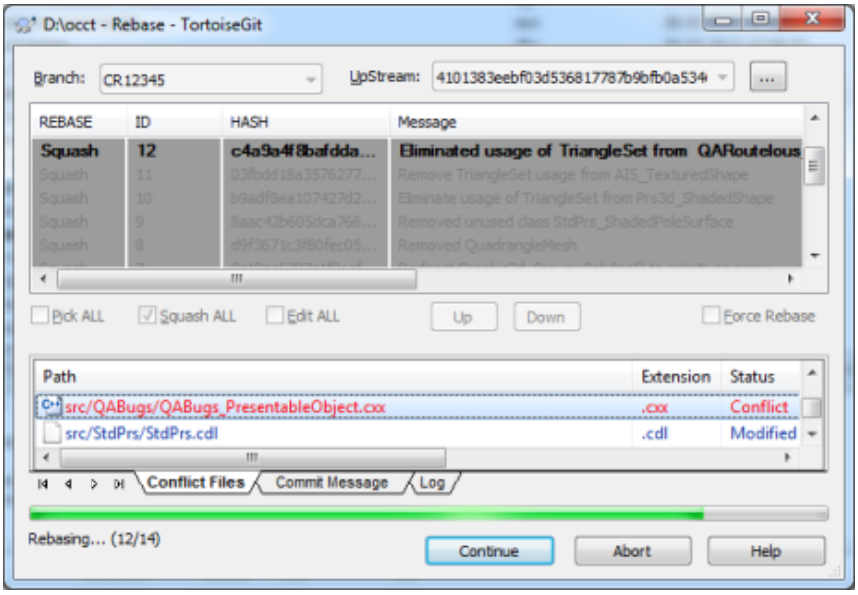
Rebasing is a good occasion to clean-up the history of commits in the branch. Consider collapsing (squashing, in terms of Git) the history of your branch into a single commit unless you deem that having separate commits is important for your future work with the branch or its code reviewing. Git also allows changing the order of commits, edit commit contents and messages, etc.

To rebase your branch into a single commit, you need to do the following:

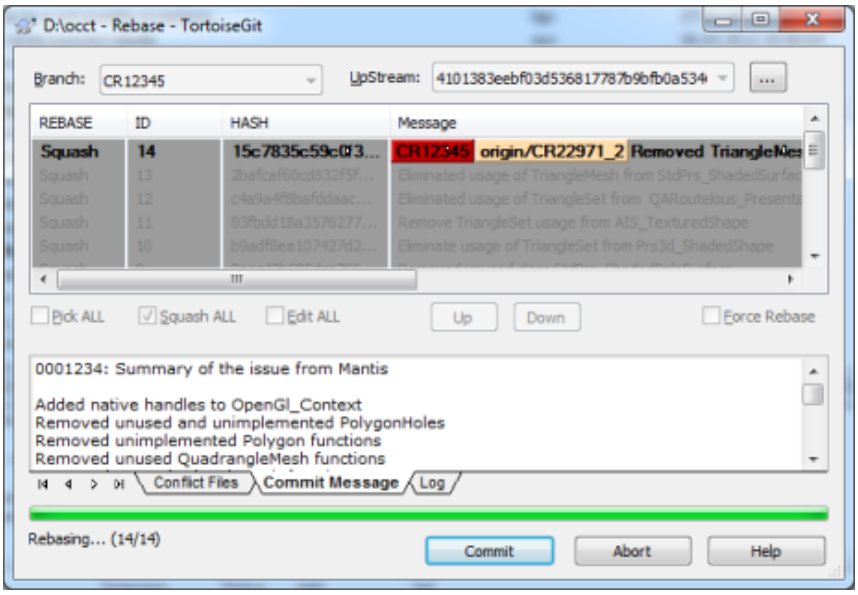
- Switch to your branch (e.g. “CR12345”)
- In TortoiseGit history log, select a branch to rebase on *(remotes/origin/master)* and in the context menu choose **Rebase “CR12345” onto this**.
- In the **Rebase** dialog, check **Squash All**. You can also change the order of commits and define for each commit whether it should be kept (**Pick**), edited, or just skipped.



- Click **Start**.
- The process will stop if a conflict is detected. In that case, find files with status **Conflicted** in the list (marked by red), and double-click on them to resolve the conflict. When all conflicts are resolved, click **Continue**.



- At the end of the process, edit the final commit message (it should start from the issue ID and a description from Mantis in the first line, followed by a summary of actual changes), and click **Commit**.


D:\occt - Rebase - TortoiseGit
Branch: CR12345
UpStream: 4101383eebf03d536817787b9bfb0a534
REBASE
ID
HASH
Message
Squash
14
CR12345
origin/CR22971_2
Pick ALL
Squash ALL
Edit ALL
Up
Down
Force Rebase
0001234: Summary of the issue from Mantis
Added native handles to OpenGl_Context
Removed unused and unimplemented PolygonHoles
Removed unimplemented Polygon functions
Removed unused QuadrangleMesh functions
Conflict Files
Commit Message
Log
Rebasing... (14/14)
Commit
Abort
Help

5 Work with repository: Reviewer operations

5.1 Review branch changes using GitWeb

The changes made in the branch can be reviewed without direct access to Git, using GitWeb interface:

- Open GitWeb in your web browser: https://git.dev.opencascade.org/gitweb/?p=occt.git
- Locate the branch you want to review among **heads** (click '...' at the bottom of the page to see the full list).
- Click **log** (or **shortlog**) to see the history of the branch.

Note that the branch can contain more than one commit, and you need to distinguish commits that belong to that branch (those to be reviewed) from the commits corresponding to the previous state of the master branch. Normally the first commit in the list that starts from the ID of the other issue indicates the branching point; commits above it are the ones to be reviewed.

- Click **commitdiff** on each log entry to review the changes (highlighted with color format).

5.2 Review branch changes with TortoiseGit

Use of TortoiseGit is recommended for convenient code review:

- Fetch the changes from the remote repository as described in Synchronizing with remote repository section.
- Right-click on the repository, choose **TortoiseGit** -> **Show** log;
- Locate the remote branch you need to review;
- To review commits one-by-one, select each commit in the log. The list of changed files is shown at the bottom of the window; double-click on the file will open visual compare tool.
- To review all changes made in the branch at once, or to compare two arbitrary revisions, select the corresponding commits in the log (e.g. the last commit in the branch and the branching point), ight-click for the context menu, and choose **Compare revisions**.