

Manual

This is the FreeCAD manual. It includes the essential parts out of the FreeCAD documentation wiki ([/wiki/index.php?title=Main_Page](http://wiki/index.php?title=Main_Page)). It is made primarily to be printed as one big document, so, if you are reading this online, you might prefer to head directly to the Online help ([/wiki/index.php?title=Online_Help_Toc](http://wiki/index.php?title=Online_Help_Toc)) version, which is easier to browse.



([/wiki/index.php?](http://wiki/index.php?title=File:Crystal_Clear_app_tutorials.png)

[title=File:Crystal_Clear_app_tutorials.png](http://wiki/index.php?title=File:Crystal_Clear_app_tutorials.png))

Welcome to the FreeCAD on-line help

This document has been automatically created from the contents of the official FreeCAD wiki documentation, which can be read online at http://www.freecadweb.org/wiki/index.php?title=Main_Page (http://www.freecadweb.org/wiki/index.php?title=Main_Page). Since the wiki is actively maintained and continuously developed by the FreeCAD community of developers and users, you may find that the online version contains more or newer information than this document. There you will also find in-progress translations of this documentation in several languages. But nevertheless, we hope you will find here all information you need. In case you have questions you can't find answers for in this document, have a look on the FreeCAD forum (<http://forum.freecadweb.org/index.php>), where you can maybe find your question answered, or someone able to help you.

How to use

This document is divided into several sections: introduction, usage, scripting and development, the last three address specifically the three broad categories of users of FreeCAD: end-users, who simply want to use the program, power-users, who are interested by the scripting capabilities of FreeCAD and would like to customize some of its aspects, and developers, who consider FreeCAD as a base for developing their own applications. If you are completely new to FreeCAD, we suggest you to start simply from the introduction.

Contribute

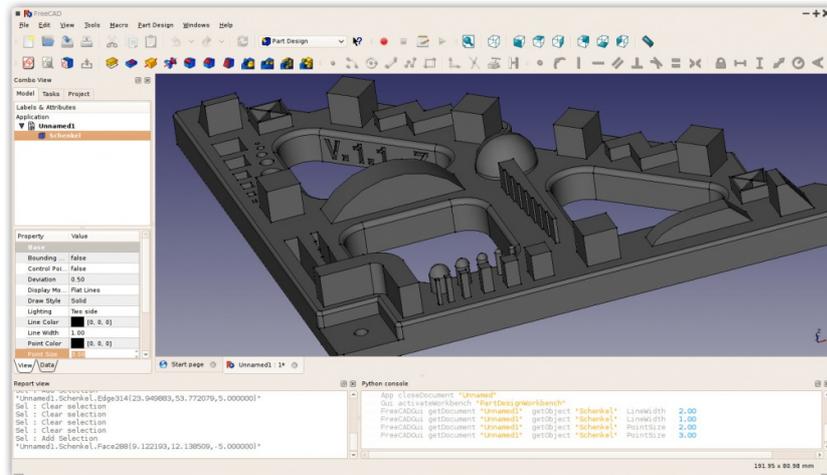
As you may have experienced sometimes, programmers are really bad help writers! For them it is all completely clear because they made it that way. Therefore it's vital that experienced users help us to write and revise the documentation. Yes, we mean you! How, you might ask? Just go to the Wiki at <http://www.freecadweb.org/wiki/index.php> (<http://www.freecadweb.org/wiki/index.php>) in the User section. You will need a FreeCAD wiki account to log in. Ask on the forum or on the irc channel for wiki write permission (the wiki is write-protected to avoid spamming) and we will create an account for you. Currently the wiki account is separate to the forum account but we will create the wiki account with the same name as your forum account. Then you can start editing! Also, check out the page at http://www.freecadweb.org/wiki/index.php?title=Help_FreeCAD (http://www.freecadweb.org/wiki/index.php?title=Help_FreeCAD) for more ways you can help FreeCAD.

< [previous: Online Help Toc \(/wiki/index.php?title=Online_Help_Toc\)](http://wiki/index.php?title=Online_Help_Toc)

[next: About FreeCAD > \(/wiki/index.php?title>About_FreeCAD\)](http://wiki/index.php?title>About_FreeCAD)

[Index \(/wiki/index.php?title=Online_Help_Toc\)](#)

Introduction



[\(/wiki/index.php?title=File:Freecad_default.jpg\)](#)

FreeCAD is a general purpose parametric 3D **CAD** (<http://en.wikipedia.org/wiki/CAD>) modeler. The development is completely Open Source (http://en.wikipedia.org/wiki/Open_source) (GPL License). FreeCAD is aimed directly at mechanical engineering (http://en.wikipedia.org/wiki/Mechanical_engineering) and product design (http://en.wikipedia.org/wiki/Product_design) but also fits in a wider range of uses around engineering, such as architecture or other engineering specialties.

FreeCAD features tools similar to Catia (<http://en.wikipedia.org/wiki/Catia>), SolidWorks (<http://en.wikipedia.org/wiki/Solidworks>) or Solid Edge (http://en.wikipedia.org/wiki/Solid_Edge), and therefore also falls into the category of **MCAD** (<http://en.wikipedia.org/wiki/CAD>), **PLM** (http://en.wikipedia.org/wiki/Product_Lifecycle_Management), **CAX** (<http://en.wikipedia.org/wiki/CAX>) and **CAE** (http://en.wikipedia.org/wiki/Computer-aided_engineering). It is a feature based parametric modeler (http://en.wikipedia.org/wiki/Parametric_feature_based_modeler) with a modular software architecture which makes it easy to provide additional functionality without modifying the core system.

As with many modern 3D CAD (<http://en.wikipedia.org/wiki/CAD>) modelers it has many 2D components in order to sketch 2D shapes or extract design details from the 3D model to create 2D production drawings, but direct 2D drawing (like AutoCAD LT (http://en.wikipedia.org/wiki/AutoCAD#AutoCAD_LT)) is not the focus, neither are animation or organic shapes (like Maya ([http://en.wikipedia.org/wiki/Maya_\(software\)](http://en.wikipedia.org/wiki/Maya_(software))), 3ds Max (http://en.wikipedia.org/wiki/3ds_Max), Blender (http://en.wikipedia.org/wiki/Blender_%28software%29) or Cinema 4D (http://en.wikipedia.org/wiki/CINEMA_4D)), although, thanks to its wide adaptability, FreeCAD might become useful in a much broader area than its current focus.

FreeCAD makes heavy use of all the great open-source libraries that exist out there in the field of Scientific Computing (http://en.wikipedia.org/wiki/Scientific_Computation). Among them are OpenCascade (<http://OpenCascade.org>), a powerful CAD kernel, Coin3D (<http://www.Coin3D.org>), an incarnation of Open Inventor (http://en.wikipedia.org/wiki/Open_Inventor), Qt (<http://www.qtsoftware.com/>), the world-famous UI framework, and Python (<http://www.python.org>), one of the best scripting languages available. FreeCAD itself can also be used as a library by other programs.

FreeCAD is also fully multi-platform (<http://en.wikipedia.org/wiki/Cross-platform>), and currently runs flawlessly on Windows and Linux/Unix and Mac OSX systems, with the exact same look and functionality on all platforms.

For more information about FreeCAD's capabilities, take a look at the Feature list (/wiki/index.php?title=Feature_list), the latest release notes (/wiki/index.php?title=Getting_started#What.27s_new) or the Getting started (/wiki/index.php?title=Getting_started) articles, or see more screenshots (</wiki/index.php?title=Screenshots>).

About the FreeCAD project

The FreeCAD project was started as far as 2001, as described in its history (</wiki/index.php?title=History>) page.

FreeCAD is maintained and developed by a community of enthusiastic developers and users (see the contributors (</wiki/index.php?title=Contributors>) page). They work on FreeCAD voluntarily, in their free time. They cannot guarantee that FreeCAD contains or will contain everything you might wish, but they will usually do their best! The community gathers on the FreeCAD forum (<http://forum.freecadweb.org>), where most of the ideas and decisions are discussed. Feel free to join us there!

< previous: Online Help Startpage (/wiki/index.php?title=Online_Help_Startpage)

Index next: Feature list > (/wiki/index.php?title=Feature_list)
(/wiki/index.php?title=Online_Help_Toc)

This is an extensive, hence not complete, list of features FreeCAD implements. If you want to look into the future see the Development roadmap (/wiki/index.php?title=Development_roadmap) for a quick overview of what's coming next. Also, the Screenshots (</wiki/index.php?title=Screenshots>) are a nice place to go.

Release notes

- Release 0.11 (/wiki/index.php?title=Release_notes_011) - March 2011
- Release 0.12 (/wiki/index.php?title=Release_notes_012) - December 2011
- Release 0.13 (/wiki/index.php?title=Release_notes_013) - January 2013
- Release 0.14 (/wiki/index.php?title=Release_notes_014) - March 2014
- Release 0.15 (/wiki/index.php?title=Release_notes_015) - March 2015
- Release 0.16 (/wiki/index.php?title=Release_notes_016) - April 2016

Key features

-  A complete Open CASCADE Technology (http://en.wikipedia.org/wiki/Open_CASCADE)-based **geometry kernel** allowing complex 3D operations on complex shape types, with native support for concepts like brep, nurbs curves and surfaces, a wide range of geometric entities, boolean operations and fillets, and built-in support of STEP and IGES formats
-  A full **parametric model**. All FreeCAD objects are natively parametric, which means their shape can be based on properties (</wiki/index.php?title=File:Feature3.jpg>)

title=Property) or even depend on other objects, all changes being recalculated on demand, and recorded by the undo/redo stack. New object types can be added easily, that can even be fully programmed in Python (/wiki/index.php?title=Scripted_objects)

- 

A **modular architecture** that allow plugins (modules) to add functionality to the core application. Those extensions can be as complex as whole new applications programmed in C++ or as simple as Python scripts (/wiki/index.php?title=Power_users_hub) or self-recorded macros (</wiki/index.php?title=Macros>). You have complete access from the **Python** built-in interpreter, macros or external scripts to almost any part of FreeCAD, being geometry creation and transformation (/wiki/index.php?title=Topological_data_scripting), the 2D or 3D representation of that geometry (scenegraph (</wiki/index.php?title=Scenegraph>)) or even the FreeCAD interface (</wiki/index.php?title=PySide>)
- 

Import/export to **standard formats** such as STEP (http://en.wikipedia.org/wiki/ISO_10303), IGES (<http://en.wikipedia.org/wiki/IGES>), OBJ (<http://en.wikipedia.org/wiki/Obj>), STL (http://en.wikipedia.org/wiki/STL_%28file_format%29), DXF (<http://en.wikipedia.org/wiki/Dxf>), SVG (<http://en.wikipedia.org/wiki/Svg>), STL ([http://en.wikipedia.org/wiki/STL_\(file_format\)](http://en.wikipedia.org/wiki/STL_(file_format))), DAE (<http://en.wikipedia.org/wiki/COLLADA>), IFC (http://en.wikipedia.org/wiki/Industry_Foundation_Classes) or OFF (<http://people.sc.fsu.edu/~jburkardt/data/off/off.html>), NASTRAN (<http://en.wikipedia.org/wiki/NASTRAN>), VRML (<http://en.wikipedia.org/wiki/VRML>) in addition to FreeCAD's native Fcstd file format (/wiki/index.php?title=Fcstd_file_format). The level of compatibility between FreeCAD and a given file format can vary, since it depends on the module that implements it.
- 

A **Sketcher** (</wiki/index.php?title=File:Feature7.jpg>) (/wiki/index.php?title=Sketcher_Workbench) with constraint-solver, allowing to sketch geometry-constrained 2D shapes. The sketcher currently allows you to build several types of constrained geometry, and use them as a base to build other objects throughout FreeCAD.
- 

A **Robot simulation** (</wiki/index.php?title=File:Feature9.jpg>) (/wiki/index.php?title=Robot_Workbench) module that allows to study robot movements. The robot module already has an extended graphical interface allowing GUI-only workflow.
- 

A **Drawing sheets** (</wiki/index.php?title=File:Feature8.jpg>) (/wiki/index.php?title=Drawing_Module) module that permit to put 2D views of your 3D models on a sheet. This modules then produces

ready-to-export SVG or PDF sheets. The module is still sparse but already features a powerful Python functionality.

-  A Rendering (</wiki/index.php?title=File:Feature-raytracing.jpg>)

(/wiki/index.php?title=Raytracing_Module) module that can export 3D objects for rendering with external renderers. Currently only supports povray (<http://en.wikipedia.org/wiki/POV-Ray>) and LuxRender (<http://en.wikipedia.org/wiki/LuxRender>), but is expected to be extended to other renderers in the future.

-  An Architecture (</wiki/index.php?title=File:Feature-arch.jpg>) (/wiki/index.php?title=Arch_Module)

module that allows BIM (http://en.wikipedia.org/wiki/Building_Information_Modeling)-like workflow, with IFC (http://en.wikipedia.org/wiki/Industry_Foundation_Classes) compatibility. The making of the Arch module is heavily discussed by the community here (<http://forum.freecadweb.org/viewtopic.php?f=10&t=821>).

General features

- **FreeCAD is multi-platform.** It runs and behaves exactly the same way on Windows Linux and Mac OSX platforms.
- **FreeCAD is a full GUI application.** FreeCAD has a complete Graphical User Interface based on the famous Qt (<http://www.qtsoftware.com/>) framework, with a 3D viewer based on Open Inventor (http://en.wikipedia.org/wiki/Open_Inventor), allowing fast rendering of 3D scenes and a very accessible scene graph representation.
- **FreeCAD also runs as a command line application,** with low memory footprint. In command line mode, FreeCAD runs without its interface, but with all its geometry tools. It can be, for example, used as server to produce content for other applications.
- **FreeCAD can be imported as a Python module (/wiki/index.php?title=Embedding_FreeCAD),** inside other applications that can run python scripts, or in a python console. Like in console mode, the interface part of FreeCAD is unavailable, but all geometry tools are accessible.
- **Workbench concept:** In the FreeCAD interface, tools are grouped by workbenches (</wiki/index.php?title=Workbenches>). This allows to display only the tools used to accomplish a certain task, keeping the workspace uncluttered and responsive, and the application fast to load.
- **Plugin/Module framework for late loading of features/data-types.** FreeCAD is divided into a core application and modules, that are loaded only when needed. Almost all the tools and geometry types are stored in modules. Modules behave like plugins, and can be added or removed to an existing installation of FreeCAD.
- **Parametric associative document objects:** All objects in a FreeCAD document can be defined by parameters. Those parameters can be modified on the fly, and recomputed anytime. The relationship between

objects is also stored, so modifying one object also modifies its dependent objects.

- **Parametric primitive creation** (box, sphere, cylinder, etc)
- Graphical **modification operations** like translation, rotation, scaling, mirroring, offset (trivial or after Jung/Shin/Choi (<http://www.ann.jussieu.fr/~frey/papers/meshing/Jung%20W.,%20Self-intersection%20removal%20in%20triangular%20mesh%20offsetting.pdf>)) or shape conversion, in any plane of the 3D space
- **Boolean operations** (http://en.wikipedia.org/wiki/Constructive_solid_geometry) (union, difference, intersect)
- Graphical creation of **simple planar geometry** like lines, wires, rectangles, arcs or circles in any plane of the 3D space
- Modeling with straight or revolution **extrusions, sections and fillets**.
- Topological components like **vertices, edges, wires and planes** (via python scripting).
- **Testing and repairing** tools for meshes: solid test, non-two-manifolds test, self-intersection test, hole filling and uniform orientation.
- **Annotations** like texts or dimensions
- **Undo/Redo framework**: Everything is undo/redoable, with access to the undo stack, so multiple steps can be undone at a time.
- **Transaction management**: The undo/redo stack stores document transactions and not single actions, allowing each tool to define exactly what must be undone or redone.
- **Built-in scripting** (</wiki/index.php?title=Scripting>) **framework**: FreeCAD features a built-in Python (<http://www.python.org/>) interpreter, and an API that covers almost any part of the application, the interface, the geometry and the representation of this geometry in the 3D viewer. The interpreter can run single commands up to complex scripts, in fact entire modules can even be programmed completely in Python.
- **Built-in Python console** with syntax highlighting, autocomplete and class browser: Python commands can be issued directly in FreeCAD and immediately return results, permitting scriptwriters to test functionality on the fly, explore the contents of the modules and easily learn about FreeCAD internals.
- **User interaction mirroring on the console**: Everything the user does in the FreeCAD interface executes python code, which can be printed on the console and recorded in macros.
- **Full macro recording & editing**: The python commands issued when the user manipulates the interface can then be recorded, edited if needed, and saved to be reproduced later.
- **Compound (ZIP based) document save format**: FreeCAD documents saved with `.fcstd` (/wiki/index.php?title=Fcstd_file_format) extension can contain many different types of information, such as geometry, scripts or thumbnail icons. The `.fcstd` file is itself a zip container, so a saved FreeCAD file has already been compressed.
- **Fully customizable/scriptable Graphical User Interface**. The Qt (<http://www.qtsoftware.com>)-based interface of FreeCAD is entirely accessible via the python interpreter. Aside from the simple functions

that FreeCAD itself provides to workbenches, the whole Qt framework is accessible too, allowing any operation on the GUI, such as creating, adding, docking, modifying or removing widgets and toolbars.

- **Thumbnailer** (Linux systems only at the moment): The FreeCAD document icons show the contents of the file in most file manager applications such as gnome's nautilus.
- **A modular MSI installer** allows flexible installations on Windows systems. Packages for Ubuntu systems are also maintained.

In development

-  An Assembly (/wiki/index.php?title=File:Feature-assembly.jpg) (/wiki/index.php?title=Assembly_project) module that allows to work with multiple projects, multiple shapes, multiple documents, multiple files, multiple relationships...
-  A Cam Module (/wiki/index.php?title=File:Feature-CAM.jpg) (/wiki/index.php?title=Cam_Module) dedicated to mechanical machining like milling, and will be able to output, display and adjust G code (<http://en.wikipedia.org/wiki/G-code>). This module is currently in planning state.

Extra Workbenches

Power users have created various custom external workbenches (/wiki/index.php?title=External_workbenches).

< previous: About FreeCAD (/wiki/index.php?title=About_FreeCAD)
 next: Install on Windows > (/wiki/index.php?title=Install_on_Windows)
 Index (/wiki/index.php?title=Online_Help_Toc)

Installation

Install on Windows

The easiest way to install FreeCAD on Windows is to download the installer below.

 (/wiki/index.php?title=File:Windows.png) Windows
 (<https://github.com/FreeCAD/FreeCAD/releases/download/0.16/FreeCAD.0.16.6704.c>
 WIN-x86_installer.exe) 32 bits  (/wiki/index.php?title=File:Windows.png)
 Windows
 (<https://github.com/FreeCAD/FreeCAD/releases/download/0.16/FreeCAD-0.16.6704.c>
 WIN-x64_Installer-1.exe) 64 bits

After downloading the .msi (Microsoft Installer) file, just double-click on it to start the installation process.

Below is more information about technical options. If it looks daunting, don't worry! Most Windows users will not need anything more than the .msi to install FreeCAD and **Get started** (/wiki/index.php?title=Getting_started)!

Simple Microsoft Installer Installation

The easiest way to **install FreeCAD on Windows** is by using the installer above. This page describes the usage and the features of the *Microsoft Installer* for more installation options.

If you would like to download either a 64 bit or unstable development version, see the Download (</wiki/index.php?title=Download>) page.

Command Line Installation

With the *msiexec.exe* command line utility, additional features are available, like non-interactive installation and administrative installation.

Non-interactive Installation

With the command line

```
msiexec /i FreeCAD<version>.msi
```

installation can be initiated programmatically. Additional parameters can be passed at the end of this command line, like

```
msiexec /i FreeCAD-2.5.msi TARGETDIR=r:\FreeCAD25
```

Limited user interface

The amount of user interface that installer displays can be controlled with */q* options, in particular:

- */qn* - No interface
- */qb* - Basic interface - just a small progress dialog
- */qb!* - Like */qb*, but hide the Cancel button
- */qr* - Reduced interface - display all dialogs that don't require user interaction (skip all modal dialogs)
- */qn+* - Like */qn*, but display "Completed" dialog at the end
- */qb+* - Like */qb*, but display "Completed" dialog at the end

Target directory

The property *TARGETDIR* determines the root directory of the FreeCAD installation. For example, a different installation drive can be specified with

```
TARGETDIR=R:\FreeCAD25
```

The default *TARGETDIR* is `[WindowsVolume\Programm Files\]FreeCAD<version>`.

Installation for All Users

Adding

```
ALLUSERS=1
```

causes an installation for all users. By default, the non-interactive installation install the package just for the current user, and the interactive installation offers a dialog which defaults to "all users" if the user is sufficiently privileged.

Feature Selection

A number of properties allow selection of features to be installed, reinstalled, or removed. The set of features for the FreeCAD installer is

- *DefaultFeature* - install the software proper, plus the core libraries
- *Documentation* - install documentation
- *Source code* - install the sources
- ... *ToDo*

In addition, ALL specifies all features. All features depend on DefaultFeature, so installing any feature automatically installs the default feature as well. The following properties control features to be installed or removed

- ADDLOCAL - list of feature to be installed on the local machine
- REMOVE - list of features to be removed
- ADDDEFAULT - list of features added in their default configuration (which is local for all FreeCAD features)
- REINSTALL - list of features to be reinstalled/repared
- ADVERTISE - list of feature for which to perform an advertise installation

There are a few additional properties available; see the MSDN documentation for details.

With these options, adding

```
ADDLOCAL=Extensions
```

installs the interpreter itself and registers the extensions, but does not install anything else.

Uninstallation

With

```
msiexec /x FreeCAD<version>.msi
```

FreeCAD can be uninstalled. It is not necessary to have the MSI file available for uninstallation; alternatively, the package or product code can also be specified. You can find the product code by looking at the properties of the Uninstall shortcut that FreeCAD installs in the start menu.

Administrative installation

With

```
msiexec /a FreeCAD<version>.msi
```

an "administrative" (network) installation can be initiated. The files get unpacked into the target directory (which should be a network directory), but no other modification is made to the local system. In addition, another (smaller) msi file is generated in the target directory, which clients can then use to perform a local installation (future versions may also offer to keep some features on the network drive altogether).

Currently, there is no user interface for administrative installations, so the target directory must be passed on the command line.

There is no specific uninstall procedure for an administrative install - just delete the target directory if no client uses it anymore.

Advertisement

With

```
msiexec /jm FreeCAD<version>.msi
```

it would be possible, in principle, to "advertise" FreeCAD to a machine (with /ju to a user). This would cause the icons to appear in the start menu, and the extensions to become registered, without the software actually being installed. The first usage of a feature would cause that feature to be installed.

The FreeCAD installer currently supports just advertisement of start menu entries, but no advertisement of shortcuts.

Automatic Installation on a Group of Machines

With Windows Group Policy, it is possible to automatically install FreeCAD an

a group of machines. To do so, perform the following steps:

1. Log on to the domain controller
2. Copy the MSI file into a folder that is shared with access granted to all target machines.
3. Open the MMC snapin "Active Directory users and computers"
4. Navigate to the group of computers that need FreeCAD
5. Open Properties
6. Open Group Policies
7. Add a new policy, and edit it
8. In Computer Configuration/Software Installation, choose New/Package
9. Select the MSI file through the network path
10. Optionally, select that you want the FreeCAD to be deinstalled if the computer leaves the scope of the policy.

Group policy propagation typically takes some time - to reliably deploy the package, all machines should be rebooted.

Installation on Linux using Crossover Office

You can install the windows version of FreeCAD on a Linux system using *CXOffice 5.0.1*. Run *msiexec* from the CXOffice command line, assuming that the install package is placed in the "software" directory which is mapped to the drive letter "Y:":

```
msiexec /i Y:\software\FreeCAD<version>.msi
```

FreeCAD is running, but it has been reported that the OpenGL display does not work, like with other programs running under Wine ([http://en.wikipedia.org/wiki/Wine_\(software\)](http://en.wikipedia.org/wiki/Wine_(software))) i.e. Google SketchUp (<http://en.wikipedia.org/wiki/SketchUp>).

< previous: Feature list (/wiki/index.php?title=Feature_list) Index
next: Install on Unix > (/wiki/index.php?title=Install_on_Unix)
(/wiki/index.php?title=Online_Help_Toc)

Install on Unix/Linux

The installation of FreeCAD on the most well-known linux systems has been now endorsed by the community, and FreeCAD should be directly available via the package manager available on your distribution. The FreeCAD team also provides a couple of "official" packages when new releases are made, and a couple of experimental PPA repositories for testing bleeding-edge features.

Once you've got FreeCAD installed, it's time to get started (/wiki/index.php?title=Getting_started)!

Ubuntu and Ubuntu-based systems

Many Linux distributions are based on Ubuntu and share its repositories. Besides official variants (Kubuntu, Lubuntu and Xubuntu), there are non official distros such as Linux Mint, Voyager and others. The installation options below should be compatible to these systems.

Official Ubuntu repository

FreeCAD is available from Ubuntu repositories and can be installed via the Software Center or with this command in a terminal:

```
sudo apt-get install freecad
```

But chances are this version will be outdated, and not have the latest features.

Latest Stable Release from the "stable releases" PPA or "daily" PPA

The FreeCAD community provides a PPA repository on Launchpad (<https://launchpad.net/~freecad-maintainers/+archive/freecad-stable>) with the latest **"stable"** FreeCAD version. There is also a more up to date **"daily"** PPA repository on Launchpad (<https://launchpad.net/~freecad-maintainers/+archive/freecad-daily>) automatically compiled daily from the official FreeCAD repository master branch, which will usually contain numerous bug fixes and feature updates.

Installing from the GUI

Add to your system's Software Sources the following PPA (read What are PPAs and how do I use them? (<http://askubuntu.com/questions/4983/what-are-ppas-and-how-do-i-use-them/5102#5102%29>) if you don't know how):

For the **"stable"** PPA

```
ppa:freecad-maintainers/freecad-stable
```

Or for the **"daily"** PPA

```
ppa:freecad-maintainers/freecad-daily
```

When a dialog window asks you to refresh your software sources, click OK.

Now you can install FreeCAD and FreeCAD documentation through the Ubuntu Software Center, or your package manager of choice.

Installing from the console

Type (or copy-paste) these commands in a console to add the PPA and install FreeCAD along with the documentation:

For the **"stable"** PPA

```
sudo add-apt-repository ppa:freecad-maintainers/freecad-stable
```

Or for the **"daily"** PPA

```
sudo add-apt-repository ppa:freecad-maintainers/freecad-daily
```

Then:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install freecad freecad-doc
```

Unstable version of FreeCAD

If you want to be on the bleeding edge of FreeCAD development, then use the **"daily"** PPA repository which provides daily builds (http://www.freecadweb.org/wiki/index.php?title=Download#Ubuntu_PPA_packages).

Debian and other debian-based systems

Since Debian Lenny, FreeCAD is available directly from the Debian software repositories and can be installed via synaptic or simply with:

```
sudo apt-get install freecad
```

OpenSUSE

FreeCAD is typically installed with:

```
zypper install FreeCAD
```

Gentoo

FreeCAD can be built/installed simply by issuing:

```
emerge freecad
```

Other

If you find out that your system features FreeCAD but is not documented in this page, please tell us on the forum (<http://forum.freecadweb.org/viewforum.php?f=21>)!

Many alternative, non-official FreeCAD packages are available on the net, for example for systems like slackware or fedora. A search on the net can quickly give you some results.

Manual install on .deb based systems

If for some reason you cannot use one of the above methods, you can always download one of the .deb packages available on the Download (</wiki/index.php?title=Download>) page.



(</wiki/index.php?title=File:Linux.png>)

Ubuntu

(<https://launchpad.net/~freecad-maintainers/+archive/freecad-stable>)
32/64bit

Once you downloaded the .deb corresponding to your system version, if you have the Gdebi (<http://en.wikipedia.org/wiki/Gdebi>) package installed (usually it is), you just need to navigate to where you downloaded the file, and double-click on it. The necessary dependencies will be taken care of automatically by your system package manager. Alternatively you can also install it from the terminal, navigating to where you downloaded the file, and type:

```
sudo dpkg -i Name_of_your_FreeCAD_package.deb
```

changing Name_of_your_FreeCAD_package.deb by the name of the file you downloaded.

After you installed FreeCAD, a startup icon will be added in the "Graphic" section of your Start Menu.

Installing on other Linux/Unix systems

Unfortunately, at the moment, no precompiled package is available for other Linux/Unix systems, so you will need to compile FreeCAD yourself (</wiki/index.php?title=CompileOnUnix>).

Installing Windows Version on Linux

See the Install on Windows (/wiki/index.php?title=Install_on_Windows) page.

< previous: Install on Windows (/wiki/index.php?title=Install_on_Windows)

Index next: Install on Mac > (/wiki/index.php?title=Install_on_Mac)
(/wiki/index.php?title=Online_Help_Toc)

Install on Mac

FreeCAD can be installed on Mac OS X in one step using the Installer.



(</wiki/index.php?title=File:Mac.png>)

Mac OS X

(https://github.com/FreeCAD/FreeCAD/releases/download/0.16/FreeCAD_0.16-6705.acfe417-OSX-x86_64.dmg) 10.9 Mavericks 64-bit

New Mac download link

This page describes the usage and features of the FreeCAD installer. It also includes uninstallation instructions. Once installed, you can get started (/wiki/index.php?title=Getting_started)!

Simple Installation

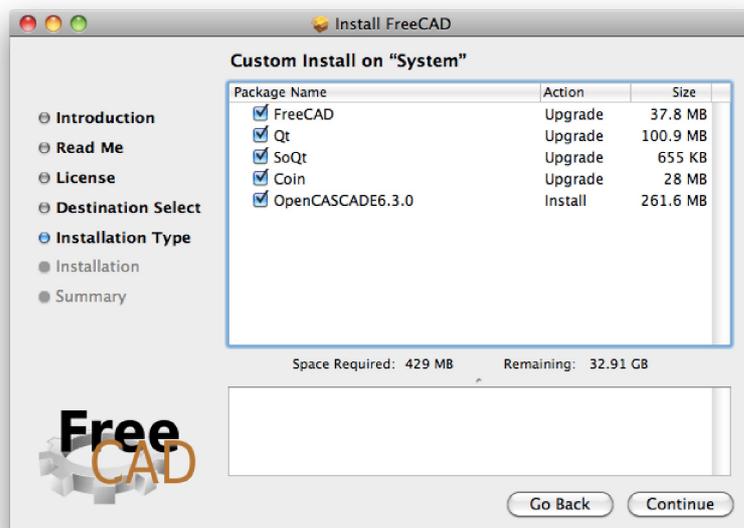
The FreeCAD installer is provided as a Installer package (.mpkg) enclosed in a disk image file.

You can download the latest installer from the Download (</wiki/index.php?title=Download>) page. After downloading the file, just mount the disk image, then run the **Install FreeCAD** package.



(/wiki/index.php?title=File:Mac_installer_1.png)

The installer will present you with a **Customize Installation** screen that lists the packages that will be installed. If you know that you already have any of these packages, you can deselect them using the checkboxes. If you're not sure, just leave all items checked.



(/wiki/index.php?title=File:Mac_installer_2.png)

Uninstallation

There currently isn't an uninstaller for FreeCAD. To completely remove FreeCAD and all installed components, drag the following files and folders to the Trash:

- In /Applications:
 - FreeCAD
- in /Library/Frameworks/
 - SoQt.framework
 - Inventor.framework

Then, from the terminal, run:

```
sudo /Developer/Tools/uninstall-qt.py
sudo rm -R /usr/local/lib/OCC
sudo rm -R /usr/local/include/OCC
```

That's it. Eventually, FreeCAD will be available as a self-contained application bundle so all this hassle will go away.

< previous: Install on Unix (/wiki/index.php?title=Install_on_Unix)
next: Getting started > (/wiki/index.php?title=Getting_started)
Index (/wiki/index.php?title=Online_Help_Toc)

Discovering FreeCAD

What's new

- [Version 0.11 Release notes \(/wiki/index.php?title=Release_notes_011\)](/wiki/index.php?title=Release_notes_011):
Check what's new in the 0.11 release of FreeCAD
- [Version 0.12 Release notes \(/wiki/index.php?title=Release_notes_012\)](/wiki/index.php?title=Release_notes_012):
Check what's new in the 0.12 release of FreeCAD
- [Version 0.13 Release notes \(/wiki/index.php?title=Release_notes_013\)](/wiki/index.php?title=Release_notes_013):
Check what's new in the 0.13 release of FreeCAD
- [Version 0.14 Release notes \(/wiki/index.php?title=Release_notes_014\)](/wiki/index.php?title=Release_notes_014):
Check what's new in the 0.14 release of FreeCAD
- [Version 0.15 Release notes \(/wiki/index.php?title=Release_notes_015\)](/wiki/index.php?title=Release_notes_015):
Check what's new in the 0.15 release of FreeCAD
- [Version 0.16 Release notes \(/wiki/index.php?title=Release_notes_016\)](/wiki/index.php?title=Release_notes_016):
Check what's new in the 0.16 release of FreeCAD

Foreword

FreeCAD is a 3D CAD/CAE parametric modeling application (/wiki/index.php?title>About_FreeCAD). It is primarily made for mechanical design, but also serves all other uses where you need to model 3D objects with precision and control over modeling history.

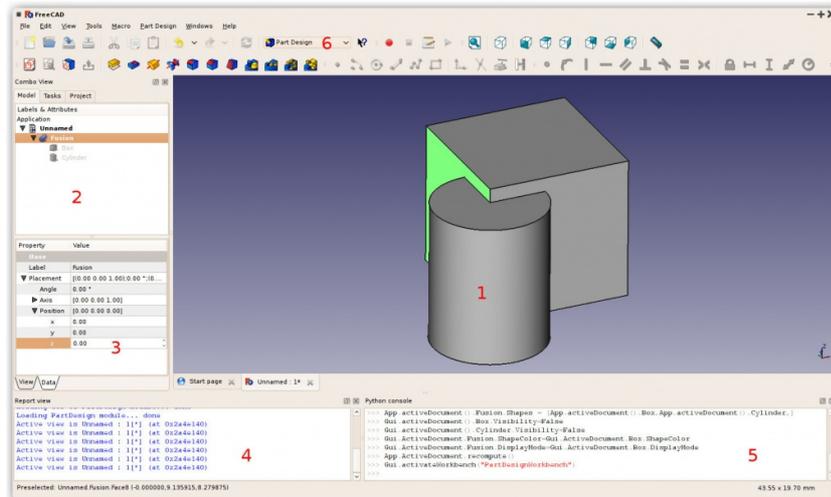
FreeCAD is still in the early stages of development, so, although it already offers you a large (and growing) list of features (/wiki/index.php?title=Feature_list), much is still missing, specially comparing it to commercial solutions, and you might not find it developed enough yet for use in production environment. Still, there is a fast-growing community (<http://forum.freecadweb.org/index.php>) of enthusiastic users, and you can already find many examples (<http://forum.freecadweb.org/viewtopic.php?f=8&t=1222>) of quality projects developed with FreeCAD.

Like all open-source projects, the FreeCAD project is not a one-way work delivered to you by its developers. It depends much on its community to grow, gain features, and stabilize (get bugs fixed). So don't forget this when starting to use FreeCAD, if you like it, you can directly influence and help (/wiki/index.php?title=Help_FreeCAD) the project!

Installing

First of all (if not done already) download and install FreeCAD. See the Download (</wiki/index.php?title=Download>) page for information about current versions and updates, and the Installing (</wiki/index.php?title=Installing>) page for information about how to install FreeCAD. There are install packages ready for Windows (.msi), Ubuntu & Debian (.deb) openSUSE (.rpm) and Mac OSX. As FreeCAD is open-source, if you are adventurous, but want to have a look at the brand-new features being developed right now, you can also grab the source code and compile (</wiki/index.php?title=Compiling>) FreeCAD yourself.

Exploring FreeCAD



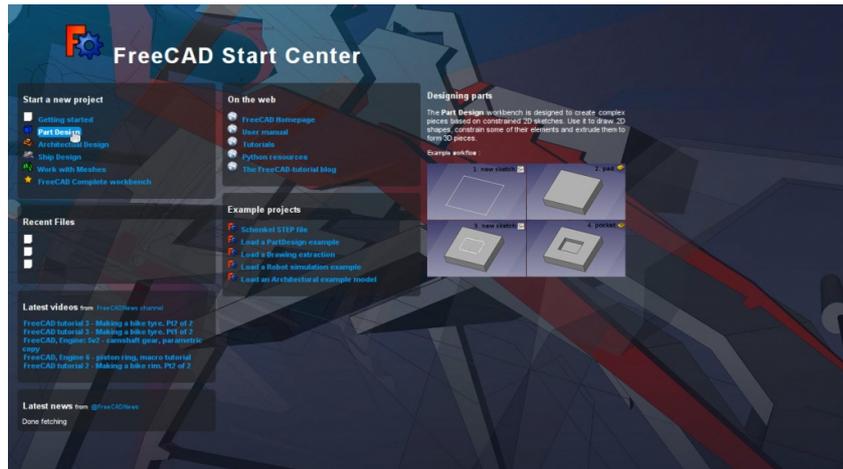
(</wiki/index.php?title=File:Freecad-interface.jpg>)

1. The 3D view, showing the contents of your document
2. The tree view, which shows the hierarchy and construction history of all the objects in your document
3. The properties editor (</wiki/index.php?title=Property>), which allows you to view and modify properties of the selected object(s)
4. The report view (or output window), which is where FreeCAD prints messages, warnings and errors
5. The python console, where all the commands executed by FreeCAD are printed, and where you can enter python code
6. The workbench selector (</wiki/index.php?title=Workbenches>), where you select the active workbench

The main concept behind the FreeCAD interface is that it is separated into workbenches (</wiki/index.php?title=Workbenches>). A workbench is a collection of tools suited for a specific task, such as working with meshes (/wiki/index.php?title=Mesh_Module), or drawing 2D objects (/wiki/index.php?title=Draft_Module), or constrained sketches (/wiki/index.php?title=Sketcher_Module). You can switch the current workbench with the workbench selector (6). You can customize (/wiki/index.php?title=Interface_Customization) the tools included in each workbench, add tools from other workbenches or even self-created tools, that we call macros (</wiki/index.php?title=Macros>). There is also a generic workbench which gathers the most commonly used tools from other workbenches, called the **complete workbench**.

When you start FreeCAD for the first time, you are presented with the start

center:

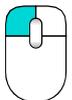
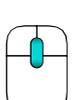
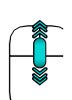


(/wiki/index.php?title=File:Startcenter.jpg)

The Start Center allows you to quickly jump to one of the most common workbenches, open one of the recent files, or see the latest news from the FreeCAD world. You can change the default workbench in the preferences (/wiki/index.php?title=Preferences_Editor).

Navigating in the 3D space

FreeCAD has several different navigation modes (/wiki/index.php?title=Mouse_Model) available, that change the way you use your mouse to interact with the objects in the 3D view and the view itself. One of them is specifically made for touchpads (/wiki/index.php?title=Mouse_Model#Touchpad_Navigation), where the middle mouse button is not used. The following table describes the default mode, called **CAD Navigation** (You can quickly change the current navigation mode by right-clicking on an empty area of the 3D view):

Select	Pan	Zoom
 (/wiki/index.php?title=File:Hand_cursor.png)	 (/wiki/index.php?title=File:Pan_cursor.png)	 (/wiki/index.php?title=File:Zoom_cursor.png)
 (/wiki/index.php?title=File:Select-mouse.svg)	 (/wiki/index.php?title=File:Pan-mouse.svg)	 (/wiki/index.php?title=File:Zoom-mouse.svg)

are working with ship design, there is a special Ship Workbench (/wiki/index.php?title=Ship_Workbench) for you. And if you come from the OpenSCAD world, try the OpenSCAD Workbench (/wiki/index.php?title=OpenSCAD_Workbench).

You can switch workbenches at any time, and also customize (/wiki/index.php?title=Interface_Customization) your favorite workbench to add tools from other workbenches.

Working with the PartDesign and Sketcher workbenches

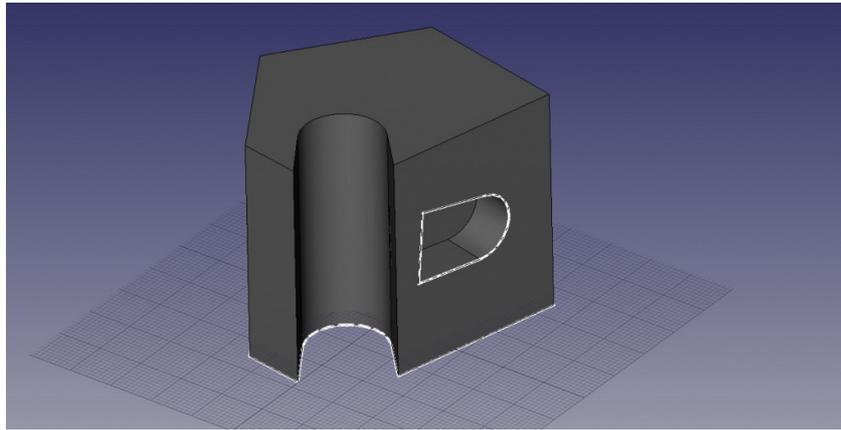
The PartDesign Workbench (/wiki/index.php?title=PartDesign_Workbench) is specially made to build complex objects, starting from simple shapes, and adding or removing pieces (that we call "features"), until you get to your final object. All the features you applied during the modelling process are stored in a separate view called the tree view (/wiki/index.php?title=Document_structure), which also contains the other objects in your document. You can think of a PartDesign object as a succession of operations, each one applied to the result of the preceding one, forming one big chain. In the tree view, you see your final object, but you can expand it and retrieve all preceding states, and change any of their parameter, which automatically updates the final object.

The PartDesign workbench makes heavy use of another workbench, the Sketcher Workbench (/wiki/index.php?title=Sketcher_Workbench). The sketcher allows you to draw 2D shapes, which are defined by applying Constraints to the 2D shape. For example, you might draw a rectangle and set the size of a side by applying a length constraint to one of the sides. That side then cannot be resized anymore (unless the constraint is changed).

Those 2D shapes made with the sketcher are used a lot in the PartDesign workbench, for example to create 3D volumes, or to draw areas on the faces of your object that will then be hollowed from its main volume. This is a typical PartDesign workflow:

1. Create a new sketch
2. Draw a closed shape (make sure all points are joined)
3. Close the sketch
4. Expand the sketch into a 3D solid by using the pad tool
5. Select one face of the solid
6. Create a second sketch (this time it will be drawn on the selected face)
7. Draw a closed shape
8. Close the sketch
9. Create a pocket from the second sketch, on the first object

Which gives you an object like this:



(/wiki/index.php?title=File:Partdesign_example.jpg)

At any moment, you can select the original sketches and modify them, or change the extrusion parameters of the pad or pocket operations, which will update the final object.

Working with the Draft and Arch workbenches

The Draft Workbench (/wiki/index.php?title=Draft_Workbench) and Arch Workbench (/wiki/index.php?title=Arch_Module) behave a bit differently than the other workbenches above, although they follow the same rules, which are common to all of FreeCAD. In short, while the Sketcher and PartDesign are made primarily to design single pieces, Draft and Arch are made to ease your work when working with several, simpler objects.

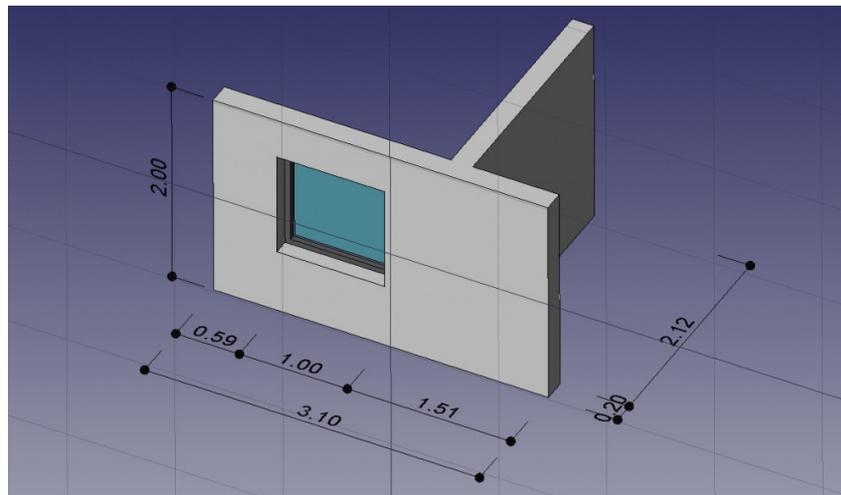
The Draft Workbench (/wiki/index.php?title=Draft_Workbench) offers you 2D tools a bit similar to what you can find in traditional 2D CAD applications such as AutoCAD (<https://en.wikipedia.org/wiki/AutoCAD>). However, 2D drafting being far away from the scope of FreeCAD, don't expect to find there the full array of tools that these dedicated applications offer. Most of the Draft tools work not only in a 2D plane but also in the full 3D space, and benefit from special helper systems such as Work planes (/wiki/index.php?title=Draft_SelectPlane) and object snapping (/wiki/index.php?title=Draft_Snap).

The Arch Workbench (/wiki/index.php?title=Arch_Module) adds BIM (http://en.wikipedia.org/wiki/Building_Information_Modeling) tools to FreeCAD, allowing you to build architectural models with parametric objects. The Arch workbench relies much on other modules such as Draft and Sketcher. All the Draft tools are also present in the Arch workbench, and most Arch tools make use of the Draft helper systems.

A typical workflow with Arch and Draft workbenches might be:

1. Draw a couple of lines with the Draft Line tool
2. Select each line and press the Wall tool to build a wall on each of them
3. Join the walls by selecting them and pressing the Arch Add tool
4. Create a floor object, and move your walls in it from the Tree view
5. Create a building object, and move your floor in it from the Tree view
6. Create a window by clicking the Window tool, select a preset in its panel, then click on a face of a wall
7. Add dimensions by first setting the working plane if necessary, then using the Draft Dimension tool

Which will give you this:



(/wiki/index.php?title=File:Arch_workflow_example.jpg)

More on the Tutorials (/wiki/index.php?title=Tutorials) page.

Scripting

And finally, one of the most powerful features of FreeCAD is the scripting (/wiki/index.php?title=Scripting) environment. From the integrated python console (or from any other external Python script), you can gain access to almost any part of FreeCAD, create or modify geometry, modify the representation of those objects in the 3D scene or access and modify the FreeCAD interface. Python scripting can also be used in macros (/wiki/index.php?title=Macros), which provide an easy method to create custom commands.

< previous: Install on Mac (/wiki/index.php?title=Install_on_Mac)

Index next: Mouse Model > (/wiki/index.php?title=Mouse_Model)

(/wiki/index.php?title=Online_Help_Toc)

Working with FreeCAD

3D navigation

The FreeCAD **mouse model** consists of the commands used to visually navigate the 3D space and interact with the objects displayed. FreeCAD supports multiple mouse model navigation styles. The default navigation style is referred to as "CAD Navigation," and is very simple and practical, but FreeCAD also provides alternative navigation styles, that you can choose according to your preferences.

Navigation

The object handling is common to all workbenches. The following mouse gestures can be used to control the object position and view according to which Navigation style is selected.

There are two ways to change the navigation style:

- In the Preferences Editor (/wiki/index.php?title=Preferences_Editor), Display section, *3D View* tab;
- By right-clicking in empty space in the 3D view area, then selecting *Navigation style* in the contextual menu.

CAD Navigation (default)

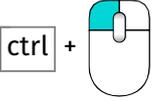
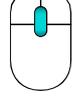
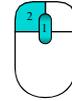
This is the default navigation style and allows the user a simple control of the view, and does not require the use of keyboard keys except to make multi-selections.

Select	Pan	Zoom
 (/wiki/index.php?title=File:Hand_cursor.png)	 (/wiki/index.php?title=File:Pan_cursor.png)	 (/wiki/index.php?title=File:Zoom_cursor.png)
 (/wiki/index.php?title=File:Select-mouse.svg)	 (/wiki/index.php?title=File:Pan-mouse.svg)	 (/wiki/index.php?title=File:Zoom-mouse.svg)
Press the left mouse button over an object you want to select. Holding down ctrl allows the selection of multiple objects.	Click the middle mouse button and move the object around to pan	Use the middle mouse button to zoom in and out. Use the scroll wheel to the local zoom.
	 (/wiki/index.php?title=File:Mouse_2_button_right.svg)	 (/wiki/index.php?title=File:Zoom-mouse.svg)
	Press and hold Ctrl key and click and release right mouse button to pan (rev 0.14)	Once in Zoom, to release local pan, to press and button (re

Inventor Navigation

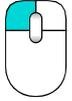
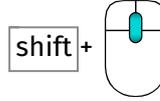
In Inventor Navigation, modeled after Open Inventor (http://en.wikipedia.org/wiki/Open_Inventor) (not to be confused with Autodesk Inventor), there is no mouse-only selection. In order to select objects, you must hold down the **CTRL** key.

Select	Pan	Zoom
 (/wiki/index.php?title=File:Hand_cursor.png)	 (/wiki/index.php?title=File:Pan_cursor.png)	 (/wiki/index.php?title=File:Zoom_cursor.png)

 <p>(/wiki/index.php?title=File:Select-mouse.svg)</p>	 <p>(/wiki/index.php?title=File:Pan-mouse.svg)</p>	 <p>(/wiki/index.php?title=File:Zoom-mouse.svg)</p> <p>or</p>  <p>(/wiki/index.php?title=File:Rotate-mouse.svg)</p>
<p>Hold ctrl and press the left mouse button over an object you want to select.</p>	<p>Click the left mouse button and move the object around.</p>	<p>Use the mouse wheel to zoom in and out, or scroll the mouse wheel and hold the right mouse button and the left mouse button to rotate.</p>

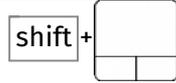
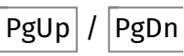
Blender Navigation

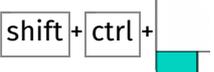
In Blender Navigation, modeled after Blender (<http://www.blender.org>), there is no mouse-only panning. In order to pan the view, you must hold down the **SHIFT** key.

Select	Pan	Zoom
 <p>(/wiki/index.php?title=File:Hand_cursor.png)</p>	 <p>(/wiki/index.php?title=File:Pan_cursor.png)</p>	 <p>(/wiki/index.php?title=File:Zoom_cursor.png)</p>
 <p>(/wiki/index.php?title=File:Select-mouse.svg)</p>	 <p>(/wiki/index.php?title=File:Pan-mouse.svg)</p>	 <p>(/wiki/index.php?title=File:Zoom-mouse.svg)</p>
<p>Press the left mouse button over an object you want to select.</p>	<p>Hold shift and click the middle mouse button and move the object around.</p>	<p>Use the mouse wheel to zoom in and out.</p>

Touchpad Navigation

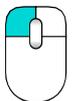
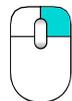
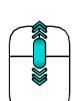
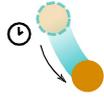
In Touchpad Navigation, neither panning, nor zooming, nor rotating the view, are mouse-only (or touchpad-only) operations.

Select	Pan	Zoom
 <p>(/wiki/index.php?title=File:Hand_cursor.png)</p>	 <p>(/wiki/index.php?title=File:Pan_cursor.png)</p>	 <p>(/wiki/index.php?title=File:Zoom_cursor.png)</p>
 <p>(/wiki/index.php?title=File:Select-touchpad.png)</p>	 <p>(/wiki/index.php?title=File:Touchpad.png)</p>	 <p>PgUp / PgDn</p>
<p>Press the left mouse button over an object you want to select.</p>	<p>Hold shift and move the object around.</p>	<p>Use PgUp and PgDn to zoom in and out.</p>
		<i>or</i>

		 (/wiki/index.php?title=File:Select_touchpad.png)
		Hold down both the shift and the ctrl keys, press the left mouse button and move the pointer.

Gesture Navigation (v0.16)

This navigation style was tailored for usability with touchscreen and pen, but is very usable with mouse too.

Select	Pan	Zoom
 (/wiki/index.php?title=File:Hand_cursor.png)	 (/wiki/index.php?title=File:Pan_cursor.png)	 (/wiki/index.php?title=File:Zoom_cursor.png)
 (/wiki/index.php?title=File:Select-mouse.svg)	 (/wiki/index.php?title=File:Pan-mouse-Ctrl.svg)	 (/wiki/index.php?title=File:Zoom-mouse.svg)
Press the left mouse button over an object you want to select. Holding down Ctrl allows the selection of multiple objects.	Hold right mouse button and drag to pan the view.	Use the mouse wheel to zoom in and out. The zoom is centered at the location.
 (/wiki/index.php?title=File:Touch_Tap.svg)	 (/wiki/index.php?title=File:Touch_Tap-Hold-Drag.svg) or  (/wiki/index.php?title=File:Touch_Two-Finger-Drag.svg)	 (/wiki/index.php?title=File:Touch_Pinch.svg)

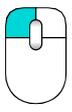
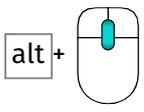
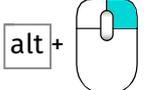
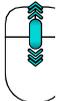
Tap to select.	Drag with two fingers to pan the view. Alternatively, tap and hold, then drag (simulates pan with right mouse button).	Pinch to zoom (i.e. two fingers to other/apart).
----------------	--	--

Notes on Gesture Navigation style:

- on Windows, the actions of two-finger gestures are separated. The action depends on how one starts the gesture. For example, if one starts two-finger pan, the gesture will only pan. Changing the distance between fingers afterwards will not affect the scaling.

Maya-Gesture Navigation

In Maya-Gesture Navigation, all view movements are archived pressing **ALT** and a mouse button, so that it will be needed to have a 3 button mouse in order to correctly use this navigation mode. Alternately it's possible to use gestures as this mode was been developed over the normal Gesture Navigation mode.

Select	Pan	Zoom
 (/wiki/index.php?title=File:Hand_cursor.png)	 (/wiki/index.php?title=File:Pan_cursor.png)	 (/wiki/index.php?title=File:Zoom_cursc
 (/wiki/index.php?title=File:Select-mouse.svg)	 (/wiki/index.php?title=File:Pan-mouse.svg)	 (/wiki/index.php?title=File:Pan-mou Ctrl.svg) or  (/wiki/index.php?title=File:Zoom-mous
Press the left mouse button over an object you want to select.	Hold alt , hold the middle mouse button and drag to pan the view.	Hold alt , hold the mouse button and d zoom in and out or u mouse wheel to g same effect.

Selecting objects

Simple selection

Objects can be selected by a click with the left mouse button either by clicking on the object in the 3D-view or by selecting it in the tree view.

Preselection

There is also a *Preselection* mechanism that highlights objects and displays information before selection by just hovering the mouse over the objects. If you don't like this behaviour or you have a slow machine, you can switch preselection off in the preferences.

Manipulating Objects

FreeCAD offers *manipulators* (</wiki/index.php?title=Manipulator>) that are handles that can be used to modify an object's appearance, shape, or other parameters.

.

Obsolete

The clipping plane (/wiki/index.php?title=Std_ClippingPlane) is a good example of an object with manipulators. A clipping plane (/wiki/index.php?title=Std_ClippingPlane) can be activated with the *View→Clipping Plane* menu. After activation the clipping plane object appears and shows seven obvious manipulators as little boxes: One on each end of its three coordinate axes and one on the center of the plane normal axis. There are four more that are not as obvious: The plane itself and the thin part of the three axis objects.

Scaling

To scale the object click with the left mouse button on the box manipulators at the end of the axes and pull them back and forth. Depending on the object the manipulators work independently or synchronously.

Out of plane shifting

To shift the object along its normal vector, pull the long box on the center of an axis with the left mouse button. For the clipping plane there is only one manipulator along the normal vector.

In plane shifting

To move the center of the clipping plane, click on the plane object and pull it to the desired location.

Rotation

Clicking on the thin part of the axes puts the manipulator in rotation mode.

Hardware support

FreeCAD also supports some 3D input devices (/wiki/index.php?title=3D_input_devices).

Mac OS X Issues

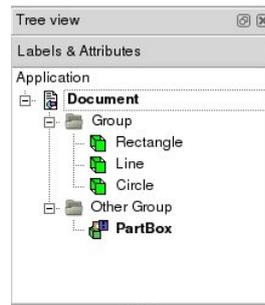
Recently we got reports on the forum (<http://forum.freecadweb.org/viewtopic.php?f=3&t=3592&start=0>) from Mac users that those mouse button and key combination do not work as expected. Unfortunately, none of the developers owns a Mac, neither do the other regular contributors. We need your help to determine which mouse buttons and key combination work so we can update this wiki.

< previous: Getting started (/wiki/index.php?title=Getting_started)

next: Document structure > (/wiki/index.php?title=Document_structure)

Index (/wiki/index.php?title=Online_Help_Toc)

The FreeCAD Document



(</wiki/index.php?>

title=File:Screenshot_treeview.jpg)

A FreeCAD document contains all the objects of your scene. It can contain groups, and objects made with any workbench. You can therefore switch between workbenches, and still work on the same document. The document is what gets saved to disk when you save your work. You can also open several documents at the same time in FreeCAD, and open several views of the same document.

Inside the document, the objects can be moved into groups, and have a unique name. Managing groups, objects and object names is done mainly from the Tree view. It can also be done, of course, like everything in FreeCAD, from the python interpreter. In the Tree view, you can create groups, move objects to groups, delete objects or groups, by right-clicking in the tree view or on an object, rename objects by double-clicking on their names, or possibly other operations, depending on the current workbench.

The objects inside a FreeCAD document can be of different types. Each workbench can create its own types of objects, for example the Mesh Workbench (/wiki/index.php?title=Mesh_Workbench) creates mesh objects, the Part Workbench (/wiki/index.php?title=Part_Workbench) create Part objects, the Draft Workbench (/wiki/index.php?title=Draft_Workbench) also creates Part objects, etc.

If there is at least one document open in FreeCAD, there is always one and only one active document. That's the document that appears in the current 3D view, the document you are currently working on.

Application and User Interface

Like almost everything else in FreeCAD, the user interface part (Gui) is separated from the base application part (App). This is also valid for documents. The documents are also made of two parts: the Application document, which contains our objects, and the View document, which contains the representation on screen of our objects.

Think of it as two spaces, where the objects are defined. Their constructive parameters (is it a cube? a cone? which size?) are stored in the Application document, while their graphical representation (is it drawn with black lines? with blue faces?) are stored in the View document. Why is that? Because FreeCAD can also be used WITHOUT graphical interface, for example inside other programs, and we must still be able to manipulate our objects, even if nothing is drawn on the screen.

Another thing that is contained inside the View document are 3D views. One document can have several views opened, so you can inspect your document from several points of view at the same time. Maybe you would want to see a top view and a front view of your work at the same time? Then, you will have two views of the same document, both stored in the View document. Creating new views or closing views can be done from the View menu or by right-clicking on a view tab.

Scripting

Documents can be easily created, accessed and modified from the python interpreter. For example:

```
FreeCAD.ActiveDocument
```

Will return the current (active) document

```
FreeCAD.ActiveDocument.Blob
```

Would access an object called "Blob" inside your document

```
FreeCADGui.ActiveDocument
```

Will return the view document associated to the current document

```
FreeCADGui.ActiveDocument.Blob
```

Would access the graphical representation (view) part of our Blob object

```
FreeCADGui.ActiveDocument.ActiveView
```

Will return the current view

< previous: Mouse Model (/wiki/index.php?title=Mouse_Model)

next: Preferences Editor > (/wiki/index.php?title=Preferences_Editor)

[Index \(/wiki/index.php?title=Online_Help_Toc\)](/wiki/index.php?title=Online_Help_Toc)

Setting User Preferences

The preferences system of FreeCAD is located in the Edit menu -> Preferences.

FreeCAD functionality is divided into different modules, each module being responsible for the working of a specific workbench (</wiki/index.php?title=Workbenches>). FreeCAD also uses a concept called late loading, which means that components are loaded only when they are needed. You may have noticed that when you select a workbench on the FreeCAD toolbar, that workbench and all its components get loaded at that moment. This includes its preferences settings.



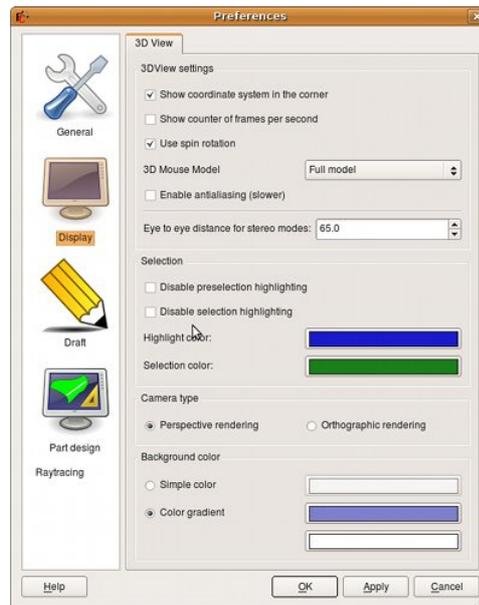
title=File:Screenshot_preferences01.jpg)

The general preferences settings

When you start FreeCAD with no workbench loaded, you will then have a minimal preferences window. As you load additional modules, new sections will appear in the preferences window, allowing you to configure the details of each workbench.

(</wiki/index.php?>

Without any module loaded, you will have access to two configuration sections, responsible for the general application settings and for the display settings.



(/wiki/index.php?

title=File:Screenshot_preferences02.jpg)

The display settings

FreeCAD is always in constant evolution, so the contents of those screens might differ from the above screenshots. The settings are usually self-explanatory, so you shouldn't have any difficulty configuring FreeCAD to your needs.

The Draft module has its preferences (/[wiki/index.php?title=Draft_Preferences](#)) screen

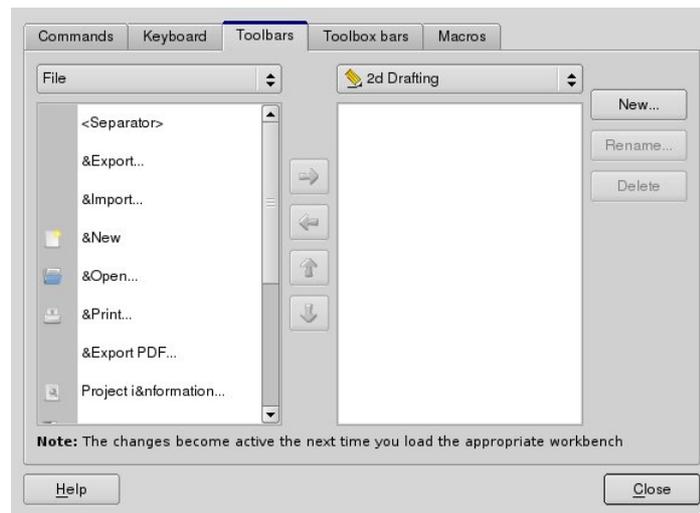
< [previous: Document structure](#) (/[wiki/index.php?title=Document_structure](#))

[next: Interface Customization](#) > (/[wiki/index.php?title=Interface_Customization](#))

[Index](#) (/[wiki/index.php?title=Online_Help_Toc](#))

Customizing the Interface

Since FreeCAD interface is based on the modern Qt ([http://en.wikipedia.org/wiki/Qt_\(toolkit\)](http://en.wikipedia.org/wiki/Qt_(toolkit))) toolkit, it has a state-of-the-art organization. Widgets, menus, toolbars and other tools can be modified, moved, shared between workbenches, keyboard shortcuts can be set, modified, and macros can be recorded and played. The customization window is accessed from the **Tools -> Customize** menu:



(/wiki/index.php?title=File:Screenshot-customize.jpg)

The **Commands** tab lets you browse all available FreeCAD commands, organized by their category.

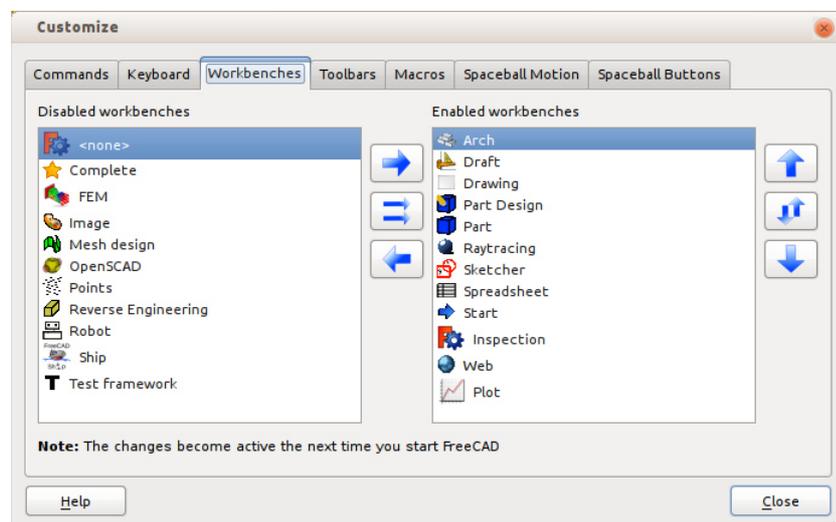
In **Keyboard**, you can see the keyboard shortcuts associated with every FreeCAD command, and if you want, modify or assign new shortcut to any command. This is where to come if you use a particular workbench often, and would like to speed up its use by using the keyboard.

The **Toolbars** and **Toolbox bars** tabs let you modify existing toolbars, or create your own custom toolbars.

The **Macros** tab lets you manage your saved Macros (/wiki/index.php?title=Macros).

Create your ToolBars for your macro Customize ToolsBar (/wiki/index.php?title=Customize_ToolsBar)

In 0.16 version is available a new tool that lets you manage your workbenches



(/wiki/index.php?title=File:CustomizeWorkbenches.png)

< previous: Preferences Editor (/wiki/index.php?title=Preferences_Editor)

next: Property editor > (/wiki/index.php?title=Property_editor)

Index (/wiki/index.php?title=Online_Help_Toc)

Object properties

A **property** is a piece of information like a number or a text string that is attached to a FreeCAD document or an object in a document. Properties can be viewed and - if allowed - modified with the Property editor (/wiki/index.php?title=Property_editor).

Properties play a very important part in FreeCAD, since it is from the beginning made to work with parametric objects, which are objects defined only by their properties.

Custom scripted objects (/wiki/index.php?title=Scripted_objects) in FreeCAD can have properties of the following types:

```

Boolean
Float
FloatList
FloatConstraint
Angle
Distance
Integer
IntegerConstraint
Percent
Enumeration
IntegerList
String
StringList
Link
LinkList
Matrix
Vector
VectorList
Placement
PlacementLink
Color
ColorList
Material
Path
File
FileIncluded
PartShape
FilletContour
Circle

```

< [previous: Interface Customization \(/wiki/index.php?title=Interface_Customization\)](/wiki/index.php?title=Interface_Customization)

[Index next: Workbenches > \(/wiki/index.php?title=Workbenches\)](/wiki/index.php?title=Workbenches)
[\(/wiki/index.php?title=Online_Help_Toc\)](/wiki/index.php?title=Online_Help_Toc)

Working with workbenches

FreeCAD, like many modern design applications such as Revit (<http://en.wikipedia.org/wiki/Revit>) or CATIA (<http://en.wikipedia.org/wiki/CATIA>), is based on the concept of Workbench (<http://en.wikipedia.org/wiki/Workbench>). A workbench can be considered as a set of tools specially grouped for a certain task. In a traditional furniture workshop, you would have a work table for the person who works with wood, another one for the one who works with metal pieces, and maybe a third one for the guy who mounts all the pieces together.

In FreeCAD, the same concept applies. Tools are grouped into workbenches according to the tasks they are related to.

The following workbenches are available:

-  (/wiki/index.php?title=File:Workbench_Arch.png) The Arch Module (/wiki/index.php?title=Arch_Module) for working with architectural elements.
-  (/wiki/index.php?title=File:Workbench_Assembly.png) The Assembly Module (/wiki/index.php?title=Assembly_project) for working with

multiple shapes, multiple documents, multiple files, multiple relationships...

-  (/wiki/index.php?title=File:Workbench_Complete.png) The Complete Workbench (/wiki/index.php?title=Complete_Workbench) hold all commands and features from all the modules and workbenches which met certain quality criteria.
-  (/wiki/index.php?title=File:Workbench_Draft.png) The Draft Workbench (/wiki/index.php?title=Draft_Module) contains 2D tools and basic 2D and 3D CAD operations.
-  (/wiki/index.php?title=File:Workbench_Drawing.png) The Drawing workbench (/wiki/index.php?title=Drawing_Module) for displaying your 3D work on a 2D sheet.
-  (/wiki/index.php?title=File:Workbench_FEM.png) The FEM Module (/wiki/index.php?title=FEM_Module) provides Finite Element Analysis (FEA) workflow.
-  (/wiki/index.php?title=File:Workbench_Image.png) The Image Module (/wiki/index.php?title=Image_Module) for working with bitmap images.
-  (/wiki/index.php?title=File:Workbench_Inspection.png) The Inspection Module (/wiki/index.php?title=Inspection_Workbench) is made to give you specific tools for examination of shapes. It is still in development.
-  (/wiki/index.php?title=File:Workbench_Mesh.png) The Mesh Module (/wiki/index.php?title=Mesh_Module) for working with triangulated meshes.
-  (/wiki/index.php?title=File:Workbench_OpenSCAD.png) The OpenSCAD Module (/wiki/index.php?title=OpenSCAD_Module) for interoperability with OpenSCAD and repairing CSG model history.
-  (/wiki/index.php?title=File:Workbench_Part.png) The Part Module (/wiki/index.php?title=Part_Module) for working with CAD parts.
-  (/wiki/index.php?title=File:Workbench_PartDesign.png) The Part Design Workbench (/wiki/index.php?title=PartDesign_Workbench) for building Part shapes from sketches.
-  (/wiki/index.php?title=File:Workbench_Path.png) The Path Workbench (/wiki/index.php?title=Path_Workbench) is used to produce G-Code instructions. It is still in early stages of development. Only v 0.16
-  (/wiki/index.php?title=File:Workbench_Plot.png) The Plot Workbench (/wiki/index.php?title=Plot_Module) The Plot module allows to edit and save output plots created from other modules and tools.
-  (/wiki/index.php?title=File:Workbench_Points.png) The Points Module (/wiki/index.php?title=Points_Module) for working with point clouds.
-  (/wiki/index.php?title=File:Workbench_Raytracing.png) The Raytracing Module (/wiki/index.php?title=Raytracing_Module) for working with ray-tracing (rendering)
-  (/wiki/index.php?title=File:Workbench_Reverse_Engineering.png) The Reverse Engineering Module (/wiki/index.php?title=Reverse_Engineering_Workbench) is intended to give you specific tools to convert shapes/solids/meshes into parametric FreeCAD-compatible features. It is still in development.
-  (/wiki/index.php?title=File:Workbench_Robot.png) The Robot Module (/wiki/index.php?title=Robot_Module) for studying robot movements.

-  (/wiki/index.php?title=File:Workbench_Ship.png) The Ship Workbench (/wiki/index.php?title=Ship_Workbench) FreeCAD-Ship works over Ship entities, that must be created on top of provided geometry.
-  (/wiki/index.php?title=File:Workbench_Sketcher.png) The Sketcher Module (/wiki/index.php?title=Sketcher_Module) for working with geometry-constrained sketches.
-  (/wiki/index.php?title=File:Workbench_Spreadsheet.png) The Spreadsheet Workbench (/wiki/index.php?title=Spreadsheet_Module) for creating and manipulating spreadsheet data.
-  (/wiki/index.php?title=File:Workbench_Start.png) The Start Center (/wiki/index.php?title=Start_Workbench) allows you to quickly jump to one of the most common workbenches.
-  (/wiki/index.php?title=File:Workbench_Test.png) The Test framework (</wiki/index.php?title=Debugging>) is for debugging FreeCAD.
-  (/wiki/index.php?title=File:Workbench_Web.png) The Web Module (/wiki/index.php?title=Web_Workbench) provides you with a browser window instead of the 3D-View within FreeCAD.

New workbenches are in development, stay tuned!

When you switch from one workbench to another, the tools available on the interface change. Toolbars, command bars and possibly other parts of the interface switch to the new workbench, but the contents of your scene doesn't change. You could, for example, start drawing 2D shapes with the Draft Workbench, then work further on them with the Part Workbench.

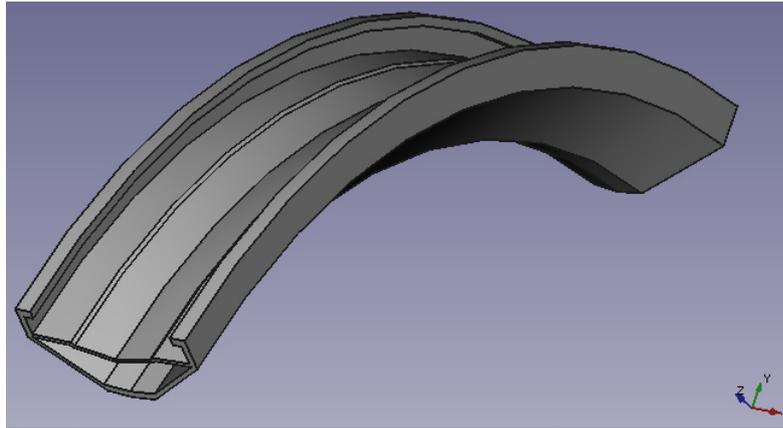
Note that sometimes a Workbench is referred to as a *Module*. However, Workbenches and Modules are different entities. A Module is any extension of FreeCAD, while a Workbench is a special GUI configuration that groups some toolbars and menus. Usually every Module contains its own Workbench, hence the cross-use of the name.

< [previous: Property editor](/wiki/index.php?title=Property_editor) (/wiki/index.php?title=Property_editor)
 next: [PartDesign Workbench](/wiki/index.php?title=PartDesign_Workbench) > (/wiki/index.php?title=PartDesign_Workbench)
[Index](/wiki/index.php?title=Online_Help_Toc) (/wiki/index.php?title=Online_Help_Toc)

The PartDesign workbench

The **Part Design Workbench** provides tools for modelling complex solid parts and is based on a **Feature editing methodology** to produce a single contiguous solid. It is intricately linked with the Sketcher Workbench (/wiki/index.php?title=Sketcher_Workbench).

What is a **single contiguous solid**? This is an item like a casting or something machined from a single block of metal. If the item involves nails, screws, glue or welding, it is not a **single contiguous solid**. As a practical example, PartDesign would not be used to model a wooden chair, but would be used to model the subcomponents (legs, slats, seat, etc). The subcomponents are combined using the Assembly (/wiki/index.php?title=Assembly_Workbench), Part (/wiki/index.php?title=Part_Workbench) or Draft (/wiki/index.php?title=Draft_Workbench) workbench.



(/wiki/index.php?title=File:Revolve3_cropped.png)

Basic Workflow

The sketch is the building block for creating and editing solid parts. The workflow can be summarized by this: a sketch containing 2D geometry is created first, then a solid creation tool is used on the sketch. At the moment the available tools are:

-  (/wiki/index.php?title=File:PartDesign_Pad.png) **Pad** which extrudes a sketch
-  (/wiki/index.php?title=File:PartDesign_Pocket.png) **Pocket** which creates a pocket on an existing solid
-  (/wiki/index.php?title=File:PartDesign_Revolution.png) **Revolution** which creates a solid by revolving a sketch along an axis
-  (/wiki/index.php?title=File:PartDesign_Groove.png) **Groove** which creates a groove in an existing solid

More tools are planned in future releases.

A very important concept in the PartDesign Workbench is the **sketch support**. Sketches can be created on standard planes (**XY**, **XZ**, **YZ** and planes parallel to them) or on a planar face of an existing solid. For this last case, the existing solid becomes the **support** of the sketch. Several tools will only work with sketches that have a support, for example, **Pocket** - without a support there would be nothing to remove material from!

After solid geometry has been created it can be modified with chamfers and fillets or transformed, e.g. mirrored or patterned.

The PartDesign Workbench is meant to create a single, connected solid. Multiple solids will be possible with the Assembly workbench (/wiki/index.php?title=Assembly_Workbench).

As we create a model in the Part Design Workbench, each feature takes the shape of the last one and adds or removes something, creating linear dependencies from feature to feature as the model is created. Hence a "Cut" feature is not only the cut hole itself, but the whole part with the cut. As a new feature is added to the model, FreeCAD turns off visibility of the old features. The user usually should only have the newest item (feature) in the model tree visible, because otherwise the other phases of the model overlay each other, and holes are filled in by the earlier model features that didn't yet have those holes.

To toggle visibility of an object on or off, select it in the hierarchy tree and press the Spacebar. Usually everything but the last item in the hierarchy tree should be greyed out and therefore not visible in the 3D view.

The Tools

The Part Design tools are all located in the **Part Design** menu that appears

when you load the Part Design module.

They include the Sketcher Workbench (/wiki/index.php?title=Sketcher_Workbench) tools, since the Part Design module is so dependent on them.

The Sketcher Tools

Sketcher Geometries

These are tools for creating objects.

-  (/wiki/index.php?title=File:Sketcher_CreatePoint.png) Point (/wiki/index.php?title=Sketcher_Point): Draws a point.
-  (/wiki/index.php?title=File:Sketcher_Line.png) Line by 2 point (/wiki/index.php?title=Sketcher_Line): Draws a line segment from 2 points.
-  (/wiki/index.php?title=File:Sketcher_Arc.png) Arc (/wiki/index.php?title=Sketcher_Arc): Draws an arc segment from center, radius, start angle and end angle.
-  32px (/wiki/index.php?title=Special:Upload&wpDestFile=Sketcher_Create3PointArc.png) Arc by 3 Point (/wiki/index.php?title=Sketcher_Arc3Point): Draws an arc segment from two endpoints and another point on the circumference.
-  (/wiki/index.php?title=File:Sketcher_Circle.png) Circle (/wiki/index.php?title=Sketcher_Circle): Draws a circle from center and radius.
-  32px (/wiki/index.php?title=Special:Upload&wpDestFile=Sketcher_Create3PointCircle.png) Circle by 3 Point (/wiki/index.php?title=Sketcher_Circle3Point): Draws a circle from three points on the circumference.
-  (/wiki/index.php?title=File:Sketcher_Conics.png) Conic sections (/wiki/index.php?title=Sketcher_Conic_Sections):
 -  (/wiki/index.php?title=File:Sketcher_CreateEllipse.png) Ellipse by center (/wiki/index.php?title=Sketcher_Ellipse): Draws an ellipse by center point, major radius point and minor radius point. (v0.15)
 -  (/wiki/index.php?title=File:Sketcher_CreateEllipse_3points.png) Ellipse by 3 points (/wiki/index.php?title=Sketcher_Ellipse_by_3_Points): Draws an ellipse by major diameter (2 points) and minor radius point. (v0.15)
 -  (/wiki/index.php?title=File:Sketcher_Elliptical_Arc.png) Arc of ellipse (/wiki/index.php?title=Sketcher_Arc_of_Ellipse): Draws an arc of ellipse by center point, major radius point, starting point and ending point. (v0.15)
-  (/wiki/index.php?title=File:Sketcher_CreatePolyline.png) Polyline (multiple-point line) (/wiki/index.php?title=Sketcher_Polyline): Draws a line made of multiple line segments. Pressing the M key while drawing a Polyline toggles between the different polyline modes.
-  (/wiki/index.php?title=File:Sketcher_CreateRectangle.png) Rectangle (/wiki/index.php?title=Sketcher_Rectangle): Draws a rectangle from 2 opposite points.
-  (/wiki/index.php?title=File:Sketcher_CreateTriangle.png) Triangle (/wiki/index.php?title=Sketcher_Triangle): Draws a regular triangle inscribed in a construction geometry circle. (v0.15)

-  (/wiki/index.php?title=File:Sketcher_CreateSquare.png) Square (/wiki/index.php?title=Sketcher_Square): Draws a regular square inscribed in a construction geometry circle. (v0.15)
-  (/wiki/index.php?title=File:Sketcher_CreatePentagon.png) Pentagon (/wiki/index.php?title=Sketcher_Pentagon): Draws a regular pentagon inscribed in a construction geometry circle. (v0.15)
-  (/wiki/index.php?title=File:Sketcher_CreateHexagon.png) Hexagon (/wiki/index.php?title=Sketcher_Hexagon): Draws a regular hexagon inscribed in a construction geometry circle. (v0.15)
-  (/wiki/index.php?title=File:Sketcher_CreateHeptagon.png) Heptagon (/wiki/index.php?title=Sketcher_Heptagon): Draws a regular heptagon inscribed in a construction geometry circle. (v0.15)
-  (/wiki/index.php?title=File:Sketcher_CreateOctagon.png) Octagon (/wiki/index.php?title=Sketcher_Octagon): Draws a regular octagon inscribed in a construction geometry circle. (v0.15)
-  (/wiki/index.php?title=File:Sketcher_CreateSlot.png) Slot (/wiki/index.php?title=Sketcher_Slot): Draws an oval by selecting the center of one semicircle and an endpoint of the other semicircle.
-  (/wiki/index.php?title=File:Sketcher_CreateFillet.png) Fillet (/wiki/index.php?title=Sketcher_Fillet): Makes a fillet between two lines joined at one point. Select both lines or click on the corner point, then activate the tool.
-  (/wiki/index.php?title=File:Sketcher_Trimming.png) Trimming (/wiki/index.php?title=Sketcher_Trimming): Trims a line, circle or arc with respect to the clicked point.
-  (/wiki/index.php?title=File:Sketcher_External.png) External Geometry (/wiki/index.php?title=Sketcher_External): Creates an edge linked to external geometry.
-  (/wiki/index.php?title=File:Sketcher_AlterConstruction.png) Construction Mode (/wiki/index.php?title=Sketcher_ConstructionMode): Toggles an element to/from construction mode. A construction object will not be used in a 3D geometry operation and is only visible while editing the Sketch that contains it. This is the icon that was used through v0.15. Until FreeCAD v0.16 the user had to first create regular (white) geometry in Sketcher and then use this tool to change it to Construction Geometry (blue).
-  (/wiki/index.php?title=File:Sketcher_ToggleConstruction.png) Construction Mode (/wiki/index.php?title=Sketcher_ToggleConstruction): In FreeCAD v0.16 the ability to create geometry directly in Construction Mode was added, and so the icon was changed to this one. Selecting existing Sketcher geometry and then clicking this tool toggles that geometry between regular and construction mode just as in previous FreeCAD versions. Starting with FreeCAD v0.16, selecting this tool when no Sketcher geometry is selected changes the mode (regular vs. construction) in which future objects will be created.

Sketcher Constraints

Constraints are used to define lengths, set rules between sketch elements, and to lock the sketch along the vertical and horizontal axes. Some constraints require the Helper constraints (/wiki/index.php?title=Sketcher_helper_constraint)

Not associated with numeric data

-  (/wiki/index.php?title=File:Constraint_PointOnPoint.png) Coincident (/wiki/index.php?title=Constraint_PointOnPoint): Affixes a point onto (coincident with) one or more other points.
-  (/wiki/index.php?title=File:Constraint_PointOnObject.png) Point On Object (/wiki/index.php?title=Constraint_PointOnObject): Affixes a point onto another object such as a line, arc, or axis.
-  (/wiki/index.php?title=File:Constraint_Vertical.png) Vertical (/wiki/index.php?title=Constraint_Vertical): Constrains the selected lines or polyline elements to a true vertical orientation. More than one object can be selected before applying this constraint.
-  (/wiki/index.php?title=File:Constraint_Horizontal.png) Horizontal (/wiki/index.php?title=Constraint_Horizontal): Constrains the selected lines or polyline elements to a true horizontal orientation. More than one object can be selected before applying this constraint.
-  (/wiki/index.php?title=File:Constraint_Parallel.png) Parallel (/wiki/index.php?title=Constraint_Parallel): Constrains two or more lines parallel to one another.
-  (/wiki/index.php?title=File:Constraint_Perpendicular.png) Perpendicular (/wiki/index.php?title=Constraint_Perpendicular): Constrains two lines perpendicular to one another, or constrains a line perpendicular to an arc endpoint.
-  (/wiki/index.php?title=File:Constraint_Tangent.png) Tangent (/wiki/index.php?title=Constraint_Tangent): Creates a tangent constraint between two selected entities, or a co-linear constraint between two line segments. A line segment does not have to lie directly on an arc or circle to be constrained tangent to that arc or circle.
-  (/wiki/index.php?title=File:Constraint_EqualLength.png) Equal Length (/wiki/index.php?title=Constraint_EqualLength): Constrains two selected entities equal to one another. If used on circles or arcs their radii will be set equal.
-  (/wiki/index.php?title=File:Constraint_Symmetric.png) Symmetric (/wiki/index.php?title=Constraint_Symmetric): Constrains two points symmetrically about a line, or constrains the first two selected points symmetrically about a third selected point.

Associated with numeric data

For these constraints you can use the expressions (/wiki/index.php?title=Expressions). The data may be taken from a spreadsheet (/wiki/index.php?title=Spreadsheet_Workbench).

-  (/wiki/index.php?title=File:Sketcher_ConstrainLock.png) Lock (/wiki/index.php?title=Constraint_Lock): Constrains the selected item by setting vertical and horizontal distances relative to the origin, thereby locking the location of that item. These constraint distances can be edited later.
-  (/wiki/index.php?title=File:Constraint_HorizontalDistance.png) Horizontal Distance (/wiki/index.php?title=Constraint_HorizontalDistance): Fixes the horizontal distance between two points or line endpoints. If only one item is selected, the distance is set to the origin.
-  (/wiki/index.php?title=File:Constraint_VerticalDistance.png) Vertical Distance (/wiki/index.php?title=Constraint_VerticalDistance): Fixes the vertical distance between 2 points or line endpoints. If only one item is selected, the distance is set to the origin.

-  (/wiki/index.php?title=File:Constraint_Length.png) Length (/wiki/index.php?title=Constraint_Length): Defines the distance of a selected line by constraining its length, or defines the distance between two points by constraining the distance between them.
-  (/wiki/index.php?title=File:Constraint_Radius.png) Radius (/wiki/index.php?title=Constraint_Radius): Defines the radius of a selected arc or circle by constraining the radius.
-  (/wiki/index.php?title=File:Constraint_InternalAngle.png) Internal Angle (/wiki/index.php?title=Constraint_InternalAngle): Defines the internal angle between two selected lines.
-  (/wiki/index.php?title=File:Constraint_SnellsLaw.png) Snell's Law (/wiki/index.php?title=Constraint_SnellsLaw): Constrains two lines to obey a refraction law to simulate the light going through an interface. (v 0.15)
-  (/wiki/index.php?title=File:Constraint_InternalAlignment.png) Internal Alignment (/wiki/index.php?title=Constraint_Internal_Alignment): Aligns selected elements to selected shape (e.g. a line to become major axis of an ellipse).
-  (/wiki/index.php?title=File:Sketcher_ToggleConstraint.png) Toggle Constraint (/wiki/index.php?title=Sketcher_ToggleConstraint): Toggles the toolbar or the selected constraints to/from reference mode. v0.16

Other

-  (/wiki/index.php?title=File:Sketcher_NewSketch.png) New sketch (/wiki/index.php?title=Sketcher_NewSketch): Creates a new sketch on a selected face or plane. If no face is selected while this tool is executed the user is prompted to select a plane from a pop-up window.
-  (/wiki/index.php?title=File:Sketcher_EditSketch.png) Edit sketch (/wiki/index.php?title=Sketcher_EditSketch): Edit the selected Sketch.
-  (/wiki/index.php?title=File:Sketcher_LeaveSketch.png) Leave sketch (/wiki/index.php?title=Sketcher_LeaveSketch): Leave the Sketch editing mode.
-  (/wiki/index.php?title=File:Sketcher_ViewSketch.png) View sketch (/wiki/index.php?title=Sketcher_ViewSketch): Sets the model view perpendicular to the sketch plane.
-  (/wiki/index.php?title=File:Sketcher_MapSketch.png) Map sketch to face (/wiki/index.php?title=Sketcher_MapSketch): Maps a sketch to the previously selected face of a solid.
- Reorient sketch (/wiki/index.php?title=Sketcher_Reorient): Allows you to change the position of a sketch
- Validate sketch (/wiki/index.php?title=Sketcher_Validate): It allows you to check if there are in the tolerance of different points and to match them.
-  (/wiki/index.php?title=File:Sketcher_MergeSketch.png) Merge sketches (/wiki/index.php?title=Sketcher_MergeSketch): Merge two or more sketches. [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_MirrorSketch.png) Mirror sketch (/wiki/index.php?title=Sketcher_MirrorSketch): Mirror a sketch along the x-axis, the y-axis or the origin [v 0.16]

-  (/wiki/index.php?title=File:Sketcher_CloseShape.png) Close Shape (/wiki/index.php?title=Sketcher_CloseShape): Creates a closed shape by applying coincident constraints to endpoints [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_ConnectLines.png) Connect Edges (/wiki/index.php?title=Sketcher_ConnectLines): Connect sketcher elements by applying coincident constraints to endpoints [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_SelectConstraints.png) Select Constraints (/wiki/index.php?title=Sketcher_SelectConstraints): Selects the constraints of a sketcher element [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_SelectOrigin.png) Select Origin (/wiki/index.php?title=Sketcher_SelectOrigin): Selects the origin of a sketch [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_SelectVerticalAxis.png) Select Vertical Axis (/wiki/index.php?title=Sketcher_SelectVerticalAxis): Selects the vertical axis of a sketch [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_SelectHorizontalAxis.png) Select Horizontal Axis (/wiki/index.php?title=Sketcher_SelectHorizontalAxis): Selects the horizontal axis of a sketch [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_SelectRedundantConstraints.png) Select Redundant Constraints (/wiki/index.php?title=Sketcher_SelectRedundantConstraints): Selects redundant constraints of a sketch [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_SelectConflictingConstraints.png) Select Conflicting Constraints (/wiki/index.php?title=Sketcher_SelectConflictingConstraints): Selects conflicting constraints of a sketch [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_SelectElementsAssociatedWithConstraints.png) Select Elements Associated with constraints (/wiki/index.php?title=Sketcher_SelectElementsAssociatedWithConstraints): Select sketcher elements associated with constraints [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_Element_Ellipse_All.png) Show/Hide internal geometry (/wiki/index.php?title=Sketcher_Show_Hide_Internal_Geometry): Recreates missing/deletes unneeded geometry aligned to internal geometry of a selected element (applicable only to ellipse so far). [v 0.15]
-  (/wiki/index.php?title=File:Sketcher_Symmetry.png) Symmetry (/wiki/index.php?title=Sketcher_Symmetry): Copies a sketcher element symmetrical to a chosen line [v 0.16]
-  (/wiki/index.php?title=File:Sketcher_Clone.png) Clone (/wiki/index.php?title=Sketcher_Clone): Clones a sketcher element [v 0.16]
-  (/wiki/index.php?title=File:Sketcher_Copy.png) Copy (/wiki/index.php?title=Sketcher_Copy): Copies a sketcher element [v 0.16]

- 
 (/wiki/index.php?title=File:Sketcher_RectangularArray.png)
 Rectangular Array (/wiki/index.php?title=Sketcher_RectangularArray):
 Creates an array of selected sketcher elements [v 0.16]

The Part Design Tools

Construction tools

These are tools for creating solid objects or removing material from an existing solid object.

- 
 (/wiki/index.php?title=File:PartDesign_Pad.png) Pad
 (/wiki/index.php?title=PartDesign_Pad): Extrudes a solid object from a selected sketch.
- 
 (/wiki/index.php?title=File:PartDesign_Pocket.png) Pocket
 (/wiki/index.php?title=PartDesign_Pocket): Creates a pocket from a selected sketch. The sketch must be mapped to an existing solid object's face.
- 
 (/wiki/index.php?title=File:PartDesign_Revolution.png) Revolution
 (/wiki/index.php?title=PartDesign_Revolution): Creates a solid by revolving a sketch around an axis. The sketch must be a closed profile to get a solid object.
- 
 (/wiki/index.php?title=File:PartDesign_Groove.png) Groove
 (/wiki/index.php?title=PartDesign_Groove): Creates a groove by revolving a sketch around an axis. The sketch must be mapped to an existing solid object's face.

Modification tools

These are tools for modifying existing objects. They will allow you to choose which object to modify.

- 
 (/wiki/index.php?title=File:PartDesign_Fillet.png) Fillet
 (/wiki/index.php?title=PartDesign_Fillet): Fillets (rounds) edges of an object.
- 
 (/wiki/index.php?title=File:PartDesign_Chamfer.png) Chamfer
 (/wiki/index.php?title=PartDesign_Chamfer): Chamfers edges of an object.
- 
 (/wiki/index.php?title=File:PartDesign_Draft.png) Draft
 (/wiki/index.php?title=PartDesign_Draft): Applies angular draft to faces of an object.

Transformation tools

These are tools for transforming existing features. They will allow you to choose which features to transform.

- 
 (/wiki/index.php?title=File:PartDesign_Mirrored.png) Mirrored
 (/wiki/index.php?title=PartDesign_Mirrored): Mirrors features on a plane or face.
- 
 (/wiki/index.php?title=File:PartDesign_LinearPattern.png) Linear Pattern
 (/wiki/index.php?title=PartDesign_LinearPattern): Creates a linear pattern of features.
- 
 (/wiki/index.php?title=File:PartDesign_PolarPattern.png) Polar Pattern
 (/wiki/index.php?title=PartDesign_PolarPattern): Creates a polar pattern of features.
- 
 (/wiki/index.php?title=File:PartDesign_Scaled.png) Scaled
 (/wiki/index.php?title=PartDesign_Scaled): Scales features to a different size.

- 
[\(/wiki/index.php?title=File:PartDesign_MultiTransform.png\)](/wiki/index.php?title=File:PartDesign_MultiTransform.png)
MultiTransform (/wiki/index.php?title=PartDesign_MultiTransform):
 Allows creating a pattern with any combination of the other transformations.

Extras

Some optional functionality that has been created for the PartDesign Workbench:

- 
[\(/wiki/index.php?title=File:PartDesign_WizardShaft.png\)](/wiki/index.php?title=File:PartDesign_WizardShaft.png) **Shaft design wizard** (/wiki/index.php?title=PartDesign_WizardShaft):
 Generates a shaft from a table of values and allows to analyze forces and moments
- 
[\(/wiki/index.php?title=File:PartDesign_InternalExternalGear.svg\)](/wiki/index.php?title=File:PartDesign_InternalExternalGear.svg) **Involute gear** (/wiki/index.php?title=PartDesign_InvoluteGear): allows you to create gear

Feature properties

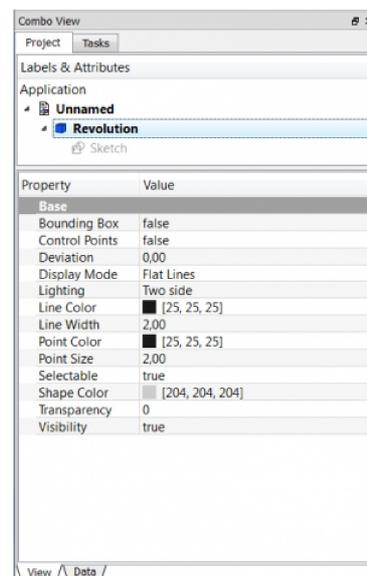
Properties

There are two types of feature properties, accessible through tabs at the bottom of the Property editor:

VIEW View : properties related to the *visual* display of the object.

DATA Data : properties related to the *physical* parameters of an object.

View



(</wiki/index.php?>

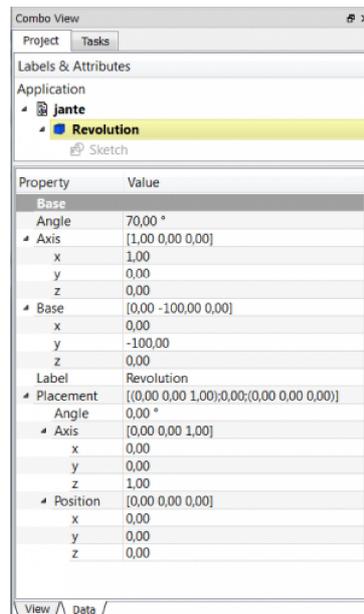
[title=File:PartDesign_Revolution_en_03.png](/wiki/index.php?title=File:PartDesign_Revolution_en_03.png))

Base

- VIEW Bounding Box** : To view the occupation, and, overall, of the object dimensions in space. Value False, or True (Default, False).
- VIEW Control Point** : Value False, or True (Default, False).

- VIEW **Deviation** : Sets the accuracy of the polygonal representation of the model in 3d view (tessellation). Lower values = better quality. The value is in percent of object's size (deviation in mm = $(w+h+d)/3 \times \text{valueInPercent}/100$, where w,h,d are sizes of bounding box).
- VIEW **Display Mode** : Display mode of the form, **Flat lines**, **Shaded**, **Wireframe**, **Points**  (/wiki/index.php?title=File:Vue_DisplayModePartDesign_fr_00.png). (Default, **Flat lines**).
- VIEW **Lighting** : Lighting **One side**, **Two side**  (/wiki/index.php?title=File:Vue_Lighting_fr_00.png). (Default, **Two side**).
- VIEW **Line Color** : Gives the color of the line (edges) (Default, **25, 25, 25**).
- VIEW **Line Width** : Gives the thickness of the line (edges) (Default, **2**).
- VIEW **Point Color** : Gives the color of the points (ends of the form) (Default, **25, 25, 25**).
- VIEW **Point Size** : Gives the size of the points (Default, **2**).
- VIEW **Selectable** : Allows the selection of the form. Value False, ou True (Default, True).
- VIEW **Shape Color** : Give the color shape (default, **204, 204, 204**).
- VIEW **Transparency** : Sets the degree of transparency in the form of **0** to **100** (Default, **0**).
- VIEW **Visibility** : Determines the visibility of the form (like the bar SPACE). Value False, or True (Default, True).

Data



(/wiki/index.php?

title=File:PartDesign_Revolution_en_04.png)

Base DATA **Angle** : The argument **Angle**, indicates the angle that will be used with the option **Axis** (below). Here, an angle is defined. The angle on the axis is set with the option **Axis**.
The object takes the specified angle around the specified axis.

An example, if you create an object with a required revolution should be rotate functionality of a certain amount, in order to enable it to take the same angle that another element existing.

DATA Axis : This option specifies the axis/axes to rotate the created object. The exact value of rotation comes from the angle (see above) option. This option takes three arguments, these arguments, are transmitted in the form of numbers, **x**, **y** or **z**. Adding a value, more of an axis, will the rotation to each specified axis angle.

For example, with a **Angle of 15 °** : specifying, **1.0 for x** and **2.0 for y**, will rotate **15 ° and 30 °** in the y-axis and the x-axis (final position),

DATA Base : This option specifies the offset in either axes **x**, **y**, or **z**, and accept any number as the argument for each field.

DATA Label : The Label is the name given to the operation, this name can be changed at convenience.

DATA Placement : **[(0.00 0.00 1.00);0.00;(0.00 0.00 0.00)]** Summary below data. Every feature has a placement that can be controlled through the Data Properties table. It controls the placement of the part with respect to the coordinate system. NOTE: The placement options do not affect the physical dimensions of the feature, but merely its position in space!

If you select the title **Placement**  (/wiki/index.php?title=File:Tache_Placement_01_fr_00.png), a button with three small points appears, clicking this button , you have access to the options window **Tasks_Placement** (/wiki/index.php?title=Tasks_Placement).

DATA Angle : The Angle argument specifies the angle to be used with the axis option (below). An angle is set here, and the axis that the angle acts upon is set with the axis option. The feature is rotated by the specified angle, about the specified axis. A usage example might be if you created a revolution feature as required, but then needed to rotate the whole feature by some amount, in order to allow it to line-up with another pre-existing feature.

DATA Axis : This option specifies the axis/axes about which the created feature is to be rotated. The exact value of rotation comes from the angle option (above). This option takes three arguments, which are passed as numbers to either the x, y, or z boxes in the tool. Adding a value to more than one of the axes will cause the part to be rotated by the angle in each axis. For example, with an angle of 15° set, specifying a value of 1.0 for x, and 2.0 for y will cause the finished part to be rotated 15° in the x-axis AND 30° in the y-axis.

DATA Position : This option specifies the base point to which all dimensions refer. This option takes three arguments, which are passed as numbers to either the x, y, or z boxes in the tool. Adding a value to more than one of the boxes will cause the part to be translated by the number of units along the corresponding axis.

PS: The displayed properties can vary, depending on the tool used.

Tutorials

Only for a development version of FreeCAD that is not currently available as a binary or installer:

- PartDesign Bearingholder Tutorial I (/wiki/index.php?title=PartDesign_Bearingholder_Tutorial_I)
- PartDesign Bearingholder Tutorial II (/wiki/index.php?title=PartDesign_Bearingholder_Tutorial_II)
- PartDesign tutorial (/wiki/index.php?title=PartDesign_tutorial)
- Basic Part Design Tutorial (/wiki/index.php?title=Basic_Part_Design_Tutorial)

- [Sketcher tutorial \(/wiki/index.php?title=Sketcher_tutorial\)](/wiki/index.php?title=Sketcher_tutorial)

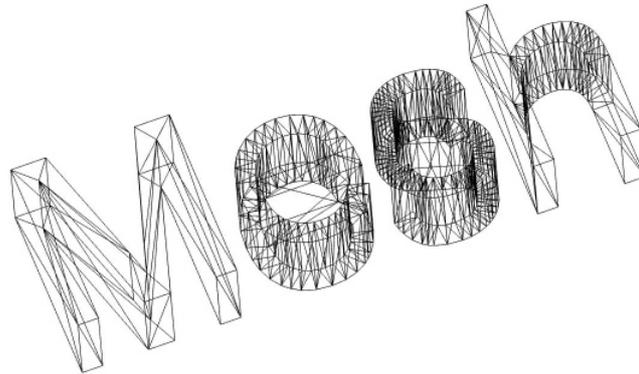
< [previous: Workbenches \(/wiki/index.php?title=Workbenches\)](/wiki/index.php?title=Workbenches)

[next: Mesh Workbench > \(/wiki/index.php?title=Mesh_Workbench\)](/wiki/index.php?title=Mesh_Workbench)

[Index \(/wiki/index.php?title=Online_Help_Toc\)](/wiki/index.php?title=Online_Help_Toc)

The Mesh workbench

The **Mesh Workbench** handles triangle meshes (http://en.wikipedia.org/wiki/Triangle_mesh). Meshes are a special type of 3D object, composed of triangles connected by their edges and their corners (also called vertices).



(/wiki/index.php?title=File:Mesh_example.jpg)

An example of a mesh object

Many 3D applications use meshes as their primary type of 3D object, like sketchup (<http://en.wikipedia.org/wiki/Sketchup>), blender ([http://en.wikipedia.org/wiki/Blender_\(software\)](http://en.wikipedia.org/wiki/Blender_(software))), maya ([http://en.wikipedia.org/wiki/Maya_\(software\)](http://en.wikipedia.org/wiki/Maya_(software))) or 3d studio max (http://en.wikipedia.org/wiki/3d_max). Since meshes are very simple objects, containing only vertices (points), edges and (triangular) faces, they are very easy to create, modify, subdivide, stretch, and can easily be passed from one application to another without any loss. Besides, since they contain very simple data, 3D applications can usually manage very large quantities of them without any problem. For those reasons, meshes are often the 3D object type of choice for applications dealing with movies, animation, and image creation.

In the field of engineering, however, meshes present one big limitation: They are very dumb objects, only composed of points, lines and faces. They are only made of surfaces, and have no mass information, so they don't behave as solids. In a mesh there is no automatic way to know if a point is inside or outside the object. This means that all solid-based operations, such as addition or subtraction, are always a bit difficult to perform on meshes, and return errors often.

In FreeCAD, since it is an engineering application, we would obviously prefer to work with more intelligent types of 3D objects, that can carry more informations, such as mass, solid behaviour, or even custom parameters. The mesh module was first created to serve as a testbed, but to be able to read, manipulate and convert meshes is also highly important for FreeCAD. Very often, in your workflow, you will receive 3D data in mesh format. You will need to handle that data, analyse it to detect errors or other problems that prevent converting them to more intelligent objects, and finally, convert them to more intelligent objects, handled by the Part Module (/wiki/index.php?title=Part_Module).

Using the mesh module

The mesh module has currently a very simple interface, all its functions are grouped in the **Mesh** menu entry. The most important operations you can currently do with meshes are:

-  (/wiki/index.php?title=File:Mesh_ImportMesh.png) Import Mesh (/wiki/index.php?title=Mesh_Import): Import meshes in several file formats
-  (/wiki/index.php?title=File:Mesh_ExportMesh.png) Export Mesh (/wiki/index.php?title=Mesh_Export): Export meshes in several file formats
-  (/wiki/index.php?title=File:Mesh_MeshFromShape.png) Create Mesh from shape (/wiki/index.php?title=Mesh_MeshFromShape): Convert Part (/wiki/index.php?title=Part_Module) objects into meshes
-  (/wiki/index.php?title=File:Mesh_HarmonizeNormals.png) Harmonize Normals (/wiki/index.php?title=Mesh_HarmonizeNormals): Harmonize normals
-  (/wiki/index.php?title=File:Mesh_FlipNormals.png) Flip Normals (/wiki/index.php?title=Mesh_FlipNormals): Flip normals (http://en.wikipedia.org/wiki/Surface_normal)
- Fill Holes... (/wiki/index.php?title=Mesh_FillHoles): Fill up holes
-  (/wiki/index.php?title=File:Mesh_FillInteractiveHole.png) Close hole (/wiki/index.php?title=Mesh_FillInteractiveHole): Close holes in meshes
-  (/wiki/index.php?title=File:Mesh_RemoveComponents.png) Remove components... (/wiki/index.php?title=Mesh_RemoveComponents): Remove components of meshes
- Remove components by hand... (/wiki/index.php?title=Mesh_RemoveCompByHand): Remove components of meshes by hand
- Add triangle (/wiki/index.php?title=Mesh_AddTriangle): Add triangle
- Smooth... (/wiki/index.php?title=Mesh_Smooth): Smooth mesh
- **Analyze** curvature, faces, and check if a mesh can be safely converted into a solid
 - Evaluate & Repair mesh... (/wiki/index.php?title=Mesh_EvaluateRepair): Evaluates and repairs meshes
 -  (/wiki/index.php?title=File:Mesh_EvaluateFacet.png) Face Info (/wiki/index.php?title=Mesh_EvaluateFacet): Gives info on faces
 - Curvature Info (/wiki/index.php?title=Mesh_EvaluateCurvature): Gives info on curvature
 - Check solid mesh (/wiki/index.php?title=Mesh_EvaluateSolid): Checks the solid if it can be converted to a mesh
 - Boundings info... (/wiki/index.php?title=Mesh_BoundingBox): Evaluates the bounding box of a mesh
-  (/wiki/index.php?title=File:Mesh_Regular_Solid.png) Regular solid... (/wiki/index.php?title=Mesh_BuildRegularSolid) Create mesh primitives, like cubes, cylinders, cones, or spheres:
 -  (/wiki/index.php?title=File:Mesh_Cube.png) Create a mesh cube

-  (/wiki/index.php?title=File:Mesh_Cylinder.png) Create a mesh cylinder
 -  (/wiki/index.php?title=File:Mesh_Cone.png) Create a mesh cone
 -  (/wiki/index.php?title=File:Mesh_Sphere.png) Create a mesh sphere
 -  (/wiki/index.php?title=File:Mesh_Ellipsoid.png) Create a mesh ellipsoid
 -  (/wiki/index.php?title=File:Mesh_Torus.png) Create a mesh torus
- Do **Boolean** operations with meshes
 - Union (/wiki/index.php?title=Mesh_Union): Does a union (fusion) on meshes
 - Intersection (/wiki/index.php?title=Mesh_Intersection): Does an intersection (common) on meshes
 - Difference (/wiki/index.php?title=Mesh_Difference): Does a difference (cut) on meshes
- Merge (/wiki/index.php?title=Mesh_Merge): Merges meshes
 - Select Mesh (/wiki/index.php?title=Mesh_SelectMesh): Selects meshes
 -  (/wiki/index.php?title=File:Mesh_Cut.png) Cut mesh (/wiki/index.php?title=Mesh_Cut): Cut meshes along a line
 - Split Mesh (/wiki/index.php?title=Mesh_SplitMesh): Splits meshes
 -  (/wiki/index.php?title=File:Mesh_MakeSegment.png) Make segment (/wiki/index.php?title=Mesh_MakeSegment): Makes a segment
 - Trim mesh (/wiki/index.php?title=Mesh_TrimMesh): Trims meshes
 - Trim mesh with a plane (/wiki/index.php?title=Mesh_TrimMeshWithPlane): Trims meshes with a plane
 - Create mesh segments... (/wiki/index.php?title=Mesh_CreateMeshSegment): Creates mesh segments
 -  (/wiki/index.php?title=File:Mesh_CurvaturePlot.png) Curvature Plot (/wiki/index.php?title=Mesh_CurvaturePlot): Creates a curvature plot

These are only some of the basic operations currently present in the Mesh module interface.

More mesh tools are available in the OpenSCAD Workbench (/wiki/index.php?title=OpenSCAD_Workbench).

But the FreeCAD meshes can also be handled in many more ways by scripting (/wiki/index.php?title=Mesh_Scripting).

Links

- FreeCAD and Mesh Import (/wiki/index.php?title=FreeCAD_and_Mesh_Import)

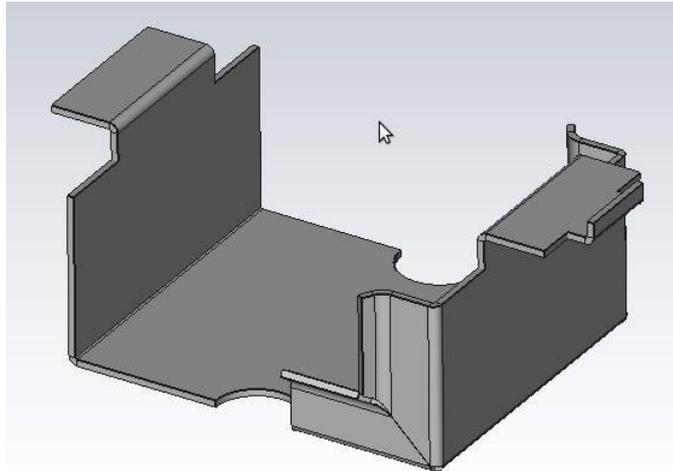
< previous: PartDesign Workbench (/wiki/index.php?title=PartDesign_Workbench)

next: OpenSCAD Module > (/wiki/index.php?title=OpenSCAD_Module)

Index (/wiki/index.php?title=Online_Help_Toc)

The Part workbench

The CAD capabilities of FreeCAD are based on the OpenCasCade (http://en.wikipedia.org/wiki/Open_CASCADE) kernel. The Part module allows FreeCAD to access and use the OpenCasCade objects and functions. OpenCasCade is a professional-level CAD kernel, that features advanced 3D geometry manipulation and objects. The Part objects, unlike Mesh Module (/wiki/index.php?title=Mesh_Module) objects, are much more complex, and therefore permit much more advanced operations, like coherent boolean operations, modifications history and parametric behaviour.



(/wiki/index.php?title=File:Part_example.jpg)

Example of Part shapes in FreeCAD

The tools

The Part module tools are all located in the **Part** menu that appears when you load the Part module.

Primitives

These are tools for creating primitive objects.

-  (/wiki/index.php?title=File:Part_Box.png) Box (/wiki/index.php?title=Part_Box): Draws a box by specifying its dimensions
-  (/wiki/index.php?title=File:Part_Cone.png) Cone (/wiki/index.php?title=Part_Cone): Draws a cone by specifying its dimensions
-  (/wiki/index.php?title=File:Part_Cylinder.png) Cylinder (/wiki/index.php?title=Part_Cylinder): Draws a cylinder by specifying its dimensions
-  (/wiki/index.php?title=File:Part_Sphere.png) Sphere (/wiki/index.php?title=Part_Sphere): Draws a sphere by specifying its dimensions
-  (/wiki/index.php?title=File:Part_Torus.png) Torus (/wiki/index.php?title=Part_Torus): Draws a torus (ring) by specifying its dimensions
-  (/wiki/index.php?title=File:Part_CreatePrimitives.png) CreatePrimitives (/wiki/index.php?title=Part_CreatePrimitives): A tool to create various parametric geometric primitives
-  (/wiki/index.php?title=File:Part_Shapebuilder.png) Shapebuilder (/wiki/index.php?title=Part_Shapebuilder): A tool to create more complex shapes from various parametric geometric primitives

Modifying objects

These are tools for modifying existing objects. They will allow you to choose which object to modify.

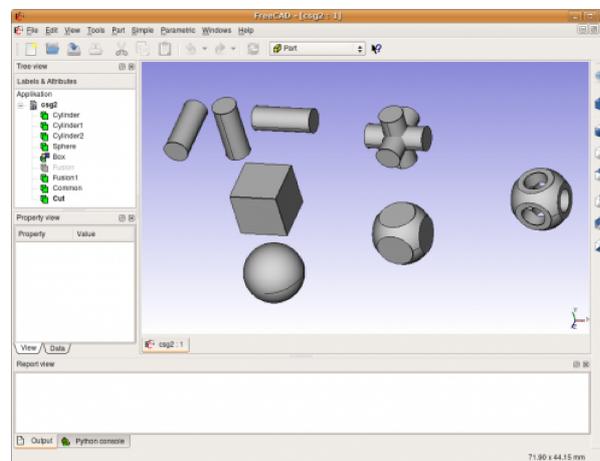
-  (/wiki/index.php?title=File:Part_Booleans.png) Booleans (/wiki/index.php?title=Part_Booleans): Performs boolean operations on objects
-  (/wiki/index.php?title=File:Part_Fuse.png) Fuse (/wiki/index.php?title=Part_Fuse): Fuses (unions) two objects
-  (/wiki/index.php?title=File:Part_Common.png) Common (/wiki/index.php?title=Part_Common): Extracts the common (intersection) part of two objects
-  (/wiki/index.php?title=File:Part_Cut.png) Cut (/wiki/index.php?title=Part_Cut): Cuts (subtracts) one object from another
-  (/wiki/index.php?title=File:Part_JoinConnect.png) Join features (/wiki/index.php?title=Part_CompJoinFeatures): smart booleans for walled objects (e.g., pipes) (v0.16)
 -  (/wiki/index.php?title=File:Part_JoinConnect.png) Connect (/wiki/index.php?title=Part_JoinConnect): Connects interiors of objects (v0.16)
 -  (/wiki/index.php?title=File:Part_JoinEmbed.png) Embed (/wiki/index.php?title=Part_JoinEmbed): Embeds a walled object into another walled object (v0.16)
 -  (/wiki/index.php?title=File:Part_JoinCutout.png) Cutout (/wiki/index.php?title=Part_JoinCutout): Creates a cutout in a wall of an object for another walled object (v0.16)
-  (/wiki/index.php?title=File:Part_Extrude.png) Extrude (/wiki/index.php?title=Part_Extrude): Extrudes planar faces of an object
-  (/wiki/index.php?title=File:Part_Fillet.png) Fillet (/wiki/index.php?title=Part_Fillet): Fillets (rounds) edges of an object
-  (/wiki/index.php?title=File:Part_Revolve.png) Revolve (/wiki/index.php?title=Part_Revolve): Creates a solid by revolving another object (not solid) around an axis
-  (/wiki/index.php?title=File:Part_Section.png) Section (/wiki/index.php?title=Part_Section): Creates a section by intersecting an object with a section plane
-  (/wiki/index.php?title=File:Part_SectionCross.png) Cross sections... (/wiki/index.php?title=Part_SectionCross):
-  (/wiki/index.php?title=File:Part_Chamfer.png) Chamfer (/wiki/index.php?title=Part_Chamfer): Chamfers edges of an object
-  (/wiki/index.php?title=File:Part_Mirror.png) Mirror (/wiki/index.php?title=Part_Mirror): Mirrors the selected object on a given mirror plane
-  (/wiki/index.php?title=File:Part_RuledSurface.png) Ruled Surface (/wiki/index.php?title=Part_RuledSurface):
-  (/wiki/index.php?title=File:Part_Sweep.png) Sweep (/wiki/index.php?title=Part_Sweep): Sweeps one or more profiles along a path
-  (/wiki/index.php?title=File:Part_Loft.png) Loft (/wiki/index.php?title=Part_Loft): Lofts from one profile to another

-  (/wiki/index.php?title=File:Part_Offset.png) Offset (/wiki/index.php?title=Part_Offset): Creates a scaled copy of the original object.
-  (/wiki/index.php?title=File:Part_Thickness.png) Thickness (/wiki/index.php?title=Part_Thickness): Assign a thickness to the faces of a shape.

Other tools

-  (/wiki/index.php?title=File:Part_ImportCAD.png) Import CAD (/wiki/index.php?title=Part_ImportCAD): This tool allows you to add a file *.IGES, *.STEP, *.BREP to the current document.
-  (/wiki/index.php?title=File:Part_ExportCAD.png) Export CAD (/wiki/index.php?title=Part_ExportCAD): This tool allows you to export a part object in a *.IGES, *.STEP, *.BREP file.
-  (/wiki/index.php?title=File:Part_ShapeFromMesh.png) Shape from Mesh (/wiki/index.php?title=Part_ShapeFromMesh): Creates a shape object from a mesh object.
- Convert to solid (/wiki/index.php?title=Part_ConvertToSolid): Converts a shape object to a solid object.
- Reverse shapes (/wiki/index.php?title=Part_ReverseShapes): Flips the normals of all faces of the selected object.
- Create simple copy (/wiki/index.php?title=Part_CreateSimpleCopy): Creates a simple copy of the selected object.
- Make compound (/wiki/index.php?title=Part_MakeCompound): Creates a compound from the selected objects.
-  (/wiki/index.php?title=File:Part_RefineShape.png) Refine shape (/wiki/index.php?title=Part_RefineShape): Cleans faces by removing unnecessary lines.
-  (/wiki/index.php?title=File:Part_CheckGeometry.png) Check geometry (/wiki/index.php?title=Part_CheckGeometry): Checks the geometry of selected objects for errors.
- Measure (/wiki/index.php?title=Std_Measure_Menu): Allows linear and angular measurement between points/edges/faces.

Boolean Operations



(/wiki/index.php?title=File:Part_BooleanOperations.png)

An example of union (Fuse), intersection (Common) and difference (Cut)

Explaining the concepts

In OpenCascade terminology, we distinguish between geometric primitives and (topological) shapes. A geometric primitive can be a point, a line, a circle, a plane, etc. or even some more complex types like a B-Spline curve or surface. A shape can be a vertex, an edge, a wire, a face, a solid or a compound of other shapes. The geometric primitives are not made to be directly displayed on the 3D scene, but rather to be used as building geometry for shapes. For example, an edge can be constructed from a line or from a portion of a circle.

We could say, to resume, that geometry primitive are "shapeless" building blocks, and shapes are the real spatial geometry built on it.

To get a complete list of all of them refer to the OCC documentation (<http://www.opencascade.org/org/doc/>) (Alternative: [sourcearchive.com](http://opencascade.sourceforge.com/documentation/6.3.0.dfs.1-1/classes.html) (<http://opencascade.sourceforge.com/documentation/6.3.0.dfs.1-1/classes.html>) and search for **Geom_*** (for geometry) and **TopoDS_*** (for shapes). There you can also read more about the differences between geometric objects and shapes. Please note that unfortunately the official OCC documentation is not available online (you must download an archive) and is mostly aimed at programmers, not at end-users. But hopefully you'll find enough information to get started here.

The geometric types actually can be divided into two major groups: curves and surfaces. Out of the curves (line, circle, ...) you can directly build an edge, out of the surfaces (plane, cylinder, ...) a face can be built. For example, the geometric primitive line is unlimited, i.e. it is defined by a base vector and a direction vector while its shape representation must be something limited by a start and end point. And a box -- a solid -- can be created by six limited planes.

From an edge or face you can also go back to its geometric primitive counter part.

Thus, out of shapes you can build very complex parts or, the other way round, extract all sub-shapes a more complex shape is made of.

Scripting

The main data structure used in the Part module is the BRep (http://en.wikipedia.org/wiki/Boundary_representation) data type from OpenCascade. Almost all contents and object types of the Part module are now available to python scripting. This includes geometric primitives, such as Line and Circle (or Arc), and the whole range of TopoShapes, like Vertexes, Edges, Wires, Faces, Solids and Compounds. For each of those objects, several creation methods exist, and for some of them, especially the TopoShapes, advanced operations like boolean union/difference/intersection are also available. Explore the contents of the Part module, as described in the FreeCAD Scripting Basics (/wiki/index.php?title=FreeCAD_Scripting_Basics) page, to know more.

Examples

To create a line element switch to the Python console and type in:

```
import Part,PartGui
doc=App.newDocument()
l=Part.Line()
l.StartPoint=(0.0,0.0,0.0)
l.EndPoint=(1.0,1.0,1.0)
doc.addObject("Part::Feature","Line").Shape=l.toShape()
doc.recompute()
```

Let's go through the above python example step by step:

```
import Part,PartGui
doc=App.newDocument()
```

loads the Part module and creates a new document

```
l=Part.Line()
l.StartPoint=(0.0,0.0,0.0)
l.EndPoint=(1.0,1.0,1.0)
```

Line is actually a line segment, hence the start and endpoint.

```
doc.addObject("Part::Feature", "Line").Shape=l.toShape()
```

This adds a Part object type to the document and assigns the shape representation of the line segment to the 'Shape' property of the added object. It is important to understand here that we used a geometric primitive (the Part.Line) to create a TopoShape out of it (the toShape() method). Only Shapes can be added to the document. In FreeCAD, geometry primitives are used as "building structures" for Shapes.

```
doc.recompute()
```

Updates the document. This also prepares the visual representation of the new part object.

Note that a Line can be created by specifying its start and endpoint directly in the constructor, for example Part.Line(point1,point2), or we can create a default line and set its properties afterwards, as we did here.

A circle can be created in a similar way:

```
import Part
doc = App.activeDocument()
c = Part.Circle()
c.Radius=10.0
f = doc.addObject("Part::Feature", "Circle")
f.Shape = c.toShape()
doc.recompute()
```

Note again, we used the circle (geometry primitive) to construct a shape out of it. We can of course still access our construction geometry afterwards, by doing:

```
s = f.Shape
e = s.Edges[0]
c = e.Curve
```

Here we take the shape of our object f, then we take its list of edges. In this case there will be only one because we made the whole shape out of a single circle, so we take only the first item of the Edges list, and we takes its curve. Every Edge has a Curve, which is the geometry primitive it is based on.

Head to the Topological data scripting (/wiki/index.php?title=Topological_data_scripting) page if you would like to know more.

Tutorials

- [Import from STL or OBJ \(/wiki/index.php?title=Import_from_STL_or_OBJ\)](/wiki/index.php?title=Import_from_STL_or_OBJ) : How to import STL/OBJ files in FreeCAD
- [Export to STL or OBJ \(/wiki/index.php?title=Export_to_STL_or_OBJ\)](/wiki/index.php?title=Export_to_STL_or_OBJ) : How to export STL/OBJ files from FreeCAD
- [Whiffle Ball tutorial \(/wiki/index.php?title=Whiffle_Ball_tutorial\)](/wiki/index.php?title=Whiffle_Ball_tutorial) : How to use the Part Module

< [previous: OpenSCAD Module \(/wiki/index.php?title=OpenSCAD_Module\)](/wiki/index.php?title=OpenSCAD_Module)

[next: Drawing Module > \(/wiki/index.php?title=Drawing_Module\)](/wiki/index.php?title=Drawing_Module)
[Index \(/wiki/index.php?title=Online_Help_Toc\)](/wiki/index.php?title=Online_Help_Toc)

The Drawing workbench

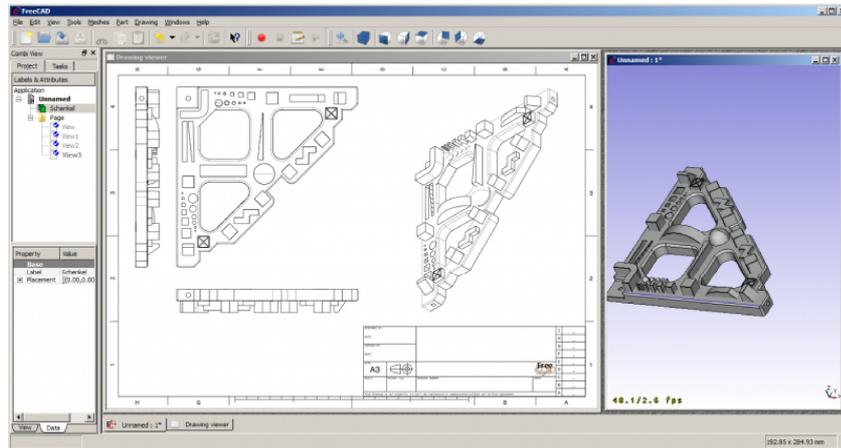
The Drawing module allows you to put your 3D work on paper. That is, to put views of your models in a 2D window and to insert that window in a drawing, for example a sheet with a border, a title and your logo and finally print that sheet. The Drawing module is currently under construction and more or less a technology preview!

GUI Tools

These are tools for creating, configuring and exporting 2D drawing sheets

-  (/wiki/index.php?title=File:Drawing_New.png) Open scalable vector graphic (/wiki/index.php?title=Drawing_Open_SVG): Opens a drawing sheet previously saved as an SVG file
-  (/wiki/index.php?title=File:Drawing_Landscape_A3.png) New A3 landscape drawing (/wiki/index.php?title=Drawing_Landscape_A3): Creates a new drawing sheet from FreeCAD's default A3 template
-  (/wiki/index.php?title=File:Drawing_View.png) Insert a view (/wiki/index.php?title=Drawing_View): Inserts a view of the selected object in the active drawing sheet
-  (/wiki/index.php?title=File:Drawing_Annotation.png) Annotation (/wiki/index.php?title=Drawing_Annotation): Adds an annotation to the current drawing sheet
-  (/wiki/index.php?title=File:Drawing_Clip.png) Clip (/wiki/index.php?title=Drawing_Clip): Adds a clip group to the current drawing sheet
-  (/wiki/index.php?title=File:Drawing_Openbrowser.png) Open Browser (/wiki/index.php?title=Drawing_Openbrowser): Opens a preview of the current sheet in the browser
-  (/wiki/index.php?title=File:Drawing_Orthoviews.png) Ortho Views (/wiki/index.php?title=Drawing_Orthoviews): Automatically creates orthographic views of an object on the current drawing sheet
-  (/wiki/index.php?title=File:Drawing_Symbol.png) Symbol (/wiki/index.php?title=Drawing_Symbol): Adds the contents of a SVG file as a symbol on the current drawing sheet
-  (/wiki/index.php?title=File:Drawing_DraftView.png) Draft View (/wiki/index.php?title=Drawing_DraftView): Inserts a special Draft view of the selected object in the current drawing sheet
-  (/wiki/index.php?title=File:Drawing_SpreadsheetView.png) Spreadsheet View (/wiki/index.php?title=Drawing_SpreadsheetView): Inserts a view of a selected spreadsheet in the current drawing sheet
-  (/wiki/index.php?title=File:Drawing_Save.png) Save sheet (/wiki/index.php?title=Drawing_Save): Saves the current sheet as a SVG file
- Project Shape (/wiki/index.php?title=Drawing_ProjectShape): Creates a projection of the selected object (Source) in the 3D view.

Note The Draft View (/wiki/index.php?title=Draft_Drawing) tool is used mainly to place Draft objects on paper. It has a couple of extra capabilities over the standard Drawing tools, and supports specific objects like Draft dimensions (/wiki/index.php?title=Draft_Dimension).



(/wiki/index.php?title=File:Drawing_extraction.png)

In the picture you see the main concepts of the Drawing module. The document contains a shape object (Schenkel) which we want to extract to a drawing. Therefore a "Page" is created. A page gets instantiated through a template, in this case the "A3_Landscape" template. The template is an SVG document which can hold your usual page frame, your logo or comply to your presentation standards.

In this page we can insert one or more views. Each view has a position on the page (Properties X,Y), a scale factor (Property scale) and additional properties. Every time the page or the view or the referenced object changes the page gets regenerated and the page display updated.

Scripting

At the moment the end user(GUI) workflow is very limited, so the scripting API is more interesting. Here follow examples on how to use the scripting API of the drawing module.

Here a script that can easily fill the Macro_CartoucheFC (/wiki/index.php?title=Macro_CartoucheFC) leaf FreeCAD A3_Landscape.

Simple example

First of all you need the Part and the Drawing module:

```
import FreeCAD, Part, Drawing
```

Create a small sample part

```
Part.show(Part.makeBox(100,100,100).cut(Part.makeCylinder(80,100)).cut(Part.makeBox(90,40,100)).cut(Part.makeBox(20,85,100)))
```

Direct projection. The G0 means hard edge, the G1 is tangent continuous.

```
Shape = App.ActiveDocument.Shape.Shape
[visibleG0,visibleG1,hiddenG0,hiddenG1] = Drawing.project(Shape)
print "visible edges:", len(visibleG0.Edges)
print "hidden edges:", len(hiddenG0.Edges)
```

Everything was projected on the Z-plane:

```
print "Bnd Box shape: X=",Shape.BoundingBox.XLength," Y=",Shape.BoundingBox.YLength," Z=",Shape.BoundingBox.ZLength
print "Bnd Box project: X=",visibleG0.BoundingBox.XLength," Y=",visibleG0.BoundingBox.YLength," Z=",visibleG0.BoundingBox.ZLength
```

Different projection vector

```
[visibleG0,visibleG1,hiddenG0,hiddenG1] = Drawing.project(Shape,App.Vector(1,1,1))
```

Project to SVG

```
resultSVG = Drawing.projectToSVG(Shape,App.Vector(1,1,1))
print resultSVG
```

The parametric way

Create the body

```
import FreeCAD
import Part
import Drawing

# Create three boxes and a cylinder
App.ActiveDocument.addObject("Part::Box", "Box")
App.ActiveDocument.Box.Length=100.00
App.ActiveDocument.Box.Width=100.00
App.ActiveDocument.Box.Height=100.00

App.ActiveDocument.addObject("Part::Box", "Box1")
App.ActiveDocument.Box1.Length=90.00
App.ActiveDocument.Box1.Width=40.00
App.ActiveDocument.Box1.Height=100.00

App.ActiveDocument.addObject("Part::Box", "Box2")
App.ActiveDocument.Box2.Length=20.00
App.ActiveDocument.Box2.Width=85.00
App.ActiveDocument.Box2.Height=100.00

App.ActiveDocument.addObject("Part::Cylinder", "Cylinder")
App.ActiveDocument.Cylinder.Radius=80.00
App.ActiveDocument.Cylinder.Height=100.00
App.ActiveDocument.Cylinder.Angle=360.00
# Fuse two boxes and the cylinder
App.ActiveDocument.addObject("Part::Fuse", "Fusion")
App.ActiveDocument.Fusion.Base = App.ActiveDocument.Cylinder
App.ActiveDocument.Fusion.Tool = App.ActiveDocument.Box1

App.ActiveDocument.addObject("Part::Fuse", "Fusion1")
App.ActiveDocument.Fusion1.Base = App.ActiveDocument.Box2
App.ActiveDocument.Fusion1.Tool = App.ActiveDocument.Fusion
# Cut the fused shapes from the first box
App.ActiveDocument.addObject("Part::Cut", "Shape")
App.ActiveDocument.Shape.Base = App.ActiveDocument.Box
App.ActiveDocument.Shape.Tool = App.ActiveDocument.Fusion1
# Hide all the intermediate shapes
Gui.ActiveDocument.Box.Visibility=False
Gui.ActiveDocument.Box1.Visibility=False
Gui.ActiveDocument.Box2.Visibility=False
Gui.ActiveDocument.Cylinder.Visibility=False
Gui.ActiveDocument.Fusion.Visibility=False
Gui.ActiveDocument.Fusion1.Visibility=False
```

Insert a Page object and assign a template

```
App.ActiveDocument.addObject('Drawing::FeaturePage', 'Page')
App.ActiveDocument.Page.Template = App.getResourceDir()+ 'Mod/Drawing/Templates/A3_Landscape.svg'
```

Create a view on the "Shape" object, define the position and scale and assign it to a Page

```
App.ActiveDocument.addObject('Drawing::FeatureViewPart', 'View')
App.ActiveDocument.View.Source = App.ActiveDocument.Shape
App.ActiveDocument.View.Direction = (0.0,0.0,1.0)
App.ActiveDocument.View.X = 10.0
App.ActiveDocument.View.Y = 10.0
App.ActiveDocument.Page.addObject(App.ActiveDocument.View)
```

Create a second view on the same object but this time the view will be rotated by 90 degrees.

```
App.ActiveDocument.addObject('Drawing::FeatureViewPart', 'ViewRot')
App.ActiveDocument.ViewRot.Source = App.ActiveDocument.Shape
App.ActiveDocument.ViewRot.Direction = (0.0,0.0,1.0)
App.ActiveDocument.ViewRot.X = 290.0
App.ActiveDocument.ViewRot.Y = 30.0
App.ActiveDocument.ViewRot.Scale = 1.0
App.ActiveDocument.ViewRot.Rotation = 90.0
App.ActiveDocument.Page.addObject(App.ActiveDocument.ViewRot)
```

Create a third view on the same object but with an isometric view direction. The hidden lines are activated too.

```

App.ActiveDocument.addObject('Drawing::FeatureViewPart', 'ViewIso')
App.ActiveDocument.ViewIso.Source = App.ActiveDocument.Shape
App.ActiveDocument.ViewIso.Direction = (1.0,1.0,1.0)
App.ActiveDocument.ViewIso.X = 335.0
App.ActiveDocument.ViewIso.Y = 140.0
App.ActiveDocument.ViewIso.ShowHiddenLines = True
App.ActiveDocument.Page.addObject(App.ActiveDocument.ViewIso)

```

Change something and update. The update process changes the view and the page.

```

App.ActiveDocument.View.X = 30.0
App.ActiveDocument.View.Y = 30.0
App.ActiveDocument.View.Scale = 1.5
App.ActiveDocument.recompute()

```

Accessing the bits and pieces

Get the SVG fragment of a single view

```

ViewSVG = App.ActiveDocument.View.ViewResult
print ViewSVG

```

Get the whole result page (it's a file in the document's temporary directory, only read permission)

```

print "Resulting SVG document: ",App.ActiveDocument.Page.PageResult
file = open(App.ActiveDocument.Page.PageResult,"r")
print "Result page is ",len(file.readlines())," lines long"

```

Important: free the file!

```

del file

```

Insert a view with your own content:

```

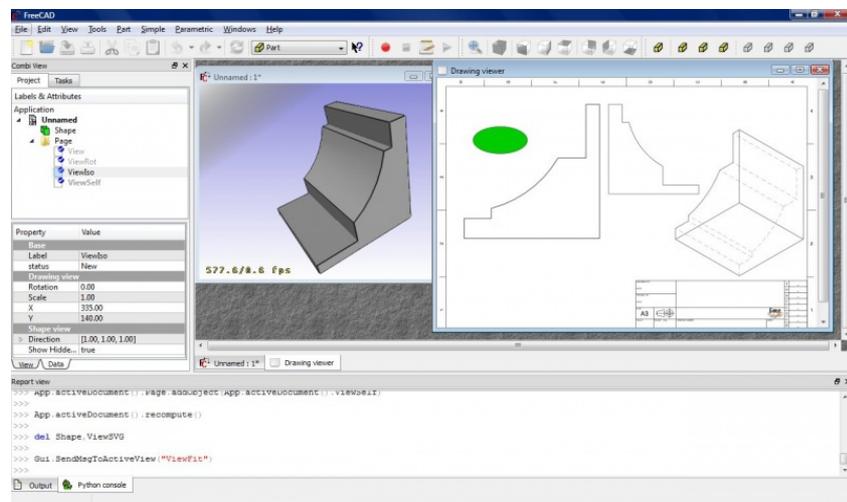
App.ActiveDocument.addObject('Drawing::FeatureView', 'ViewSelf')
App.ActiveDocument.ViewSelf.ViewResult = """<g id="ViewSelf"
stroke="rgb(0, 0, 0)"
stroke-width="0.35"
stroke-linecap="butt"
stroke-linejoin="miter"
transform="translate(30,30)"
fill="#00cc00"
>

<ellipse cx="40" cy="40" rx="30" ry="15"/>
</g>"""
App.ActiveDocument.Page.addObject(App.ActiveDocument.ViewSelf)
App.ActiveDocument.recompute()

del ViewSVG

```

That leads to the following result:



(/wiki/index.php?title=File:DrawingScriptResult.jpg)

General Dimensioning and Tolerancing

Drawing dimensions and tolerances are still under development but you can get some basic functionality with a bit of work.

First you need to get the `gdtsvg` python module from here (WARNING: This could be broken at any time!):

<https://github.com/jcc242/FreeCAD> (<https://github.com/jcc242/FreeCAD>)

To get a feature control frame, try out the following:

```
import gdtsvg as g # Import the module, I like to give it an easy handle
ourFrame = g.ControlFrame("0", "0", g.Perpendicularity(), ".5", g.Diameter(), g.ModifyingSymbols("M"), "A",
                        g.ModifyingSymbols("F"), "B", g.ModifyingSymbols("L"), "C", g.ModifyingSymbols("I")
)
```

Here is a good breakdown of the contents of a feature control frame: <http://www.cadblog.net/adding-geometric-tolerances.htm> (<http://www.cadblog.net/adding-geometric-tolerances.htm>)

The parameters to pass to control frame are:

1. X-coordinate in SVG-coordinate system (type string)
2. Y-coordinate in SVG-coordinate system (type string)
3. The desired geometric characteristic symbol (tuple, svg string as first, width of symbol as second, height of symbol as third)
4. The tolerance (type string)
5. (optional) The diameter symbol (tuple, svg string as first, width of symbol as second, height of symbol as third)
6. (optional) The condition modifying material (tuple, svg string as first, width of symbol as second, height of symbol as third)
7. (optional) The first datum (type string)
8. (optional) The first datum's modifying condition (tuple, svg string as first, width of symbol as second, height of symbol as third)
9. (optional) The second datum (type string)
10. (optional) The second datum's modifying condition (tuple, svg string as first, width of symbol as second, height of symbol as third)
11. (optional) The third datum (type string)
12. (optional) The third datum's material condition (tuple, svg string as first, width of symbol as second, height of symbol as third)

The `ControlFrame` function returns a type containing (svg string, overall width of control frame, overall height of control frame)

To get a dimension, try out the following:

```
import gdtsvg
ourDimension = linearDimension(point1, point2, textpoint, dimensiontext, linestyle=getStyle("visible"),
                             arrowstyle=getStyle("filled"), textstyle=getStyle("text"))
```

Inputs for linear dimension are:

1. `point1`, an (x,y) tuple with svg-coordinates, this is one of the points you would like to dimension between
2. `point2`, an (x,y) tuple with svg-coordinates, this is the second point you would like to dimension between

3. `textpoint`, an (x,y) tuple of svg-coordinates, this is where the text of your dimension will be
4. `dimensiontext`, a string containing the text you want the dimension to say
5. `linestyle`, a string containing svg (i.e. css) styles, using the `getStyle` function to retrieve a preset string, for styling the how the lines look
6. `arrowstyle`, a string containing svg (i.e. css) styles, using the `getStyle` function to retrieve a preset string, for styling how the arrows look
7. `textstyle`, a string containing svg (i.e. css) styles, using the `getStyle` function to retrieve a preset string, for styling how the text looks

With those two, you can proceed as above for displaying them on the drawing page. This module is very buggy and can be broken at any given moment, bug reports are welcome on the github page for now, or contact `jcc242` on the forums if you post a bug somewhere else.

Templates

FreeCAD comes bundled with a set of default templates, but you can find more on the Drawing templates (/wiki/index.php?title=Drawing_templates) page.

Extending the Drawing Module

Some notes on the programming side of the drawing module will be added to the Drawing Documentation (/wiki/index.php?title=Drawing_Documentation) page. This is to help quickly understand how the drawing module works, enabling programmers to rapidly start programming for it.

Tutorials

- [Drawing tutorial \(/wiki/index.php?title=Drawing_tutorial\)](/wiki/index.php?title=Drawing_tutorial)

External links

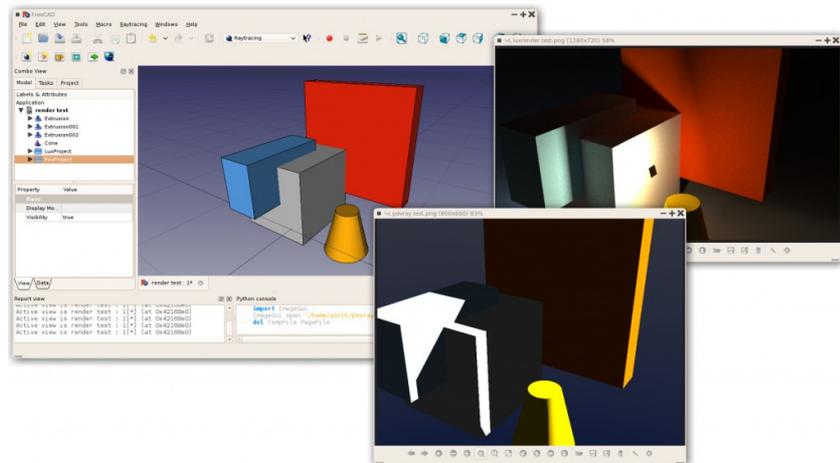
- [Intro to mechanical drawing on Youtube - by Normal Universe \(https://www.youtube.com/watch?v=1Hm5Zyjmic\)](https://www.youtube.com/watch?v=1Hm5Zyjmic)

< [previous: Part Module \(/wiki/index.php?title=Part_Module\)](/wiki/index.php?title=Part_Module) [Index next: Raytracing Module \(/wiki/index.php?title=Raytracing_Module\)](/wiki/index.php?title=Raytracing_Module) > [\(/wiki/index.php?title=Online_Help_Toc\)](/wiki/index.php?title=Online_Help_Toc)

The Raytracing workbench

The Raytracing module is used to generate photorealistic images of your models by rendering them with an external renderer. The Raytracing workbench works with templates (/wiki/index.php?title=Raytracing_Module#Templates), the same way as the Drawing workbench (/wiki/index.php?title=Drawing_Module), by allowing you to

create a Raytracing project in which you add views of your objects. The project can then be exported to a ready-to-render file, or be rendered directly.



(/wiki/index.php?title=File:Raytracing_example.jpg)

Currently, two renderers are supported: povray (<http://en.wikipedia.org/wiki/POV-Ray>) and luxrender (<http://en.wikipedia.org/wiki/LuxRender>). To be able to render directly from FreeCAD, at least one of those renderers must be installed on your system, and its path must be configured in the FreeCAD Raytracing preferences. Without any renderer installed, though, you are still able to export a scene file that can be used in any of those renderers later, or on another machine.

The raytracing workbench works with templates (/wiki/index.php?title=Raytracing_Module#Templates), which are complete scene files for the given external renderer, including lights and possibly additional geometry such as ground planes. These scene files contain placeholders, where FreeCAD will insert the position of the camera, and geometry and materials information of each of the objects you insert in the project. That modified scene file is what is then exported to the external renderer.

Tools

Raytracing project tools

These are the main tools for exporting your 3D work to external renderers

-  (/wiki/index.php?title=File:Raytracing_New.png) New PovRay project (/wiki/index.php?title=Raytracing_New): Insert new PovRay project in the document
-  (/wiki/index.php?title=File:Raytracing_Lux.png) New LuxRender project (/wiki/index.php?title=Raytracing_Lux): Insert new LuxRender project in the document
-  (/wiki/index.php?title=File:Raytracing_InsertPart.png) Insert part (/wiki/index.php?title=Raytracing_InsertPart): Insert a view of a Part in a raytracing project
-  (/wiki/index.php?title=File:Raytracing_ResetCamera.png) Reset camera (/wiki/index.php?title=Raytracing_ResetCamera): Matches the camera position of a raytracing project to the current view
-  (/wiki/index.php?title=File:Raytracing_ExportProject.png) Export project (/wiki/index.php?title=Raytracing_ExportProject): Exports a raytracing project to a scene file for rendering in an external renderer
-  (/wiki/index.php?title=File:Raytracing_Render.png) Render (/wiki/index.php?title=Raytracing_Render): Renders a raytracing project with an external renderer

Utilities

These are helper tools to perform specific tasks manually

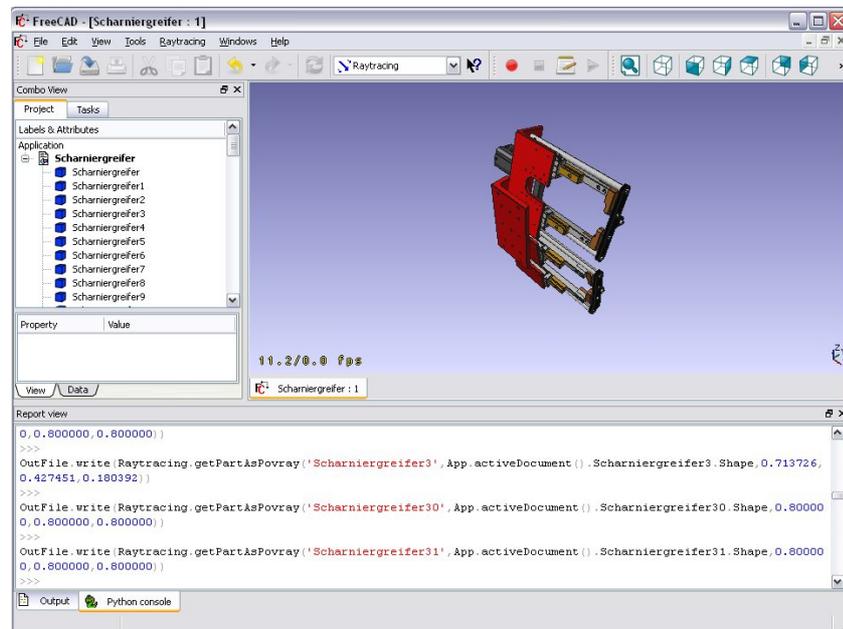
-  (/wiki/index.php?title=File:Raytracing_Export.png) Export view to povray (/wiki/index.php?title=Raytracing_Export): Write the active 3D view with camera and all its content to a povray file
-  (/wiki/index.php?title=File:Raytracing_Camera.png) Export camera to povray (/wiki/index.php?title=Raytracing_Camera): Export the camera position of the active 3D view in POV-Ray format to a file
-  (/wiki/index.php?title=File:Raytracing_Part.png) Export part to povray (/wiki/index.php?title=Raytracing_Part): Write the selected Part (object) as a povray file

Typical workflow

1. Create or open a FreeCAD project, add some Part-based (/wiki/index.php?title=Part_Module) objects (meshes are currently not supported)
2. Create a Raytracing project (luxrender or povray)
3. Select the objects you wish to add to the raytracing project and add them to the project with the "Insert Part" tool
4. Export or render directly

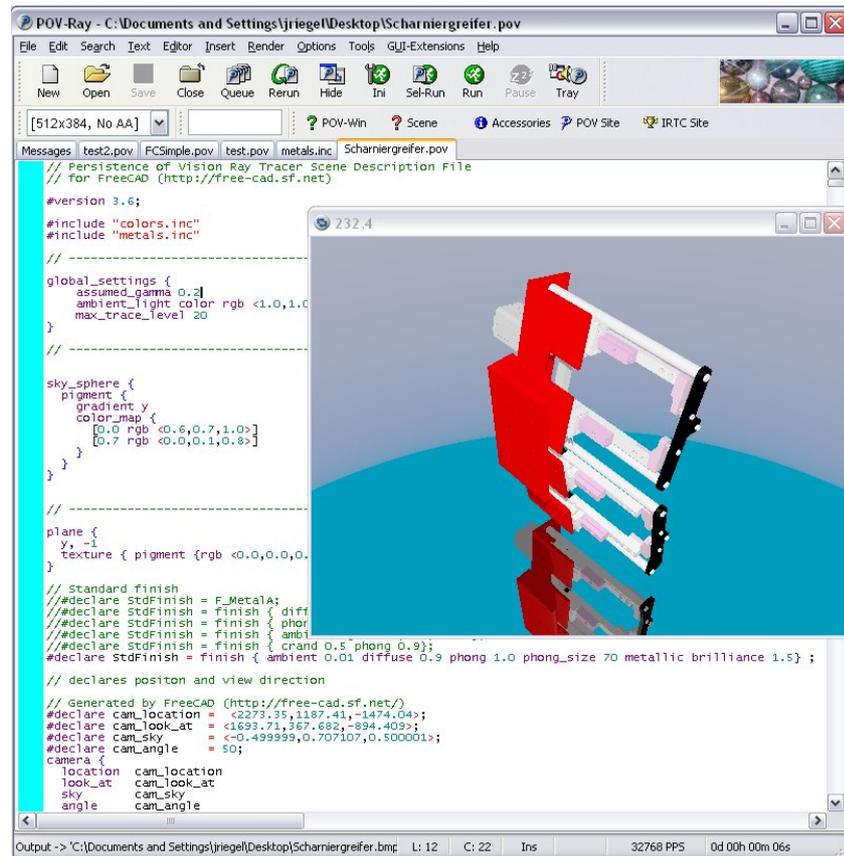
Creating a povray file manually

The utility tools described above allow you to export the current 3D view and all of its content to a Povray (<http://www.povray.org/>) file. First, you must load or create your CAD data and position the 3D View orientation as you wish. Then choose "Utilities->Export View..." from the raytracing menu.



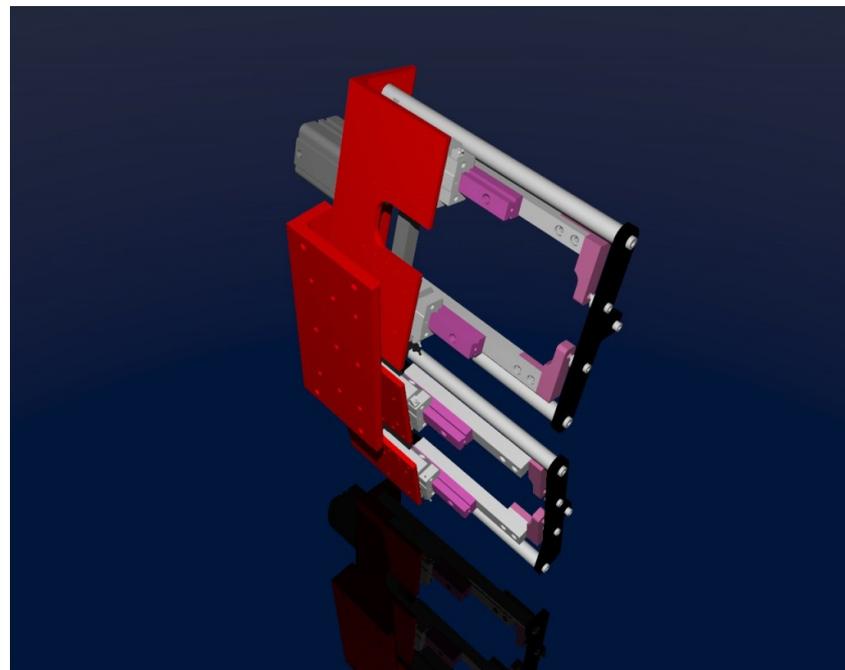
(/wiki/index.php?title=File:FreeCAD_Raytracing.jpg)

You will be asked for a location to save the resulting *.pov file. After that you can open it in Povray (<http://www.povray.org/>) and render:



(/wiki/index.php?title=File:Povray.jpg)

As usual in a renderer you can make big and nice pictures:



(/wiki/index.php?title=File:Scharniergreifer_render.jpg)

Scripting

Outputting render files

The Raytracing and RaytracingGui modules provide several methods to write scene contents as povray or luxrender data. The most useful are `Raytracing.getPartAsPovray()` and `Raytracing.getPartAsLux()` to render a FreeCAD Part object into a povray or luxrender definition, and

RaytracingGui.povViewCamera() and RaytracinGui.luxViewCamera() to get the current point of view of the FreeCAD 3D window into povray or luxrender format.

Here is how to write a povray file from python, assuming your document contains a "Box" object:

```
import Raytracing,RaytracingGui
OutFile = open('C:/Documents and Settings/jriegel/Desktop/test.pov','w')
OutFile.write(open(App.getResourceDir()+ 'Mod/Raytracing/Templates/ProjectStd.pov').read())
OutFile.write(RaytracingGui.povViewCamera())
OutFile.write(Raytracing.getPartAsPovray('Box',App.activeDocument().Box.Shape,0.800000,0.800000,0,0.800000))
OutFile.close()
del OutFile
```

And the same for luxrender:

```
import Raytracing,RaytracingGui
OutFile = open('C:/Documents and Settings/jriegel/Desktop/test.lxs','w')
OutFile.write(open(App.getResourceDir()+ 'Mod/Raytracing/Templates/LuxClassic.lxs').read())
OutFile.write(RaytracingGui.luxViewCamera())
OutFile.write(Raytracing.getPartAsLux('Box',App.activeDocument().Box.Shape,0.800000,0.800000,0.800000))
OutFile.close()
del OutFile
```

Creating a custom render object

Apart from standard povray and luxrender view objects that provide a view of an existing Part object, and that can be inserted in povray and luxrender projects respectively, a third object exist, called RaySegment, that can be inserted either in povray or luxrender projects. That RaySegment object is not linked to any of the FreeCAD objects, and can contain custom povray or luxrender code, that you might wish to insert into your raytracing project. You can also use it, for example, to output your FreeCAD objects a certain way, if you are not happy with the standard way. You can create and use it like this from the python console:

```
myRaytracingProject = FreeCAD.ActiveDocument.PovProject
myCustomRenderObject = FreeCAD.ActiveDocument.addObject("Raytracing::RaySegment","myRenderObject")
myRaytracingProject.addObject(myCustomRenderObject)
myCustomRenderObject.Result = "// Hello from python!"
```

Links

POVRay

- <http://www.spiritone.com/~english/cyclopedia/>
(<http://www.spiritone.com/~english/cyclopedia/>)
- <http://www.povray.org/> (<http://www.povray.org/>)
- <http://en.wikipedia.org/wiki/POV-Ray>
(<http://en.wikipedia.org/wiki/POV-Ray>)

Luxrender

- <http://www.luxrender.net/> (<http://www.luxrender.net/>)

Future possible renderers to implement

- <http://www.yafaray.org/> (<http://www.yafaray.org/>)
- <http://www.mitsuba-renderer.org/> (<http://www.mitsuba-renderer.org/>)
- <http://www.kerkythea.net/> (<http://www.kerkythea.net/>)
- <http://www.artofillusion.org/> (<http://www.artofillusion.org/>)

Currently there is a new Renderer Workbench in development to support multiple back-ends such as Lux Renderer and Yafaray. Information for using the development version can be viewed at [Render_project \(/wiki/index.php?title=Render_project\)](/wiki/index.php?title=Render_project)

For Development status of the Render Module look here [Raytracing_project \(/wiki/index.php?title=Raytracing_project\)](/wiki/index.php?title=Raytracing_project)

Templates

FreeCAD comes with a couple of default templates for povray and luxrender, but you can easily create your own. All you need to do is to create a scene file for the given renderer, then edit it manually with a text editor to insert special tags that FreeCAD will recognize and where it will insert its contents (camera and objects data)

Povray

Povray scene files (with extension .pov) can be created manually with a text editor (povray is made primarily to be used as a scripting language), but also with a wide range of 3D applications, such as blender (<http://www.blender.org>). On the povray website (<http://www.povray.org/>) you can find further information and a list of applications able to produce .pov files.

When you have a .pov file ready, you need to open it with a text editor, and do two operations:

1. Strip out the camera information, because FreeCAD will place its own camera data. To do so, locate a text block like this: `camera { ... }`, which describes the camera parameters, and delete it (or put `/// in front of each line to comment them out).`
2. Insert the following line somewhere: `//RaytracingContent`. This is where FreeCAD will insert its contents (camera and objects data). You can, for example, put this line at the very end of the file.

Note that FreeCAD will also add some declarations, that you can use in your template, after the `//RaytracingContent` tag. These are:

- `cam_location`: the location of the camera
- `cam_look_at`: the location of the target point of the camera
- `cam_sky`: the up vector of the camera.
- `cam_angle`: the angle of the camera

If you want, for example, to place a lamp above the camera, you can use this:

```
light_source {
  cam_location + cam_angle * 100
  color rgb <10, 10, 10>
}
```

Luxrender

Luxrender scene files (with extension .lxs) can either be single files, or a master .lxs file that includes world definition (.lxw), material definition (.lxm) and geometry definition (.lxo) files. You can work with both styles, but it is also easy to transform a group of 4 files in a single .lxs file, by copying the contents of each .lxw, .lxm and .lxo file and pasting it at the point where that file is inserted in the master .lxs file.

Luxrender scene files are hard to produce by hand, but are easy to produce with many 3D applications such as blender (<http://www.blender.org>). On the luxrender website (<http://www.luxrender.net>), you'll find more information and plugins for the main 3D applications out there.

If you will work with separated .lxw, .lxm and .lxo files, beware that the final .lxs exported by FreeCAD might be at a different location than the template file, and therefore these files might not be found by Luxrender at render time. In this case you should or copy these files to the location of your final file, or edit their paths in the exported .lxs file.

If you are exporting a scene file from blender, and wish to merge everything into one single file, you will need to perform one step before exporting: By default, the luxrender exporter in blender exports all mesh geometry as separate .ply files, instead of placing the mesh geometry directly inside the .lxo file. To change that behaviour, you need to select each of your meshes in blender, go to the "mesh" tab and set the option "export as" to "luxrender mesh" for each one of them.

After you have your scene file ready, to turn it into a FreeCAD template, you need to perform the following steps:

1. Locate the camera position, a single line that begins with `LookAt`, and delete it (or place a "#" at the beginning of the line to comment it out)
2. At that place, insert the following line: `#RaytracingCamera`
3. At a desired point, for example just after the end of the materials definition, before the geometry information, or at the very end, just before the final `WorldEnd` line, insert the following line: `#RaytracingContent`. That is where FreeCAD will insert its own objects.

Note that in luxrender, the objects stored in a scene file can define transformation matrixes, that perform location, rotation or scaling operations. These matrixes can stack and affect everything that come after them, so, by placing your `#RaytracingContent` tag at the end of the file, you might see your FreeCAD objects affected by a transformation matrix placed earlier in the template. To make sure that this doesn't happen, place your `#RaytracingContent` tag before any other geometry object present in the template. FreeCAD itself won't define any of those transformation matrixes.

Exporting to Kerkythea

Although direct export to the Kerkythea XML-File-Format is not supported yet, you can export your Objects as Mesh-Files (.obj) and then import them in Kerkythea.

- if using Kerkythea for Linux, remember to install the WINE-Package (needed by Kerkythea for Linux to run)
- you can convert your models with the help of the mesh workbench to meshes and then export these meshes as .obj-files
- If your mesh-export resulted in errors (flip of normals, holes ...) you may try your luck with netfabb studio basic (<http://www.netfabb.com/downloadcenter.php?basic=1>)

Free for personal use, available for Windows, Linux and Mac OSX. It has standard repair tools which will repair you model in most cases.

- another good program for mesh analysing/repairing is Meshlab (<http://sourceforge.net/projects/meshlab/>)

Open Source, available for Windows, Linux and Mac OSX.

It has standard repair tools which will repair you model in most cases (fill holes, re-orient normals, etc.)

- you can use "make compound" and then "make single copy" or you can fuse solids to group them before converting to meshes
- remember to set in Kerkythea an import-factor of 0.001 for obj-modeler, since Kerkythea expects the obj-file to be in m (but standard units-scheme in FreeCAD is mm)

Within WIndows 7 64-bit Kerkythea does not seem to be able to save these settings.

So remember to do that each time you start Kerkythea

- if importing multiple objects in Kerkythea you can use the "File > Merge" command in Kerkythea

Links

- Render project (/wiki/index.php?title=Render_project)
- Raytracing tutorial (/wiki/index.php?title=Raytracing_tutorial)

< [previous: Drawing Module](/wiki/index.php?title=Drawing_Module) (/wiki/index.php?title=Drawing_Module)

[Index next: Image Module](/wiki/index.php?title=Image_Module) > (/wiki/index.php?title=Image_Module)
(/wiki/index.php?title=Online_Help_Toc)

The Image workbench

The image module manages different types of bitmap images (http://en.wikipedia.org/wiki/Raster_graphics), and lets you open them in FreeCAD.

Currently, the modules lets you open .bmp, .jpg, .png and .xpm file formats in a separate viewer window.

The image workbenches also allows you to import an image on a plane in the 3D-space of FreeCAD. This function is available via the second button of the image workbench.  (/wiki/index.php?title=File:Image_Import.png).

The imported image can be attached like a sketch to one of the main three planes (XY/XZ/YZ) with positive or negativ offset.

This function is only available if you have opened a FreeCAD document.

The image can be moved in 3D-space by editing the placement in the Property editor (/wiki/index.php?title=Property_editor).

The major use is tracing over the image, in order to generate a new part at using the image as template.

The image is imported with 1 pixel = 1mm. Therefore it is recommended to have the imported image in a reasonable resolution. The image can be scaled by editing the "XSize" and "YSize" values in the Property editor (/wiki/index.php?title=Property_editor). The image can be also moved by editing the X/Y/Z-values in the Placement-Tab. The image can also be rotated around any axis by using the placement-dialogue.

Tip:

Tracing with sketcher elements over an image works best if the image has a small (negative) offset to the sketch plane.

You can set an offset of -0,1 mm at import or later by editing the placement of the image.

Tools

 (/wiki/index.php?title=File:Image_Import.png) Image Import
(/wiki/index.php?title=Image_Import)

< [previous: Raytracing Module](/wiki/index.php?title=Raytracing_Module) (/wiki/index.php?title=Raytracing_Module)

Index [next: Draft Module](/wiki/index.php?title=Draft_Module) > (/wiki/index.php?title=Draft_Module)
(/wiki/index.php?title=Online_Help_Toc)

The Draft workbench

The Draft workbench allows to quickly draw simple 2D objects in the current document, and offers several tools to modify them afterwards. Some of these tools also work on all other FreeCAD objects, not only those created with the Draft workbench. It also provides a complete snapping system, and several utilities to manage objects and settings.

Drawing objects

These are tools for creating objects.

-  (/wiki/index.php?title=File:Draft_Line.png) Line (/wiki/index.php?title=Draft_Line): Draws a line segment between 2 points
-  (/wiki/index.php?title=File:Draft_Wire.png) Wire (/wiki/index.php?title=Draft_Wire): Draws a line made of multiple line segments (polyline)
-  (/wiki/index.php?title=File:Draft_Circle.png) Circle (/wiki/index.php?title=Draft_Circle): Draws a circle from center and radius
-  (/wiki/index.php?title=File:Draft_Arc.png) Arc (/wiki/index.php?title=Draft_Arc): Draws an arc segment from center, radius, start angle and end angle
-  (/wiki/index.php?title=File:Draft_Ellipse.png) Ellipse (/wiki/index.php?title=Draft_Ellipse): Draws an ellipse from two corner points
-  (/wiki/index.php?title=File:Draft_Polygon.png) Polygon (/wiki/index.php?title=Draft_Polygon): Draws a regular polygon from a center and a radius
-  (/wiki/index.php?title=File:Draft_Rectangle.png) Rectangle (/wiki/index.php?title=Draft_Rectangle): Draws a rectangle from 2 opposite points
- **A** (/wiki/index.php?title=File:Draft_Text.png) Text (/wiki/index.php?title=Draft_Text): Draws a multi-line text annotation
-  (/wiki/index.php?title=File:Draft_Dimension.png) Dimension (/wiki/index.php?title=Draft_Dimension): Draws a dimension annotation
-  (/wiki/index.php?title=File:Draft_BSpline.png) BSpline (/wiki/index.php?title=Draft_BSpline): Draws a B-Spline from a series of points
-  (/wiki/index.php?title=File:Draft_Point.png) Point (/wiki/index.php?title=Draft_Point): Inserts a point object
- **S** (/wiki/index.php?title=File:Draft_ShapeString.png) ShapeString (/wiki/index.php?title=Draft_ShapeString): The ShapeString tool inserts a compound shape representing a text string at a given point in the current document

-  (/wiki/index.php?title=File:Draft_Facebinder.png) Facebinder (/wiki/index.php?title=Draft_Facebinder): Creates a new object from selected faces on existing objects
-  (/wiki/index.php?title=File:Draft_BezCurve.png) Bezier Curve (/wiki/index.php?title=Draft_BezCurve): Draws a Bezier curve from a series of points

Modifying objects

These are tools for modifying existing objects. They work on selected objects, but if no object is selected, you will be invited to select one.

-  (/wiki/index.php?title=File:Draft_Move.png) Move (/wiki/index.php?title=Draft_Move): Moves object(s) from one location to another
-  (/wiki/index.php?title=File:Draft_Rotate.png) Rotate (/wiki/index.php?title=Draft_Rotate): Rotates object(s) from a start angle to an end angle
-  (/wiki/index.php?title=File:Draft_Offset.png) Offset (/wiki/index.php?title=Draft_Offset): Moves segments of an object about a certain distance
-  (/wiki/index.php?title=File:Draft_Trimex.png) Trim/Extend (Trimex) (/wiki/index.php?title=Draft_Trimex): Trims or extends an object
-  (/wiki/index.php?title=File:Draft_Upgrade.png) Upgrade (/wiki/index.php?title=Draft_Upgrade): Joins objects into a higher-level object
-  (/wiki/index.php?title=File:Draft_Downgrade.png) Downgrade (/wiki/index.php?title=Draft_Downgrade): Explodes objects into lower-level objects
-  (/wiki/index.php?title=File:Draft_Scale.png) Scale (/wiki/index.php?title=Draft_Scale): Scales selected object(s) around a base point
-  (/wiki/index.php?title=File:Draft_PutOnSheet.png) Drawing (/wiki/index.php?title=Draft_Drawing): Writes selected objects to a Drawing sheet (/wiki/index.php?title=Drawing_Module)
-  (/wiki/index.php?title=File:Draft_Edit.png) Edit (/wiki/index.php?title=Draft_Edit): Edits a selected object
-  (/wiki/index.php?title=File:Draft_WireToBSpline.png) Wire to BSpline (/wiki/index.php?title=Draft_WireToBSpline): Converts a wire to a BSpline and vice-versa
-  (/wiki/index.php?title=File:Draft_AddPoint.png) Add point (/wiki/index.php?title=Draft_AddPoint): Adds a point to a wire or BSpline
-  (/wiki/index.php?title=File:Draft_DelPoint.png) Delete point (/wiki/index.php?title=Draft_DelPoint): Deletes a point from a wire or BSpline
-  (/wiki/index.php?title=File:Draft_Shape2DView.png) Shape 2D View (/wiki/index.php?title=Draft_Shape2DView): Creates a 2D object which is a flattened 2D view of another 3D object
-  (/wiki/index.php?title=File:Draft_Draft2Sketch.png) Draft to Sketch (/wiki/index.php?title=Draft_Draft2Sketch): Converts a Draft object to Sketch and vice-versa
-  (/wiki/index.php?title=File:Draft_Array.png) Array (/wiki/index.php?title=Draft_Array): Creates a polar or rectangular array from selected objects

-  (/wiki/index.php?title=File:Draft_PathArray.png) Path Array (/wiki/index.php?title=Draft_PathArray): Creates an array of objects by placing the copies along a path
-  (/wiki/index.php?title=File:Draft_Clone.png) Clone (/wiki/index.php?title=Draft_Clone): Clones the selected objects
-  (/wiki/index.php?title=File:Draft_Mirror.png) Mirror (/wiki/index.php?title=Draft_Mirror): Mirrors the selected objects

Utility tools

Additional tools available via right-click context menu, depending on the selected objects.

-  (/wiki/index.php?title=File:Draft_SelectPlane.png) Set working plane (/wiki/index.php?title=Draft_SelectPlane): Sets a working plane from a standard view or a selected face
-  (/wiki/index.php?title=File:Draft_FinishLine.png) Finish line (/wiki/index.php?title=Draft_FinishLine): Ends the drawing of the current wire or bspline, without closing it
-  (/wiki/index.php?title=File:Draft_CloseLine.png) Close line (/wiki/index.php?title=Draft_CloseLine): Ends the drawing of the current wire or bspline, and closes it
-  (/wiki/index.php?title=File:Draft_UndoLine.png) Undo line (/wiki/index.php?title=Draft_UndoLine): Undoes the last segment of a line
-  (/wiki/index.php?title=File:Draft_ToggleConstructionMode.png) Toggle construction mode (/wiki/index.php?title=Draft_ToggleConstructionMode): Toggles the Draft construction mode on/off
- (/wiki/index.php?title=File:Draft_ToggleContinueMode.png) Toggle continue mode (/wiki/index.php?title=Draft_ToggleContinueMode): Toggles the Draft continue mode on/off
-  (/wiki/index.php?title=File:Draft_ApplyStyle.png) Apply style (/wiki/index.php?title=Draft_Apply): Applies the current style and color to selected objects
-  (/wiki/index.php?title=File:Draft_ToggleDisplayMode.png) Toggle display mode (/wiki/index.php?title=Draft_ToggleDisplayMode): Switches the display mode of selected objects between "flat lines" and "wireframe"
-  (/wiki/index.php?title=File:Draft_AddToGroup.png) Add to group (/wiki/index.php?title=Draft_AddToGroup): Quickly adds selected objects to an existing group
-  (/wiki/index.php?title=File:Draft_SelectGroup.png) Select group contents (/wiki/index.php?title=Draft_SelectGroup): Selects the contents of a selected group
-  (/wiki/index.php?title=File:Draft_ToggleSnap.png) Toggle snap (/wiki/index.php?title=Draft_ToggleSnap): Toggles object snapping (/wiki/index.php?title=Draft_Snap) on/off
-  (/wiki/index.php?title=File:Draft_ToggleGrid.png) Toggle grid (/wiki/index.php?title=Draft_ToggleGrid): Toggles the grid on/off
-  (/wiki/index.php?title=File:Draft_ShowSnapBar.png) Show snap bar (/wiki/index.php?title=Draft_ShowSnapBar): Shows/hides the snapping (/wiki/index.php?title=Draft_Snap) toolbar

-  (/wiki/index.php?title=File:Draft_Heal.png) Heal (/wiki/index.php?title=Draft_Heal): Heals problematic Draft objects found in very old files
-  (/wiki/index.php?title=File:Draft_FlipDimension.png) Flip Dimension (/wiki/index.php?title=Draft_FlipDimension): Flips the orientation of the text of a dimension (/wiki/index.php?title=Draft_Dimension)
-  (/wiki/index.php?title=File:Draft_VisGroup.png) VisGroup (/wiki/index.php?title=Draft_VisGroup): Creates a VisGroup in the current document

File formats

The Draft module provides FreeCAD with importers and exporters for the following file formats:

- Autodesk .DXF (/wiki/index.php?title=Draft_DXF): Imports and exports Drawing Exchange Format (http://en.wikipedia.org/wiki/AutoCAD_DXF) files created with 2D CAD applications
- SVG (as geometry) (/wiki/index.php?title=Draft_SVG): Imports and exports Scalable Vector Graphics (http://en.wikipedia.org/wiki/Scalable_Vector_Graphics) files created with vector drawing applications
- Open Cad format .OCA (/wiki/index.php?title=Draft_OCA): Imports and exports OCA/GCAD files, a potentially new open CAD file format (http://groups.google.com/group/open_cad_format)
- Airfoil Data Format .DAT (/wiki/index.php?title=Draft_DAT): Imports DAT files describing Airfoil profiles (http://www.ae.illinois.edu/m-selig/ads/coord_database.html)
- Autodesk .DWG (/wiki/index.php?title=Draft_DXF): Import and exports DWG files via the DXF importer, when the Teigha Converter (/wiki/index.php?title=Extra_python_modules) utility is installed.
- FreeCAD and DWG Import (/wiki/index.php?title=FreeCAD_and_DWG_Import): Import and exports DWG files
- FreeCAD and DXF Import (/wiki/index.php?title=FreeCAD_and_DXF_Import): Import and exports DXF files

Additional features

- Snapping (/wiki/index.php?title=Draft_Snap): Allows to place new points on special places on existing objects
- Constraining (/wiki/index.php?title=Draft_Constrain): Allows to place new points horizontally or vertically in relation to previous points
- Working with manual coordinates (/wiki/index.php?title=Draft_Coordinates): Allows to enter manual coordinates instead of clicking on screen
- Working plane (/wiki/index.php?title=Draft_SelectPlane): Allows you to define a plane in the 3D space, where next operations will take place

Preference settings

- The Draft module has its preferences (/wiki/index.php?title=Draft_Preferences) screen

Scripting

The Draft module features a complete Draft API (<http://www.freecadweb.org/api/Draft.html>) so you can use its functions in scripts and macros

Tutorials

- [Draft tutorial \(/wiki/index.php?title=Draft_tutorial\)](/wiki/index.php?title=Draft_tutorial)
- [Draft tutorial Outdated \(/wiki/index.php?title=Draft_tutorial_Outdated\)](/wiki/index.php?title=Draft_tutorial_Outdated)
- [Draft ShapeString tutorial \(/wiki/index.php?title=Draft_ShapeString_tutorial\)](/wiki/index.php?title=Draft_ShapeString_tutorial)

< [previous: Image Module \(/wiki/index.php?title=Image_Module\)](/wiki/index.php?title=Image_Module)

[Index next: Arch Module > \(/wiki/index.php?title=Arch_Module\)](/wiki/index.php?title=Arch_Module)
[\(/wiki/index.php?title=Online_Help_Toc\)](/wiki/index.php?title=Online_Help_Toc)

Scripting and Macros

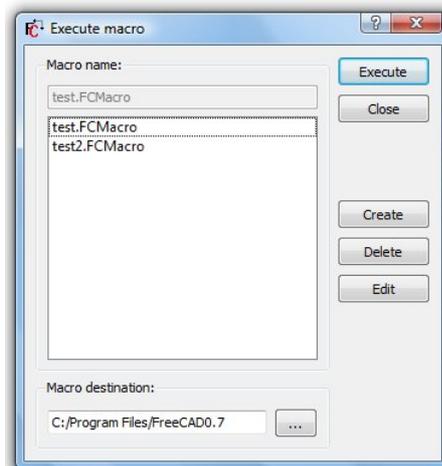
Macros

Macros are a convenient way to create complex actions in FreeCAD. You simply record actions as you do them, then save that under a name, and replay them whenever you want. Since macros are in reality a list of python commands, you can also edit them, and create very complex scripts.

How it works

If you enable console output (Menu Edit -> Preferences -> General -> Macros -> Show scripts commands in python console), you will see that in FreeCAD, every action you do, such as pressing a button, outputs a python command. Those commands are what can be recorded in a macro. The main tool for making macros is the macros toolbar:  (/wiki/index.php?title=File:Macros_toolbar.jpg). On it you have 4 buttons: Record, stop recording, edit and play the current macro.

It is very simple to use: Press the record button, you will be asked to give a name to your macro, then perform some actions. When you are done, click the stop recording button, and your actions will be saved. You can now access the macro dialog with the edit button:



(</wiki/index.php?title=File:Macros.jpg>)

[title=File:Macros.jpg](/wiki/index.php?title=File:Macros.jpg))

There you can manage your macros, delete, edit or create new ones from scratch. If you edit a macro, it will be opened in an editor window where you can make changes to its code.

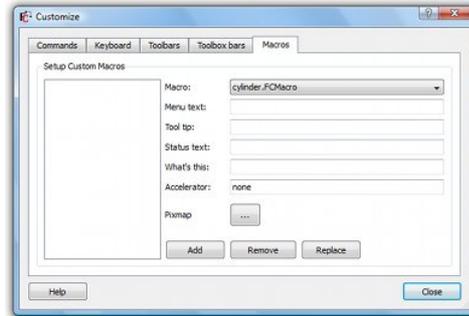
Example

Press the record button, give a name, let's say "cylinder 10x10", then, in the Part Workbench (/wiki/index.php?title=Part_Workbench), create a cylinder with radius = 10 and height = 10. Then, press the "stop recording" button. In the edit macros dialog, you can see the python code that has been recorded, and, if you want, make alterations to it. To execute your macro,

simply press the execute button on the toolbar while your macro is in the editor. Your macro is always saved to disk, so any change you make, or any new macro you create, will always be available next time you start FreeCAD.

Customizing

Of course it is not practical to load a macro in the editor in order to use it. FreeCAD provides much better ways to use your macro, such as assigning a keyboard shortcut to it or putting an entry in the menu. Once your macro is created, all this can be done via the Tools -> Customize menu:



(/wiki/index.php?

title=File:Macros_config.jpg)

Customize ToolsBar (/wiki/index.php?title=Customize_ToolsBar) This way you can make your macro become a real tool, just like any standard FreeCAD tool. This, added to the power of python scripting within FreeCAD, makes it possible to easily add your own tools to the interface. Read on to the Scripting (/wiki/index.php?title=Scripting) page if you want to know more about python scripting...

Creating macros without recording

How to install macros (/wiki/index.php?title=How_to_install_macros) You can also directly copy/paste python code into a macro, without recording GUI action. Simply create a new macro, edit it, and paste your code. You can then save your macro the same way as you save a FreeCAD document. Next time you start FreeCAD, the macro will appear under the "Installed Macros" item of the Macro menu.

Macros repository

Visit the Macros recipes (/wiki/index.php?title=Macros_recipes) page to pick some useful macros to add to your FreeCAD installation.

Links

Installing more workbenches (/wiki/index.php?title=Installing_more_workbenches)

< previous: Standard Menu (/wiki/index.php?title=Standard_Menu)

next: Introduction to Python > (/wiki/index.php?title=Introduction_to_Python)

Index (/wiki/index.php?title=Online_Help_Toc)

Introduction to Python

<translate> This is a short tutorial made for who is totally new to Python. Python (http://en.wikipedia.org/wiki/Python_%28programming_language%29) is an open-source, multiplatform programming language (http://en.wikipedia.org/wiki/Programming_language). Python has several features that make it very different than other common programming languages, and very accessible to new users like yourself:

- It has been designed specially to be easy to read by human beings, and so it is very easy to learn and understand.

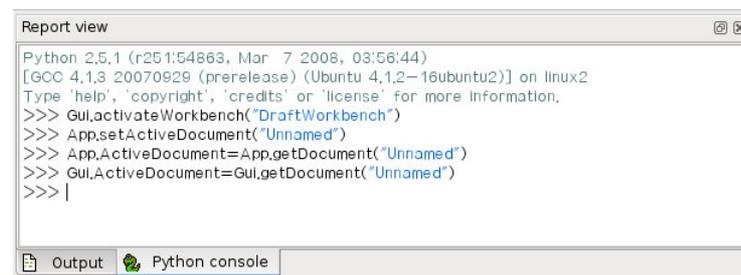
- It is interpreted, that is, unlike compiled languages like C, your program doesn't need to be compiled before it is executed. The code you write can be immediately executed, line by line if you want so. This makes it extremely easy to learn and to find errors in your code, because you go slowly, step-by-step.
- It can be embedded in other programs to be used as scripting language. FreeCAD has an embedded Python interpreter, so you can write Python code in FreeCAD, that will manipulate parts of FreeCAD, for example to create geometry. This is extremely powerful, because instead of just clicking a button labeled "create sphere", that a programmer has placed there for you, you have the freedom to create easily your own tool to create exactly the geometry you want.
- It is extensible, you can easily plug new modules in your Python installation and extend its functionality. For example, you have modules that allow Python to read and write jpg images, to communicate with twitter, to schedule tasks to be performed by your operating system, etc.

So, hands on! Be aware that what will come next is a very simple introduction, by no means a complete tutorial. But my hope is that after that you'll get enough basics to explore deeper into the FreeCAD mechanisms.

The interpreter

Usually, when writing computer programs, you simply open a text editor or your special programming environment which is in most case a text editor with several tools around it, write your program, then compile it and execute it. Most of the time you made errors while writing, so your program won't work, and you will get an error message telling you what went wrong. Then you go back to your text editor, correct the mistakes, run again, and so on until your program works fine.

That whole process, in Python, can be done transparently inside the Python interpreter. The interpreter is a Python window with a command prompt, where you can simply type Python code. If you install Python on your computer (download it from the Python website (<http://www.python.org>) if you are on Windows or Mac, install it from your package repository if you are on GNU/Linux), you will have a Python interpreter in your start menu. But FreeCAD also has a Python interpreter in its bottom part:



```

Report view
Python 2.5.1 (r251:54863, Mar  7 2008, 03:56:44)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type 'help', 'copyright', 'credits' or 'license' for more information.
>>> Gui.activateWorkbench("DraftWorkbench")
>>> App.setActiveDocument("Unnamed")
>>> App.ActiveDocument=App.getDocument("Unnamed")
>>> Gui.ActiveDocument=Gui.getDocument("Unnamed")
>>> |
  
```

(/wiki/index.php?title=File:Screenshot_pythoninterpreter.jpg)

(If you don't have it, click on View ? Views ? Python console.)

The interpreter shows the Python version, then a >>> symbol, which is the command prompt, that is, where you enter Python code. Writing code in the interpreter is simple: one line is one instruction. When you press Enter, your line of code will be executed (after being instantly and invisibly compiled). For example, try writing this: < /translate>

```
print "hello"
```

<translate> `print` is a special Python keyword that means, obviously, to print something on the screen. When you press Enter, the operation is executed, and the message "hello" is printed. If you make an error, for example let's write:< /translate>

```
print hello
```

<translate> Python will tell us that it doesn't know what hello is. The " characters specify that the content is a string, which is simply, in programming jargon, a piece of text. Without the ", the `print` command believed hello was not a piece of text but a special Python keyword. The important thing is, you immediately get notified that you made an error. By pressing the up arrow (or, in the FreeCAD interpreter, CTRL+up arrow), you can go back to the last command you wrote and correct it.

The Python interpreter also has a built-in help system. Try typing:< /translate>

```
help
```

<translate> or, for example, let's say we don't understand what went wrong with our `print hello` command above, we want specific information about the "print" command:< /translate>

```
help("print")
```

<translate> You'll get a long and complete description of everything the `print` command can do.

Now we dominate totally our interpreter, we can begin with serious stuff.

Variables

Of course, printing "hello" is not very interesting. More interesting is printing stuff you don't know before, or let Python find for you. That's where the concept of variable comes in. A variable is simply a value that you store under a name. For example, type this:< /translate>

```
a = "hello"
print a
```

<translate> I guess you understood what happened, we "saved" the string "hello" under the name `a`. Now, `a` is not an unknown name anymore! We can use it anywhere, for example in the `print` command. We can use any name we want, just respecting simple rules, like not using spaces or punctuation. For example, we could very well write:< /translate>

```
hello = "my own version of hello"
print hello
```

<translate> See? now `hello` is not an undefined word anymore. What if, by terrible bad luck, we choosed a name that already exists in Python? Let's say we want to store our string under the name "print":< /translate>

```
print = "hello"
```

<translate> Python is very intelligent and will tell us that this is not possible. It has some "reserved" keywords that cannot be modified. But our own variables can be modified anytime, that's exactly why they are called variables, the contents can vary. For example:< /translate>

```
myVariable = "hello"
print myVariable
myVariable = "good bye"
print myVariable
```

<translate> We changed the value of `myVariable`. We can also copy variables:< /translate>

```
var1 = "hello"  
var2 = var1  
print var2
```

<translate> Note that it is interesting to give good names to your variables, because when you'll write long programs, after a while you won't remember what your variable named "a" is for. But if you named it for example myWelcomeMessage, you'll remember easily what it is used for when you'll see it.

Numbers

Of course you must know that programming is useful to treat all kind of data, and especially numbers, not only text strings. One thing is important, Python must know what kind of data it is dealing with. We saw in our print hello example, that the print command recognized our "hello" string. That is because by using the ", we told specifically the print command that what it would come next is a text string.

We can always check what data type is the contents of a variable with the special Python keyword type: < /translate>

```
myVar = "hello"  
type(myVar)
```

<translate> It will tell us the contents of myVar is 'str', or string in Python jargon. We have also other basic types of data, such as integer and float numbers: < /translate>

```
firstNumber = 10  
secondNumber = 20  
print firstNumber + secondNumber  
type(firstNumber)
```

<translate> This is already much more interesting, isn't it? Now we already have a powerful calculator! Look well at how it worked, Python knows that 10 and 20 are integer numbers. So they are stored as "int", and Python can do with them everything it can do with integers. Look at the results of this:< /translate>

```
firstNumber = "10"  
secondNumber = "20"  
print firstNumber + secondNumber
```

<translate> See? We forced Python to consider that our two variables are not numbers but mere pieces of text. Python can add two pieces of text together, but it won't try to find out any sum. But we were talking about integer numbers. There are also float numbers. The difference is that integer numbers don't have decimal part, while foat numbers can have a decimal part:< /translate>

```
var1 = 13  
var2 = 15.65  
print "var1 is of type ", type(var1)  
print "var2 is of type ", type(var2)
```

<translate> Int and Floats can be mixed together without problem:< /translate>

```
total = var1 + var2  
print total  
print type(total)
```

<translate> Of course the total has decimals, right? Then Python automatically decided that the result is a float. In several cases such as this one, Python automatically decides what type to give to something. In other cases it doesn't. For example: < /translate>

```
varA = "hello 123"  
varB = 456  
print varA + varB
```

<translate> This will give us an error, varA is a string and varB is an int, and Python doesn't know what to do. But we can force Python to convert between types: </translate>

```
varA = "hello"
varB = 123
print varA + str(varB)
```

<translate> Now both are strings, the operation works! Note that we "stringified" varB at the time of printing, but we didn't change varB itself. If we wanted to turn varB permanently into a string, we would need to do this:</translate>

```
varB = str(varB)
```

<translate> We can also use int() and float() to convert to int and float if we want:</translate>

```
varA = "123"
print int(varA)
print float(varA)
```

<translate> **Note on Python commands**

You must have noticed that in this section we used the print command in several ways. We printed variables, sums, several things separated by commas, and even the result of other Python command such as type(). Maybe you also saw that doing those two commands:</translate>

```
type(varA)
print type(varA)
```

<translate> have exactly the same result. That is because we are in the interpreter, and everything is automatically printed on screen. When we'll write more complex programs that run outside the interpreter, they won't print automatically everything on screen, so we'll need to use the print command. But from now on, let's stop using it here, it'll go faster. So we can simply write: </translate>

```
myVar = "hello friends"
myVar
```

<translate> You must also have seen that most of the Python commands (or keywords) we already know have parenthesis used to tell them on what contents the command must work: type(), int(), str(), etc. Only exception is the print command, which in fact is not an exception, it also works normally like this: print("hello"), but, since it is used often, the Python programmers made a simplified version.

Lists

Another interesting data type is lists. A list is simply a list of other data. The same way as we define a text string by using " ", we define lists by using []: </translate>

```
myList = [1,2,3]
type(myList)
myOtherList = ["Bart", "Frank", "Bob"]
myMixedList = ["hello", 345, 34.567]
```

<translate> You see that it can contain any type of data. Lists are very useful because you can group variables together. You can then do all kind of things within that groups, for example counting them:</translate>

```
len(myOtherList)
```

<translate> or retrieving one item of a list:</translate>

```
myName = myOtherList[0]
myFriendsName = myOtherList[1]
```

<translate> You see that while the len() command returns the total number of items in a list, their "position" in the list begins with 0. The first item in a list is always at position 0, so in our myOtherList, "Bob" will be at position 2. We can do much more stuff with lists such as you can read here (http://www.diveintopython.net/native_data_types/lists.html), such as sorting contents, removing or adding elements.

A funny and interesting thing for you: a text string is very similar to a list of characters! Try doing this:< /translate>

```
myvar = "hello"
len(myvar)
myvar[2]
```

<translate> Usually all you can do with lists can also be done with strings. In fact both lists and strings are sequences.

Outside strings, ints, floats and lists, there are more built-in data types, such as dictionaries (http://www.diveintopython.net/native_data_types/index.html#d0e5174), or you can even create your own data types with classes (<http://www.freenetpages.co.uk/hp/alan.gauld/tutclass.htm>).

Indentation

One big cool use of lists is also browsing through them and do something with each item. For example look at this: < /translate>

```
alldaltons = ["Joe", "William", "Jack", "Averell"]
for dalton in alldaltons:
    print dalton + " Dalton"
```

<translate> We iterated (programming jargon again!) through our list with the "for ... in ..." command and did something with each of the items. Note the special syntax: the for command terminates with : which indicates that what will come after will be a block of one or more commands. Immediately after you enter the command line ending with :, the command prompt will change to ... which means Python knows that a :-ended line has happened and that what will come next will be part of it.

How will Python know how many of the next lines will be to be executed inside the for...in operation? For that, Python uses indentation. That is, your next lines won't begin immediately. You will begin them with a blank space, or several blank spaces, or a tab, or several tabs. Other programming languages use other methods, like putting everything inside parenthesis, etc. As long as you write your next lines with the **same** indentation, they will be considered part of the for-in block. If you begin one line with 2 spaces and the next one with 4, there will be an error. When you finished, just write another line without indentation, or simply press Enter to come back from the for-in block

Indentation is cool because if you make big ones (for example use tabs instead of spaces because it's larger), when you write a big program you'll have a clear view of what is executed inside what. We'll see that many other commands than for-in can have indented blocks of code too.

For-in commands can be used for many things that must be done more than once. It can for example be combined with the range() command:< /translate>

```
serie = range(1,11)
total = 0
print "sum"
for number in serie:
    print number
    total = total + number
print "---"
print total
```

<translate> For use float in for loop range.< /translate>

```

decimales = 1000          # for 3 decimales
#decimales = 10000       # for 4 decimales ...
for i in range(int(0 * decimales),int(180 * decimales),int(0.5 * decimales)):
    print float(i) / decimales

```

<translate> Or more complex things like this:< /translate>

```

alldaltons = ["Joe", "William", "Jack", "Averell"]
for n in range(4):
    print alldaltons[n], " is Dalton number ", n

```

<translate> You see that the range() command also has that strange particularity that it begins with 0 (if you don't specify the starting number) and that its last number will be one less than the ending number you specify. That is, of course, so it works well with other Python commands. For example:< /translate>

```

alldaltons = ["Joe", "William", "Jack", "Averell"]
total = len(alldaltons)
for n in range(total):
    print alldaltons[n]

```

<translate> Another interesting use of indented blocks is with the if command. It executes a code block only if a certain condition is met, for example:< /translate>

```

alldaltons = ["Joe", "William", "Jack", "Averell"]
if "Joe" in alldaltons:
    print "We found that Dalton!!!"

```

<translate> Of course this will always print the first sentence, but try replacing the second line by:< /translate>

```

if "Lucky" in alldaltons:

```

<translate> Then nothing is printed. We can also specify an else: statement:< /translate>

```

alldaltons = ["Joe", "William", "Jack", "Averell"]
if "Lucky" in alldaltons:
    print "We found that Dalton!!!"
else:
    print "Such Dalton doesn't exist!"

```

<translate>

Functions

The standard Python commands (http://docs.python.org/reference/lexical_analysis.html#identifiers) are not many. In current version of Python there are about 30, and we already know several of them. But imagine if we could invent our own commands? Well, we can, and it's extremely easy. In fact, most the additional modules that you can plug into your Python installation do just that, they add commands that you can use. A custom command in Python is called a function and is made like this:< /translate>

```

def printsqm(myValue):
    print str(myValue)+" square meters"

printsqm(45)

```

<translate> Extremely simple: the def() command defines a new function. You give it a name, and inside the parenthesis you define arguments that we'll use in our function. Arguments are data that will be passed to the function. For example, look at the len() command. If you just write len() alone, Python will tell you it needs an argument. That is, you want len() of something, right? Then, for example, you'll write len(myList) and you'll get the length of myList. Well, myList is an argument that you pass to the len() function. The len() function is defined in such a way that it knows what to do with what is passed to it. Same as we did here.

The "myValue" name can be anything, and it will be used only inside the function. It is just a name you give to the argument so you can do something with it, but it also serves so the function knows how many arguments to expect. For example, if you do this:< /translate>

```
printsum(45,34)
```

<translate> There will be an error. Our function was programmed to receive just one argument, but it received two, 45 and 34. We could instead do something like this:< /translate>

```
def sum(val1,val2):
    total = val1 + val2
    return total

sum(45,34)
myTotal = sum(45,34)
```

<translate> We made a function that receives two arguments, sums them, and returns that value. Returning something is very useful, because we can do something with the result, such as store it in the myTotal variable. Of course, since we are in the interpreter and everything is printed, doing:< /translate>

```
sum(45,34)
```

<translate> will print the result on the screen, but outside the interpreter, since there is no more print command inside the function, nothing would appear on the screen. You would need to do: < /translate>

```
print sum(45,34)
```

<translate> to have something printed. Read more about functions here (http://www.diveintopython.net/getting_to_know_python/declaring_functions.html)

Modules

Now that we have a good idea of how Python works, we'll need one last thing: How to work with files and modules.

Until now, we wrote Python instructions line by line in the interpreter, right? What if we could write several lines together, and have them executed all at once? It would certainly be handier for doing more complex things. And we could save our work too. Well, that too, is extremely easy. Simply open a text editor (such as the windows notepad), and write all your Python lines, the same way as you write them in the interpreter, with indentations, etc. Then, save that file somewhere, preferably with a .py extension. That's it, you have a complete Python program. Of course, there are much better editors than notepad, but it is just to show you that a Python program is nothing else than a text file.

To make Python execute that program, there are hundreds of ways. In windows, simply right-click your file, open it with Python, and execute it. But you can also execute it from the Python interpreter itself. For this, the interpreter must know where your .py program is. In FreeCAD, the easiest way is to place your program in a place that FreeCAD's Python interpreter knows by default, such as FreeCAD's bin folder, or any of the Mod folders. Suppose we write a file like this:< /translate>

```
def sum(a,b):
    return a + b

print "test.py succesfully loaded"
```

<translate>

and we save it as test.py in our FreeCAD/bin directory. Now, let's start FreeCAD, and in the interpreter window, write:< /translate>

```
import test
```

<translate> without the .py extension. This will simply execute the contents of the file, line by line, just as if we had written it in the interpreter. The sum function will be created, and the message will be printed. There is one big difference: the import command is made not only to execute programs written in files, like ours, but also to load the functions inside, so they become available in the interpreter. Files containing functions, like ours, are called modules.

Normally when we write a sum() function in the interpreter, we execute it simply like that: < /translate>

```
sum(14,45)
```

<translate> Like we did earlier. When we import a module containing our sum() function, the syntax is a bit different. We do:< /translate>

```
test.sum(14,45)
```

<translate> That is, the module is imported as a "container", and all its functions are inside. This is extremely useful, because we can import a lot of modules, and keep everything well organized. So, basically, everywhere you see something.somethingElse, with a dot in between, that means somethingElse is inside something.

We can also throw out the test part, and import our sum() function directly into the main interpreter space, like this:< /translate>

```
from test import *
sum(12,54)
```

<translate> Basically all modules behave like that. You import a module, then you can use its functions like that: module.function(argument). Almost all modules do that: they define functions, new data types and classes that you can use in the interpreter or in your own Python modules, because nothing prevents you to import modules inside your module!

One last extremely useful thing. How do we know what modules we have, what functions are inside and how to use them (that is, what kind of arguments they need)? We saw already that Python has a help() function. Doing:< /translate>

```
help()
modules
```

<translate> Will give us a list of all available modules. We can now type q to get out of the interactive help, and import any of them. We can even browse their content with the dir() command< /translate>

```
import math
dir(math)
```

<translate> We'll see all the functions contained in the math module, as well as strange stuff named __doc__, __file__, __name__. The __doc__ is extremely useful, it is a documentation text. Every function of (well-made) modules has a __doc__ that explains how to use it. For example, we see that there is a sin function in side the math module. Want to know how to use it? < /translate>

```
print math.sin.__doc__
```

<translate> And finally one last little goodie: When we work on programming a new module, we often want to test it.

Then it's best to replace the file extension with py and it is a normal Python module myModule.fcmacro => myModule.py. < /translate>

```
import myModule
myModule.myTestFunction()
```

<translate> But what if we see that myTestFunction() doesn't work correctly? We go back to our editor and modify it. Then, instead of closing and reopening the python interpreter, we can simply update the module like this:< /translate>

```
reload(myModule)
```

<translate> This is because Python doesn't know about the extension fcmacro.

However, there are two ways you can go: 1. Inside the one macro use Python's exec or execfile functions.< /translate>

```
f = open("myModule", "r")
d = f.read()
exec d
```

<translate> or < /translate>

```
execfile "myModule"
```

<translate> For share code across macros, you can e.g. access the FreeCAD or FreeCADGui module (or any other Python module) and set any attribute to it. This should survive the execution of the macro.< /translate>

```
import FreeCAD
if hasattr(FreeCAD, "macro2_executed"):
    ...
else:
    FreeCAD.macro2_executed = True # you can assign any value because we only check for the existence of the attribute
    ... execute macro2
```

<translate>

Starting with FreeCAD

Well, I think you must know have a good idea of how Python works, and you can start exploring what FreeCAD has to offer. FreeCAD's Python functions are all well organized in different modules. Some of them are already loaded (imported) when you start FreeCAD. So, just do< /translate>

```
dir()
```

<translate> and read on to FreeCAD Scripting Basics (/wiki/index.php?title=FreeCAD_Scripting_Basics)...

Of course, we saw here only a very small part of the Python world. There are many important concepts that we didn't mention here. There are three very important Python reference documents on the net:

- the official Python tutorial with way more information than this one (<http://docs.python.org/3/tutorial/index.html>)
- the official Python reference (<http://docs.python.org/reference/>)
- the Dive into Python (<http://www.diveintopython.net>) wikibook/ book.

Be sure to bookmark them!

< previous: [Macros \(/wiki/index.php?title=Macros\)](/wiki/index.php?title=Macros) [Index](#)
 next: [Python scripting tutorial > \(/wiki/index.php?title=Python_scripting_tutorial\)](/wiki/index.php?title=Python_scripting_tutorial)
 (/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate>

Python scripting in FreeCAD

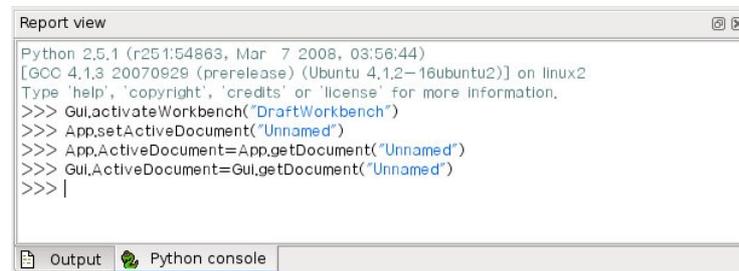
FreeCAD is built from scratch to be totally controlled by Python scripts. Almost all parts of FreeCAD, such as the interface, the scene contents, and even the representation of this content in the 3D views, are accessible from the built-in Python interpreter or from your own scripts. As a result, FreeCAD is probably one of the most deeply customizable engineering applications available today.

In its current state however, FreeCAD has very few 'native' commands to interact with your 3D objects, mainly because it is still in the early stages of development, but also because the philosophy behind it is more to provide a platform for CAD development than a specific use application. But the ease of Python scripting inside FreeCAD is a quick way to see new functionality being developed by 'power users', typically users who know a bit of Python programming. Python is one of the most popular interpreted languages, and because it is generally regarded as easy to learn, you too can soon be making your own FreeCAD 'power user' scripts.

If you are not familiar with Python, we recommend you search for tutorials on the internet and have a quick look at its structure. Python is a very easy language to learn, especially because it can be run inside an interpreter, where simple commands, right up to complete programs, can be executed on the fly without the need to compile anything. FreeCAD has a built-in Python interpreter. If you don't see the window labeled 'Python console' as shown below, you can activate it under the View -> Views -> Python console to bring-up the interpreter.

The interpreter

From the interpreter, you can access all your system-installed Python modules, as well as the built-in FreeCAD modules, and all additional FreeCAD modules you installed later. The screenshot below shows the Python interpreter:

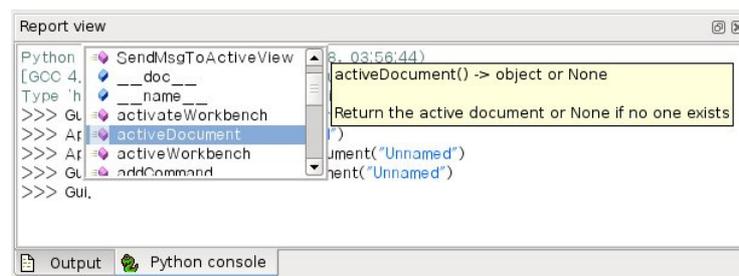


```

Python 2.5.1 (r251:54863, Mar  7 2008, 03:56:44)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type 'help', 'copyright', 'credits' or 'license' for more information.
>>> Gui.activateWorkbench("DraftWorkbench")
>>> App.setActiveDocument("Unnamed")
>>> App.ActiveDocument=App.getDocument("Unnamed")
>>> Gui.ActiveDocument=Gui.getDocument("Unnamed")
>>> |
  
```

(/wiki/index.php?title=File:Screenshot_pythoninterpreter.jpg)

From the interpreter, you can execute Python code and browse through the available classes and functions. FreeCAD provides a very handy class browser for exploration of your new FreeCAD world: When you type the name of a known class followed by a period (meaning you want to add something from that class), a class browser window opens, where you can navigate between available subclasses and methods. When you select something, an associated help text (if it exists) is displayed:



```

Python 2.5.1 (r251:54863, Mar  7 2008, 03:56:44)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type 'help', 'copyright', 'credits' or 'license' for more information.
>>> Gui.activateWorkbench("DraftWorkbench")
>>> App.setActiveDocument("Unnamed")
>>> App.ActiveDocument=App.getDocument("Unnamed")
>>> Gui.ActiveDocument=Gui.getDocument("Unnamed")
>>> |
  
```

(/wiki/index.php?title=File:Screenshot_classbrowser.jpg)

So, start here by typing **App**. or **Gui**. and see what happens. Another more generic Python way of exploring the content of modules and classes is to use the 'print dir()' command. For example, typing **print dir()** will list all modules currently loaded in FreeCAD. **print dir(App)** will show you everything inside the App module, etc.

Another useful feature of the interpreter is the possibility to go back through the command history and retrieve a line of code that you already typed earlier. To navigate through the command history, just use CTRL+UP or CTRL+DOWN.

By right-clicking in the interpreter window, you also have several other options, such as copy the entire history (useful when you want to experiment with things before making a full script of them), or insert a filename with complete path.

Python Help

In the FreeCAD Help menu, you'll find an entry labeled 'Python help', which will open a browser window containing a complete, realtime-generated documentation of all Python modules available to the FreeCAD interpreter, including Python and FreeCAD built-in modules, system-installed modules, and FreeCAD additional modules. The documentation available there depends on how much effort each module developer put into documenting his code, but usually Python modules have a reputation for being fairly well documented. Your FreeCAD window must stay open for this documentation system to work.

Built-in modules

Since FreeCAD is designed to be run without a Graphical User Interface (GUI), almost all its functionality is separated into two groups: Core functionality, named 'App', and GUI functionality, named 'Gui'. So, our two main FreeCAD built-in modules are called App and Gui. These two modules can also be accessed from scripts outside of the interpreter, by the names 'FreeCAD' and 'FreeCADGui' respectively.

- In the **App module**, you'll find everything related to the application itself, like methods for opening or closing files, and to the documents, like setting the active document or listing their contents.
- In the **Gui module**, you'll find tools for accessing and managing Gui elements, like the workbenches and their toolbars, and, more interestingly, the graphical representation of all FreeCAD content.

Listing all the content of those modules is a bit counter-productive task, since they grow quite fast with FreeCAD development. But the two browsing tools provided (the class browser and the Python help) should give you, at any moment, complete and up-to-date documentation of these modules.

The App and Gui objects

As we said, in FreeCAD, everything is separated between core and representation. This includes the 3D objects too. You can access defining properties of objects (called features in FreeCAD) via the App module, and change the way they are represented on screen via the Gui module. For example, a cube has properties that define it, (like width, length, height) that are stored in an App object, and representation properties, (like faces color, drawing mode) that are stored in a corresponding Gui object.

This way of doing things allows a very wide range of uses, like having algorithms work only on the definition part of features, without the need to care about any visual part, or even redirect the content of the document to non-graphical application, such as lists, spreadsheets, or element analysis.

For every App object in your document, there exists a corresponding Gui object. Infact the document itself has both App and a Gui objects. This, of course, is only valid when you run FreeCAD with its full interface. In the command-line version no GUI exists, so only App objects are available. Note that the Gui part of objects is re-generated every time an App object is

marked as 'to be recomputed' (for example when one of its parameters changes), so changes you might have made directly to the Gui object may be lost.

To access the App part of something, you type:< /translate>

```
myObject = App.ActiveDocument.getObject("ObjectName")
```

<translate> where "ObjectName" is the name of your object. You can also type: < /translate>

```
myObject = App.ActiveDocument.ObjectName
```

<translate> To access the Gui part of the same object, you type:< /translate>

```
myViewObject = Gui.ActiveDocument.getObject("ObjectName")
```

<translate> where "ObjectName" is the name of your object. You can also type: < /translate>

```
myViewObject = App.ActiveDocument.ObjectName.ViewObject
```

<translate> If we have no GUI (for example we are in command-line mode), the last line will return 'None'.

The Document objects

In FreeCAD all your work resides inside Documents. A document contains your geometry and can be saved to a file. Several documents can be opened at the same time. The document, like the geometry contained inside, has App and Gui objects. App object contains your actual geometry definitions, while the Gui object contains the different views of your document. You can open several windows, each one viewing your work with a different zoom factor or point of view. These views are all part of your document's Gui object.

To access the App part the currently open (active) document, you type:< /translate>

```
myDocument = App.ActiveDocument
```

<translate> To create a new document, type:< /translate>

```
myDocument = App.newDocument("Document Name")
```

<translate> To access the Gui part the currently open (active) document, you type: < /translate>

```
myGuiDocument = Gui.ActiveDocument
```

<translate> To access the current view, you type:< /translate>

```
myView = Gui.ActiveDocument.ActiveView
```

<translate>

Using additional modules

The FreeCAD and FreeCADGui modules are solely responsible for creating and managing objects in the FreeCAD document. They don't actually do anything such as creating or modifying geometry. That is because that geometry can be of several types, and so it is managed by additional modules, each responsible for managing a certain geometry type. For example, the Part Module (/wiki/index.php?title=Part_Module) uses the OpenCascade kernel, and therefore is able to create and manipulate B-rep (http://en.wikipedia.org/wiki/Boundary_representation) type geometry, which is what OpenCascade is built for. The Mesh Module (/wiki/index.php?title=Mesh_Module) is able to build and modify mesh objects. That way, FreeCAD is able to handle a wide variety of object types, that can all coexist in the same document, and new types could be added easily in the future.

Creating objects

Each module has its own way to treat its geometry, but one thing they usually all can do is create objects in the document. But the FreeCAD document is also aware of the available object types provided by the modules:< /translate>

```
FreeCAD.ActiveDocument.supportedTypes()
```

<translate> will list you all the possible objects you can create. For example, let's create a mesh (treated by the mesh module) and a part (treated by the part module):< /translate>

```
myMesh = FreeCAD.ActiveDocument.addObject("Mesh::Feature", "myMeshName")
myPart = FreeCAD.ActiveDocument.addObject("Part::Feature", "myPartName")
```

<translate> The first argument is the object type, the second the name of the object. Our two objects look almost the same: They don't contain any geometry yet, and most of their properties are the same when you inspect them with `dir(myMesh)` and `dir(myPart)`. Except for one, `myMesh` has a "Mesh" property and "Part" has a "Shape" property. That is where the Mesh and Part data are stored. For example, let's create a Part cube and store it in our `myPart` object:< /translate>

```
import Part
cube = Part.makeBox(2,2,2)
myPart.Shape = cube
```

<translate> You could try storing the cube inside the Mesh property of the `myMesh` object, it will return an error complaining of the wrong type. That is because those properties are made to store only a certain type. In the `myMesh`'s Mesh property, you can only save stuff created with the Mesh module. Note that most modules also have a shortcut to add their geometry to the document:< /translate>

```
import Part
cube = Part.makeBox(2,2,2)
Part.show(cube)
```

<translate>

Modifying objects

Modifying an object is done the same way: < /translate>

```
import Part
cube = Part.makeBox(2,2,2)
myPart.Shape = cube
```

<translate> Now let's change the shape by a bigger one:< /translate>

```
biggercube = Part.makeBox(5,5,5)
myPart.Shape = biggercube
```

<translate>

Querying objects

You can always look at the type of an object like this: < /translate>

```
myObj = FreeCAD.ActiveDocument.getObject("myObjectName")
print myObj.TypeId
```

<translate> or know if an object is derived from one of the basic ones (Part Feature, Mesh Feature, etc):< /translate>

```
print myObj.isDerivedFrom("Part::Feature")
```

<translate> Now you can really start playing with FreeCAD! To look at what you can do with the Part Module (/wiki/index.php?title=Part_Module), read the Part scripting (/wiki/index.php?title=Topological_data_scripting) page, or the Mesh Scripting (/wiki/index.php?title=Mesh_Scripting) page for working with the Mesh Module (/wiki/index.php?title=Mesh_Module). Note

that, although the Part and Mesh modules are the most complete and widely used, other modules such as the Draft Module (/wiki/index.php?title=Draft_Module) also have scripting (/wiki/index.php?title=Draft_API) APIs that can be useful to you. For a complete list of each modules and their available tools, visit the [Category:API](/wiki/index.php?title=Category:API) (</wiki/index.php?title=Category:API>) section.

< previous: [Python scripting tutorial](/wiki/index.php?title=Python_scripting_tutorial) (/wiki/index.php?title=Python_scripting_tutorial)

next: [Mesh Scripting](/wiki/index.php?title=Mesh_Scripting) > (/wiki/index.php?title=Mesh_Scripting)
[Index](/wiki/index.php?title=Online_Help_Toc) (/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate>

Introduction

First of all you have to import the Mesh module:< /translate>

```
import Mesh
```

<translate> After that you have access to the Mesh module and the Mesh class which facilitate the functions of the FreeCAD C++ Mesh-Kernel.

Creation and Loading

To create an empty mesh object just use the standard constructor:

</translate>

```
mesh = Mesh.Mesh()
```

<translate>

You can also create an object from a file

</translate>

```
mesh = Mesh.Mesh('D:/temp/Something.stl')
```

<translate>

(A list of compatible filetypes can be found under 'Meshes' here (/wiki/index.php?title=Feature_list#IO)).

Or create it out of a set of triangles described by their corner points:

</translate>

```
planarMesh = [  
# triangle 1  
[-0.5000,-0.5000,0.0000],[0.5000,0.5000,0.0000],[-0.5000,0.5000,0.0000],  
#triangle 2  
[-0.5000,-0.5000,0.0000],[0.5000,-0.5000,0.0000],[0.5000,0.5000,0.0000],  
]  
planarMeshObject = Mesh.Mesh(planarMesh)  
Mesh.show(planarMeshObject)
```

<translate>

The Mesh-Kernel takes care about creating a topological correct data structure by sorting coincident points and edges together.

Later on you will see how you can test and examine mesh data.

Modeling

To create regular geometries you can use the Python script `BuildRegularGeoms.py`.

</translate>

```
import BuildRegularGeoms
```

<translate>

This script provides methods to define simple rotation bodies like spheres, ellipsoids, cylinders, toroids and cones. And it also has a method to create a simple cube. To create a toroid, for instance, can be done as follows:

</translate>

```
t = BuildRegularGeoms.Toroid(8.0, 2.0, 50) # list with several thousands triangles
m = Mesh.Mesh(t)
```

<translate>

The first two parameters define the radiuses of the toroid and the third parameter is a sub-sampling factor for how many triangles are created. The higher this value the smoother and the lower the coarser the body is. The Mesh class provides a set of boolean functions that can be used for modeling purposes. It provides union, intersection and difference of two mesh objects.

</translate>

```
m1, m2          # are the input mesh objects
m3 = Mesh.Mesh(m1) # create a copy of m1
m3.unite(m2)      # union of m1 and m2, the result is stored in m3
m4 = Mesh.Mesh(m1)
m4.intersect(m2)  # intersection of m1 and m2
m5 = Mesh.Mesh(m1)
m5.difference(m2) # the difference of m1 and m2
m6 = Mesh.Mesh(m2)
m6.difference(m1) # the difference of m2 and m1, usually the result is different to m5
```

<translate>

Finally, a full example that computes the intersection between a sphere and a cylinder that intersects the sphere.

</translate>

```
import Mesh, BuildRegularGeoms
sphere = Mesh.Mesh( BuildRegularGeoms.Sphere(5.0, 50) )
cylinder = Mesh.Mesh( BuildRegularGeoms.Cylinder(2.0, 10.0, True, 1.0, 50) )
diff = sphere
diff = diff.difference(cylinder)
d = FreeCAD.newDocument()
d.addObject("Mesh::Feature", "Diff_Sphere_Cylinder").Mesh=diff
d.recompute()
```

<translate>

Examining and Testing

Write your own Algorithms

Exporting

You can even write the mesh to a python module:

</translate>

```
m.write("D:/Develop/Projekte/FreeCAD/FreeCAD_0.7/Mod/Mesh/SavedMesh.py")
import SavedMesh
m2 = Mesh.Mesh(SavedMesh.faces)
```

<translate>

Gui related stuff

Odds and Ends

An extensive (though hard to use) source of Mesh related scripting are the unit test scripts of the Mesh-Module. In this unit tests literally all methods are called and all properties/attributes are tweaked. So if you are bold enough, take a look at the Unit Test module (<http://freecad.svn.sourceforge.net/viewvc/freecad/trunk/src/Mod/Mesh/App/MeshTestsApp.py?view=markup>).

See also Mesh API (/wiki/index.php?title=Mesh_API)

< previous: FreeCAD Scripting Basics (/wiki/index.php?title=FreeCAD_Scripting_Basics)

next: Topological data scripting > (/wiki/index.php?title=Topological_data_scripting)

Index (/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate>

 (/wiki/index.php?title=File:Base_ExampleCommandModel.png) Tutorial

Topic

Programming

Level

Intermediate

Time to complete

Author

FreeCAD version

Example File(s)

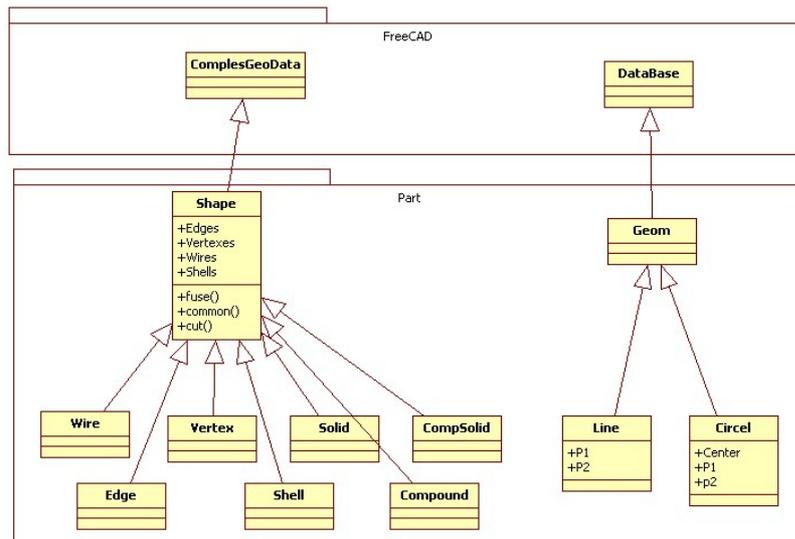
This page describes several methods for creating and modifying Part shapes (/wiki/index.php?title=Part_Module) from python. Before reading this page, if you are new to python, it is a good idea to read about python scripting (/wiki/index.php?title=Introduction_to_Python) and how python scripting works in FreeCAD (/wiki/index.php?title=FreeCAD_Scripting_Basics).

Introduction

We will here explain you how to control the Part Module (/wiki/index.php?title=Part_Module) directly from the FreeCAD python interpreter, or from any external script. The basics about Topological data scripting are described in Part Module Explaining the concepts (/wiki/index.php?title=Part_Module#Explaining_the_concepts). Be sure to browse the Scripting (</wiki/index.php?title=Scripting>) section and the FreeCAD Scripting Basics (/wiki/index.php?title=FreeCAD_Scripting_Basics) pages if you need more information about how python scripting works in FreeCAD.

Class Diagram

This is a Unified Modeling Language (UML) (http://en.wikipedia.org/wiki/Unified_Modeling_Language) overview of the most important classes of the Part module:



(/wiki/index.php?title=File:Part_Classes.jpg)

Geometry

The geometric objects are the building block of all topological objects:

- **Geom** Base class of the geometric objects
- **Line** A straight line in 3D, defined by starting point and end point
- **Circle** Circle or circle segment defined by a center point and start and end point
- And soon some more

Topology

The following topological data types are available:

- **Compound** A group of any type of topological object.
- **Compsolid** A composite solid is a set of solids connected by their faces. It expands the notions of WIRE and SHELL to solids.
- **Solid** A part of space limited by shells. It is three dimensional.
- **Shell** A set of faces connected by their edges. A shell can be open or closed.
- **Face** In 2D it is part of a plane; in 3D it is part of a surface. Its geometry is constrained (trimmed) by contours. It is two dimensional.
- **Wire** A set of edges connected by their vertices. It can be an open or closed contour depending on whether the edges are linked or not.
- **Edge** A topological element corresponding to a restrained curve. An edge is generally limited by vertices. It has one dimension.
- **Vertex** A topological element corresponding to a point. It has zero dimension.
- **Shape** A generic term covering all of the above.

Quick example : Creating simple topology



(/wiki/index.php?title=File:Wire.png)

We will now create a topology by constructing it out of simpler geometry. As a case study we use a part as seen in the picture which consists of four vertexes, two circles and two lines.

Creating Geometry

First we have to create the distinct geometric parts of this wire. And we have to take care that the vertexes of the geometric parts are at the **same** position. Otherwise later on we might not be able to connect the geometric parts to a topology!

So we create first the points:

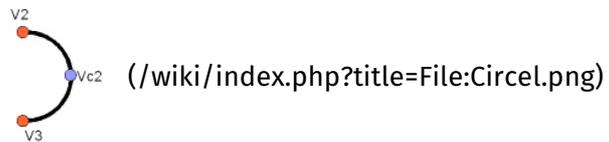
</translate>

```
from FreeCAD import Base
V1 = Base.Vector(0,10,0)
V2 = Base.Vector(30,10,0)
V3 = Base.Vector(30,-10,0)
V4 = Base.Vector(0,-10,0)
```

<translate>

Arc

To create an arc of circle we make a helper point and create the arc of circle through three points:



</translate>

```
VC1 = Base.Vector(-10,0,0)
C1 = Part.Arc(V1,VC1,V4)
# and the second one
VC2 = Base.Vector(40,0,0)
C2 = Part.Arc(V2,VC2,V3)
```

<translate>

Line

The line can be created very simple out of the points:



</translate>

```
L1 = Part.Line(V1,V2)
# and the second one
L2 = Part.Line(V4,V3)
```

<translate>

Putting all together

The last step is to put the geometric base elements together and bake a topological shape:

</translate>

```
S1 = Part.Shape([C1,C2,L1,L2])
```

<translate>

Make a prism

Now extrude the wire in a direction and make an actual 3D shape:

</translate>

```
W = Part.Wire(S1.Edges)
P = W.extrude(Base.Vector(0,0,10))
```

<translate>

Show it all

</translate>

```
Part.show(P)
```

<translate>

Creating basic shapes

You can easily create basic topological objects with the "make...()" methods from the Part Module:

</translate>

```
b = Part.makeBox(100,100,100)
Part.show(b)
```

<translate>

A couple of other make...() methods available:

- **makeBox(l,w,h):** Makes a box located in p and pointing into the direction d with the dimensions (l,w,h)
- **makeCircle(radius):** Makes a circle with a given radius
- **makeCone(radius1,radius2,height):** Makes a cone with a given radii and height
- **makeCylinder(radius,height):** Makes a cylinder with a given radius and height.
- **makeLine((x1,y1,z1),(x2,y2,z2)):** Makes a line of two points
- **makePlane(length,width):** Makes a plane with length and width
- **makePolygon(list):** Makes a polygon of a list of points
- **makeSphere(radius):** Make a sphere with a given radius
- **makeTorus(radius1,radius2):** Makes a torus with a given radii

See the Part API (/wiki/index.php?title=Part_API) page for a complete list of available methods of the Part module.

Importing the needed modules

First we need to import the Part module so we can use its contents in python. We'll also import the Base module from inside the FreeCAD module:

</translate>

```
import Part
from FreeCAD import Base
```

<translate>

Creating a Vector

Vectors (http://en.wikipedia.org/wiki/Euclidean_vector) are one of the most important pieces of information when building shapes. They contain a 3 numbers usually (but not necessarily always) the x, y and z cartesian coordinates. You create a vector like this:

</translate>

```
myVector = Base.Vector(3,2,0)
```

<translate>

We just created a vector at coordinates x=3, y=2, z=0. In the Part module, vectors are used everywhere. Part shapes also use another kind of point representation, called Vertex, which is actually nothing else than a container for a vector. You access the vector of a vertex like this:

</translate>

```
myVertex = myShape.Vertexes[0]
print myVertex.Point
> Vector (3, 2, 0)
```

<translate>

Creating an Edge

An edge is nothing but a line with two vertexes:

</translate>

```
edge = Part.makeLine((0,0,0), (10,0,0))
edge.Vertexes
> [<Vertex object at 01877430>, <Vertex object at 014888E0>]
```

<translate>

Note: You can also create an edge by passing two vectors:

</translate>

```
vec1 = Base.Vector(0,0,0)
vec2 = Base.Vector(10,0,0)
line = Part.Line(vec1,vec2)
edge = line.toShape()
```

<translate>

You can find the length and center of an edge like this:

</translate>

```
edge.Length
> 10.0
edge.CenterOfMass
> Vector (5, 0, 0)
```

<translate>

Putting the shape on screen

So far we created an edge object, but it doesn't appear anywhere on screen. This is because we just manipulated python objects here. The FreeCAD 3D scene only displays what you tell it to display. To do that, we use this simple method:

</translate>

```
Part.show(edge)
```

<translate>

An object will be created in our FreeCAD document, and our "edge" shape will be attributed to it. Use this whenever it's time to display your creation on screen.

Creating a Wire

A wire is a multi-edge line and can be created from a list of edges or even a list of wires:

</translate>

```
edge1 = Part.makeLine((0,0,0), (10,0,0))
edge2 = Part.makeLine((10,0,0), (10,10,0))
wire1 = Part.Wire([edge1,edge2])
edge3 = Part.makeLine((10,10,0), (0,10,0))
edge4 = Part.makeLine((0,10,0), (0,0,0))
wire2 = Part.Wire([edge3,edge4])
wire3 = Part.Wire([wire1,wire2])
wire3.Edges
> [<Edge object at 016695F8>, <Edge object at 0197AE08>, <Edge object at 01828B20>, <Edge object at 0190A788>]
Part.show(wire3)
```

<translate>

Part.show(wire3) will display the 4 edges that compose our wire. Other useful information can be easily retrieved:

</translate>

```
wire3.Length
> 40.0
wire3.CenterOfMass
> Vector (5, 5, 0)
wire3.isClosed()
> True
wire2.isClosed()
> False
```

<translate>

Creating a Face

Only faces created from closed wires will be valid. In this example, wire3 is a closed wire but wire2 is not a closed wire (see above)

</translate>

```
face = Part.Face(wire3)
face.Area
> 99.999999999999972
face.CenterOfMass
> Vector (5, 5, 0)
face.Length
> 40.0
face.isValid()
> True
sface = Part.Face(wire2)
face.isValid()
> False
```

<translate>

Only faces will have an area, not wires nor edges.

Creating a Circle

A circle can be created as simply as this:

</translate>

```
circle = Part.makeCircle(10)
circle.Curve
> Circle (Radius : 10, Position : (0, 0, 0), Direction : (0, 0, 1))
```

<translate>

If you want to create it at certain position and with certain direction:

</translate>

```
ccircle = Part.makeCircle(10, Base.Vector(10,0,0), Base.Vector(1,0,0))
ccircle.Curve
> Circle (Radius : 10, Position : (10, 0, 0), Direction : (1, 0, 0))
```

<translate>

ccircle will be created at distance 10 from origin on x and will be facing towards x axis. Note: makeCircle only accepts Base.Vector() for position and normal but not tuples. You can also create part of the circle by giving start angle and end angle as:

</translate>

```
from math import pi
arc1 = Part.makeCircle(10, Base.Vector(0,0,0), Base.Vector(0,0,1), 0, 180)
arc2 = Part.makeCircle(10, Base.Vector(0,0,0), Base.Vector(0,0,1), 180, 360)
```

<translate>

Both arc1 and arc2 jointly will make a circle. Angles should be provided in degrees, if you have radians simply convert them using formula: degrees = radians * 180/PI or using python's math module (after doing import math, of course):

</translate>

```
degrees = math.degrees(radians)
```

<translate>

Creating an Arc along points

Unfortunately there is no `makeArc` function but we have `Part.Arc` function to create an arc along three points. Basically it can be supposed as an arc joining start point and end point along the middle point. `Part.Arc` creates an arc object on which `.toShape()` has to be called to get the edge object, the same way as when using `Part.Line` instead of `Part.makeLine`.

</translate>

```
arc = Part.Arc(Base.Vector(0,0,0),Base.Vector(0,5,0),Base.Vector(5,5,0))
arc
> <Arc object>
arc_edge = arc.toShape()
```

<translate>

`Arc` only accepts `Base.Vector()` for points but not tuples. `arc_edge` is what we want which we can display using `Part.show(arc_edge)`. You can also obtain an arc by using a portion of a circle:

</translate>

```
from math import pi
circle = Part.Circle(Base.Vector(0,0,0),Base.Vector(0,0,1),10)
arc = Part.Arc(c,0,pi)
```

<translate>

Arcs are valid edges, like lines. So they can be used in wires too.

Creating a polygon

A polygon is simply a wire with multiple straight edges. The `makePolygon` function takes a list of points and creates a wire along those points:

</translate>

```
lshape_wire = Part.makePolygon([Base.Vector(0,5,0),Base.Vector(0,0,0),Base.Vector(5,0,0)])
```

<translate>

Creating a Bezier curve

Bézier curves are used to model smooth curves using a series of poles (points) and optional weights. The function below makes a `Part.BezierCurve` from a series of `FreeCAD.Vector` points. (Note: when "getting" and "setting" a single pole or weight indices start at 1, not 0.)

</translate>

```
def makeBCurveEdge(Points):
    geomCurve = Part.BezierCurve()
    geomCurve.setPoles(Points)
    edge = Part.Edge(geomCurve)
    return(edge)
```

<translate>

Creating a Plane

A Plane is simply a flat rectangular surface. The method used to create one is this: `makePlane(length,width,[start_pnt,dir_normal])`. By default `start_pnt = Vector(0,0,0)` and `dir_normal = Vector(0,0,1)`. Using `dir_normal = Vector(0,0,1)` will create the plane facing z axis, while `dir_normal = Vector(1,0,0)` will create the plane facing x axis:

</translate>

```
plane = Part.makePlane(2,2)
plane
>><Face object at 028AF990>
plane = Part.makePlane(2,2, Base.Vector(3,0,0), Base.Vector(0,1,0))
plane.BoundingBox
> BoundingBox (3, 0, 0, 5, 0, 2)
```

<translate>

`BoundingBox` is a cuboid enclosing the plane with a diagonal starting at (3,0,0) and ending at (5,0,2). Here the `BoundingBox` thickness in y axis is zero, since our shape is totally flat.

Note: `makePlane` only accepts `Base.Vector()` for `start_pnt` and `dir_normal`

but not tuples

Creating an ellipse

To create an ellipse there are several ways:

</translate>

```
Part.Ellipse()
```

<translate>

Creates an ellipse with major radius 2 and minor radius 1 with the center in (0,0,0)

</translate>

```
Part.Ellipse(Ellipse)
```

<translate>

Create a copy of the given ellipse

</translate>

```
Part.Ellipse(S1,S2,Center)
```

<translate>

Creates an ellipse centered on the point Center, where the plane of the ellipse is defined by Center, S1 and S2, its major axis is defined by Center and S1, its major radius is the distance between Center and S1, and its minor radius is the distance between S2 and the major axis.

</translate>

```
Part.Ellipse(Center,MajorRadius,MinorRadius)
```

<translate>

Creates an ellipse with major and minor radii MajorRadius and MinorRadius, and located in the plane defined by Center and the normal (0,0,1)

</translate>

```
eli = Part.Ellipse(Base.Vector(10,0,0),Base.Vector(0,5,0),Base.Vector(0,0,0))
Part.show(eli.toShape())
```

<translate>

In the above code we have passed S1, S2 and center. Similarly to Arc, Ellipse also creates an ellipse object but not edge, so we need to convert it into edge using toShape() to display.

Note: Arc only accepts Base.Vector() for points but not tuples

</translate>

```
eli = Part.Ellipse(Base.Vector(0,0,0),10,5)
Part.show(eli.toShape())
```

<translate>

for the above Ellipse constructor we have passed center, MajorRadius and MinorRadius

Creating a Torus

Using the method **makeTorus(radius1,radius2,[pnt,dir,angle1,angle2,angle])**. By default pnt=Vector(0,0,0),dir=Vector(0,0,1),angle1=0,angle2=360 and angle=360. Consider a torus as small circle sweeping along a big circle. Radius1 is the radius of big circle, radius2 is the radius of small circle, pnt is the center of torus and dir is the normal direction. angle1 and angle2 are angles in radians for the small circle, the last parameter angle is to make a section of the torus:

</translate>

```
torus = Part.makeTorus(10, 2)
```

<translate>

The above code will create a torus with diameter 20(radius 10) and thickness 4 (small circle radius 2)

</translate>

```
tor=Part.makeTorus(10,5,Base.Vector(0,0,0),Base.Vector(0,0,1),0,180)
```

<translate>

The above code will create a slice of the torus

</translate>

```
tor=Part.makeTorus(10,5,Base.Vector(0,0,0),Base.Vector(0,0,1),0,360,180)
```

<translate>

The above code will create a semi torus, only the last parameter is changed i.e the angle and remaining angles are defaults. Giving the angle 180 will create the torus from 0 to 180, that is, a half torus.

Creating a box or cuboid

Using **makeBox(length,width,height,[pnt,dir])**. By default pnt=Vector(0,0,0) and dir=Vector(0,0,1)

</translate>

```
box = Part.makeBox(10,10,10)
len(box.Vertices)
> 8
```

<translate>

Creating a Sphere

Using **makeSphere(radius,[pnt, dir, angle1,angle2,angle3])**. By default pnt=Vector(0,0,0), dir=Vector(0,0,1), angle1=-90, angle2=90 and angle3=360. angle1 and angle2 are the vertical minimum and maximum of the sphere, angle3 is the sphere diameter itself.

</translate>

```
sphere = Part.makeSphere(10)
hemisphere = Part.makeSphere(10,Base.Vector(0,0,0),Base.Vector(0,0,1),-90,90,180)
```

<translate>

Creating a Cylinder

Using **makeCylinder(radius,height,[pnt,dir,angle])**. By default pnt=Vector(0,0,0),dir=Vector(0,0,1) and angle=360

</translate>

```
cylinder = Part.makeCylinder(5,20)
partCylinder = Part.makeCylinder(5,20,Base.Vector(20,0,0),Base.Vector(0,0,1),180)
```

<translate>

Creating a Cone

Using **makeCone(radius1,radius2,height,[pnt,dir,angle])**. By default pnt=Vector(0,0,0), dir=Vector(0,0,1) and angle=360

</translate>

```
cone = Part.makeCone(10,0,20)
semicone = Part.makeCone(10,0,20,Base.Vector(20,0,0),Base.Vector(0,0,1),180)
```

<translate>

Modifying shapes

There are several ways to modify shapes. Some are simple transformation operations such as moving or rotating shapes, other are more complex, such as unioning and subtracting one shape from another. Be aware that

Transform operations

Translating a shape

Translating is the act of moving a shape from one place to another. Any shape (edge, face, cube, etc...) can be translated the same way:

</translate>

```
myShape = Part.makeBox(2,2,2)
myShape.translate(Base.Vector(2,0,0))
```

<translate>

This will move our shape "myShape" 2 units in the x direction.

Rotating a shape

To rotate a shape, you need to specify the rotation center, the axis, and the rotation angle:

</translate>

```
myShape.rotate(Vector(0,0,0),Vector(0,0,1),180)
```

<translate>

The above code will rotate the shape 180 degrees around the Z Axis.

Generic transformations with matrixes

A matrix is a very convenient way to store transformations in the 3D world. In a single matrix, you can set translation, rotation and scaling values to be applied to an object. For example:

</translate>

```
myMat = Base.Matrix()
myMat.move(Base.Vector(2,0,0))
myMat.rotateZ(math.pi/2)
```

<translate>

Note: FreeCAD matrixes work in radians. Also, almost all matrix operations that take a vector can also take 3 numbers, so those 2 lines do the same thing:

</translate>

```
myMat.move(2,0,0)
myMat.move(Base.Vector(2,0,0))
```

<translate>

When our matrix is set, we can apply it to our shape. FreeCAD provides 2 methods to do that: transformShape() and transformGeometry(). The difference is that with the first one, you are sure that no deformations will occur (see "scaling a shape" below). So we can apply our transformation like this:

</translate>

```
myShape.transformShape(myMat)
```

<translate>

or

</translate>

```
myShape.transformGeometry(myMat)
```

<translate>

Scaling a shape

Scaling a shape is a more dangerous operation because, unlike translation or rotation, scaling non-uniformly (with different values for x, y and z) can modify the structure of the shape. For example, scaling a circle with a higher

value horizontally than vertically will transform it into an ellipse, which behaves mathematically very differently. For scaling, we can't use the `transformShape`, we must use `transformGeometry()`:

</translate>

```
myMat = Base.Matrix()
myMat.scale(2,1,1)
myShape=myShape.transformGeometry(myMat)
```

<translate>

Boolean Operations

Subtraction

Subtracting a shape from another one is called "cut" in OCC/FreeCAD jargon and is done like this:

</translate>

```
cylinder = Part.makeCylinder(3,10,Base.Vector(0,0,0),Base.Vector(1,0,0))
sphere = Part.makeSphere(5,Base.Vector(5,0,0))
diff = cylinder.cut(sphere)
```

<translate>

Intersection

The same way, the intersection between 2 shapes is called "common" and is done this way:

</translate>

```
cylinder1 = Part.makeCylinder(3,10,Base.Vector(0,0,0),Base.Vector(1,0,0))
cylinder2 = Part.makeCylinder(3,10,Base.Vector(5,0,-5),Base.Vector(0,0,1))
common = cylinder1.common(cylinder2)
```

<translate>

Union

Union is called "fuse" and works the same way:

</translate>

```
cylinder1 = Part.makeCylinder(3,10,Base.Vector(0,0,0),Base.Vector(1,0,0))
cylinder2 = Part.makeCylinder(3,10,Base.Vector(5,0,-5),Base.Vector(0,0,1))
fuse = cylinder1.fuse(cylinder2)
```

<translate>

Section

A Section is the intersection between a solid shape and a plane shape. It will return an intersection curve, a compound with edges

</translate>

```
cylinder1 = Part.makeCylinder(3,10,Base.Vector(0,0,0),Base.Vector(1,0,0))
cylinder2 = Part.makeCylinder(3,10,Base.Vector(5,0,-5),Base.Vector(0,0,1))
section = cylinder1.section(cylinder2)
section.Wires
> []
section.Edges
> [<Edge object at 0D87CFE8>, <Edge object at 019564F8>, <Edge object at 0D998458>,
<Edge object at 0D86DE18>, <Edge object at 0D9B8E80>, <Edge object at 012A3640>,
<Edge object at 0D8F4BB0>]
```

<translate>

Extrusion

Extrusion is the act of "pushing" a flat shape in a certain direction resulting in a solid body. Think of a circle becoming a tube by "pushing it out":

</translate>

```
circle = Part.makeCircle(10)
tube = circle.extrude(Base.Vector(0,0,2))
```

<translate>

If your circle is hollow, you will obtain a hollow tube. If your circle is actually a disc, with a filled face, you will obtain a solid cylinder:

</translate>

```
wire = Part.Wire(circle)
disc = Part.Face(wire)
cylinder = disc.extrude(Base.Vector(0,0,2))
```

<translate>

Exploring shapes

You can easily explore the topological data structure:

</translate>

```
import Part
b = Part.makeBox(100,100,100)
b.Wires
w = b.Wires[0]
w
w.Wires
w.Vertexes
Part.show(w)
w.Edges
e = w.Edges[0]
e.Vertexes
v = e.Vertexes[0]
v.Point
```

<translate>

By typing the lines above in the python interpreter, you will gain a good understanding of the structure of Part objects. Here, our makeBox() command created a solid shape. This solid, like all Part solids, contains faces. Faces always contain wires, which are lists of edges that border the face. Each face has at least one closed wire (it can have more if the face has a hole). In the wire, we can look at each edge separately, and inside each edge, we can see the vertexes. Straight edges have only two vertexes, obviously.

Edge analysis

In case of an edge, which is an arbitrary curve, it's most likely you want to do a discretization. In FreeCAD the edges are parametrized by their lengths. That means you can walk an edge/curve by its length:

</translate>

```
import Part
box = Part.makeBox(100,100,100)
anEdge = box.Edges[0]
print anEdge.Length
```

<translate>

Now you can access a lot of properties of the edge by using the length as a position. That means if the edge is 100mm long the start position is 0 and the end position 100.

</translate>

```
anEdge.tangentAt(0.0)      # tangent direction at the beginning
anEdge.valueAt(0.0)       # Point at the beginning
anEdge.valueAt(100.0)     # Point at the end of the edge
anEdge.derivative1At(50.0) # first derivative of the curve in the middle
anEdge.derivative2At(50.0) # second derivative of the curve in the middle
anEdge.derivative3At(50.0) # third derivative of the curve in the middle
anEdge.centerOfCurvatureAt(50) # center of the curvature for that position
anEdge.curvatureAt(50.0)  # the curvature
anEdge.normalAt(50)       # normal vector at that position (if defined)
```

<translate>

Using the selection

Here we see now how we can use the selection the user did in the viewer. First of all we create a box and shows it in the viewer

</translate>

```
import Part
Part.show(Part.makeBox(100,100,100))
Gui.SendMsgToActiveView("ViewFit")
```

<translate>

Select now some faces or edges. With this script you can iterate all selected objects and their sub elements:

</translate>

```
for o in Gui.Selection.getSelectionEx():
    print o.ObjectName
    for s in o.SubElementNames:
        print "name: ",s
    for s in o.SubObjects:
        print "object: ",s
```

<translate>

Select some edges and this script will calculate the length:

</translate>

```
length = 0.0
for o in Gui.Selection.getSelectionEx():
    for s in o.SubObjects:
        length += s.Length
print "Length of the selected edges:" ,length
```

<translate>

Complete example: The OCC bottle

A typical example found in the OpenCasCade Technology Tutorial (http://www.opencascade.com/doc/occt-6.9.0/overview/html/occt__tutorial.html#) is how to build a bottle. This is a good exercise for FreeCAD too. In fact, you can follow our example below and the OCC page simultaneously, you will understand well how OCC structures are implemented in FreeCAD. The complete script below is also included in FreeCAD installation (inside the Mod/Part folder) and can be called from the python interpreter by typing:

</translate>

```
import Part
import MakeBottle
bottle = MakeBottle.makeBottle()
Part.show(bottle)
```

<translate>

The complete script

Here is the complete MakeBottle script:

</translate>

```

import Part, FreeCAD, math
from FreeCAD import Base

def makeBottle(myWidth=50.0, myHeight=70.0, myThickness=30.0):
    aPnt1=Base.Vector(-myWidth/2.,0,0)
    aPnt2=Base.Vector(-myWidth/2.,-myThickness/4.,0)
    aPnt3=Base.Vector(0,-myThickness/2.,0)
    aPnt4=Base.Vector(myWidth/2.,-myThickness/4.,0)
    aPnt5=Base.Vector(myWidth/2.,0,0)

    aArcOfCircle = Part.Arc(aPnt2,aPnt3,aPnt4)
    aSegment1=Part.Line(aPnt1,aPnt2)
    aSegment2=Part.Line(aPnt4,aPnt5)
    aEdge1=aSegment1.toShape()
    aEdge2=aArcOfCircle.toShape()
    aEdge3=aSegment2.toShape()
    aWire=Part.Wire([aEdge1,aEdge2,aEdge3])

    aTrsf=Base.Matrix()
    aTrsf.rotateZ(math.pi) # rotate around the z-axis

    aMirroredWire=aWire.transformGeometry(aTrsf)
    myWireProfile=Part.Wire([aWire,aMirroredWire])
    myFaceProfile=Part.Face(myWireProfile)
    aPrismVec=Base.Vector(0,0,myHeight)
    myBody=myFaceProfile.extrude(aPrismVec)
    myBody=myBody.makeFillet(myThickness/12.0,myBody.Edges)
    neckLocation=Base.Vector(0,0,myHeight)
    neckNormal=Base.Vector(0,0,1)
    myNeckRadius = myThickness / 4.
    myNeckHeight = myHeight / 10
    myNeck = Part.makeCylinder(myNeckRadius,myNeckHeight,neckLocation,neckNormal)
    myBody = myBody.fuse(myNeck)

    faceToRemove = 0
    zMax = -1.0

    for xp in myBody.Faces:
        try:
            surf = xp.Surface
            if type(surf) == Part.Plane:
                z = surf.Position.z
                if z > zMax:
                    zMax = z
                    faceToRemove = xp
        except:
            continue

    myBody = myBody.makeThickness([faceToRemove],-myThickness/50 , 1.e-3)

    return myBody

```

<translate>

Detailed explanation

</translate>

```

import Part, FreeCAD, math
from FreeCAD import Base

```

<translate>

We will need,of course, the Part module, but also the FreeCAD.Base module, which contains basic FreeCAD structures like vectors and matrixes.

</translate>

```

def makeBottle(myWidth=50.0, myHeight=70.0, myThickness=30.0):
    aPnt1=Base.Vector(-myWidth/2.,0,0)
    aPnt2=Base.Vector(-myWidth/2.,-myThickness/4.,0)
    aPnt3=Base.Vector(0,-myThickness/2.,0)
    aPnt4=Base.Vector(myWidth/2.,-myThickness/4.,0)
    aPnt5=Base.Vector(myWidth/2.,0,0)

```

<translate>

Here we define our makeBottle function. This function can be called without arguments, like we did above, in which case default values for width, height, and thickness will be used. Then, we define a couple of points that will be used for building our base profile.

</translate>

```
aArcOfCircle = Part.Arc(aPnt2,aPnt3,aPnt4)
aSegment1=Part.Line(aPnt1,aPnt2)
aSegment2=Part.Line(aPnt4,aPnt5)
```

<translate>

Here we actually define the geometry: an arc, made of 3 points, and two line segments, made of 2 points.

</translate>

```
aEdge1=aSegment1.toShape()
aEdge2=aArcOfCircle.toShape()
aEdge3=aSegment2.toShape()
aWire=Part.Wire([aEdge1,aEdge2,aEdge3])
```

<translate>

Remember the difference between geometry and shapes? Here we build shapes out of our construction geometry. 3 edges (edges can be straight or curved), then a wire made of those three edges.

</translate>

```
aTrsf=Base.Matrix()
aTrsf.rotateZ(math.pi) # rotate around the z-axis
aMirroredWire=aWire.transformGeometry(aTrsf)
myWireProfile=Part.Wire([aWire,aMirroredWire])
```

<translate>

Until now we built only a half profile. Easier than building the whole profile the same way, we can just mirror what we did, and glue both halves together. So we first create a matrix. A matrix is a very common way to apply transformations to objects in the 3D world, since it can contain in one structure all basic transformations that 3D objects can suffer (move, rotate and scale). Here, after we create the matrix, we mirror it, and we create a copy of our wire with that transformation matrix applied to it. We now have two wires, and we can make a third wire out of them, since wires are actually lists of edges.

</translate>

```
myFaceProfile=Part.Face(myWireProfile)
aPrismVec=Base.Vector(0,0,myHeight)
myBody=myFaceProfile.extrude(aPrismVec)
myBody=myBody.makeFillet(myThickness/12.0,myBody.Edges)
```

<translate>

Now that we have a closed wire, it can be turned into a face. Once we have a face, we can extrude it. Doing so, we actually made a solid. Then we apply a nice little fillet to our object because we care about good design, don't we?

</translate>

```
neckLocation=Base.Vector(0,0,myHeight)
neckNormal=Base.Vector(0,0,1)
myNeckRadius = myThickness / 4.
myNeckHeight = myHeight / 10
myNeck = Part.makeCylinder(myNeckRadius,myNeckHeight,neckLocation,neckNormal)
```

<translate>

Then, the body of our bottle is made, we still need to create a neck. So we make a new solid, with a cylinder.

</translate>

```
myBody = myBody.fuse(myNeck)
```

<translate>

The fuse operation, which in other apps is sometimes called union, is very powerful. It will take care of gluing what needs to be glued and remove parts that need to be removed.

</translate>

```
return myBody
```

<translate>

Then, we return our Part solid as the result of our function. That Part solid, like any other Part shape, can be attributed to an object in a FreeCAD document, with:

</translate>

```
myObject = FreeCAD.ActiveDocument.addObject("Part::Feature", "myObject")
myObject.Shape = bottle
```

<translate>

or, more simple:

</translate>

```
Part.show(bottle)
```

<translate>

Box pierced

Here a complete example of building a box pierced.

The construction is done side by side and when the cube is finished, it is hollowed out of a cylinder through.

</translate>

```
import Draft, Part, FreeCAD, math, PartGui, FreeCADGui, PyQt4
from math import sqrt, pi, sin, cos, asin
from FreeCAD import Base

size = 10
poly = Part.makePolygon( [ (0,0,0), (size, 0, 0), (size, 0, size), (0, 0, size), (0, 0, 0)])

face1 = Part.Face(poly)
face2 = Part.Face(poly)
face3 = Part.Face(poly)
face4 = Part.Face(poly)
face5 = Part.Face(poly)
face6 = Part.Face(poly)

myMat = FreeCAD.Matrix()
myMat.rotateZ(math.pi/2)
face2.transformShape(myMat)
face2.translate(FreeCAD.Vector(size, 0, 0))

myMat.rotateZ(math.pi/2)
face3.transformShape(myMat)
face3.translate(FreeCAD.Vector(size, size, 0))

myMat.rotateZ(math.pi/2)
face4.transformShape(myMat)
face4.translate(FreeCAD.Vector(0, size, 0))

myMat = FreeCAD.Matrix()
myMat.rotateX(-math.pi/2)
face5.transformShape(myMat)

face6.transformShape(myMat)
face6.translate(FreeCAD.Vector(0,0,size))

myShell = Part.makeShell([face1,face2,face3,face4,face5,face6])

mySolid = Part.makeSolid(myShell)
mySolidRev = mySolid.copy()
mySolidRev.reverse()

myCyl = Part.makeCylinder(2,20)
myCyl.translate(FreeCAD.Vector(size/2, size/2, 0))

cut_part = mySolidRev.cut(myCyl)

Part.show(cut_part)
```

<translate>

Loading and Saving

There are several ways to save your work in the Part module. You can of course save your FreeCAD document, but you can also save Part objects directly to common CAD formats, such as BREP, IGS, STEP and STL.

Saving a shape to a file is easy. There are `exportBrep()`, `exportIges()`, `exportStl()` and `exportStep()` methods availables for all shape objects. So, doing:

</translate>

```
import Part
s = Part.makeBox(0,0,0,10,10,10)
s.exportStep("test.stp")
```

<translate>

this will save our box into a STEP file. To load a BREP, IGES or STEP file, simply do the contrary:

</translate>

```
import Part
s = Part.Shape()
s.read("test.stp")
```

<translate>

To convert an **.stp** in **.igs** file simply :

</translate>

```
import Part
s = Part.Shape()
s.read("file.stp") # incoming file igs, stp, stl, brep
s.exportIges("file.igs") # outbound file igs
```

<translate>

Note that importing or opening BREP, IGES or STEP files can also be done directly from the File -> Open or File -> Import menu, while exporting is with File -> Export

< previous: [Mesh Scripting \(/wiki/index.php?title=Mesh_Scripting\)](/wiki/index.php?title=Mesh_Scripting)
 Index next: [Mesh to Part > \(/wiki/index.php?title=Mesh_to_Part\)](/wiki/index.php?title=Mesh_to_Part)
[\(/wiki/index.php?title=Online_Help_Toc\)](/wiki/index.php?title=Online_Help_Toc)

< /translate>

< translate>

Converting Part objects to Meshes

Converting higher-level objects such as Part shapes (/wiki/index.php?title=Part_Module) into simpler objects such as meshes (/wiki/index.php?title=Mesh_Module) is a pretty simple operation, where all faces of a Part object get triangulated. The result of that triangulation (tessellation) is then used to construct a mesh: (let's assume our document contains one part object)< /translate>

```
#let's assume our document contains one part object
import Mesh
faces = []
shape = FreeCAD.ActiveDocument.ActiveObject.Shape
triangles = shape.tessellate(1) # the number represents the precision of the tessellation)
for tri in triangles[1]:
    face = []
    for i in range(3):
        vindex = tri[i]
        face.append(triangles[0][vindex])
    faces.append(face)
m = Mesh.Mesh(faces)
Mesh.show(m)
```

<translate> Sometimes the triangulation of certain faces offered by OpenCascade is quite ugly. If the face has a rectangular parameter space and doesn't contain any holes or other trimming curves you can also create a mesh on your own: < /translate>

```
import Mesh
def makeMeshFromFace(u,v,face):
    (a,b,c,d)=face.ParameterRange
    pts=[]
    for j in range(v):
        for i in range(u):
            s=1.0/(u-1)*(i*b+(u-1-i)*a)
            t=1.0/(v-1)*(j*d+(v-1-j)*c)
            pts.append(face.valueAt(s,t))

    mesh=Mesh.Mesh()
    for j in range(v-1):
        for i in range(u-1):
            mesh.addFacet(pts[u*j+i],pts[u*j+i+1],pts[u*(j+1)+i])
            mesh.addFacet(pts[u*(j+1)+i],pts[u*j+i+1],pts[u*(j+1)+i+1])

    return mesh
```

<translate>

Converting Meshes to Part objects

Converting Meshes to Part objects is an extremely important operation in CAD work, because very often you receive 3D data in mesh format from other people or outputted from other applications. Meshes are very practical to represent free-form geometry and big visual scenes, as it is very lightweight, but for CAD we generally prefer higher-level objects that carry much more information, such as the idea of solid, or faces made of curves instead of triangles.

Converting meshes to those higher-level objects (handled by the Part Module (/wiki/index.php?title=Part_Module) in FreeCAD) is not an easy operation. Meshes can be made of thousands of triangles (for example when generated by a 3D scanner), and having solids made of the same number of faces would be extremely heavy to manipulate. So you generally want to optimize the object when converting.

FreeCAD currently offers two methods to convert Meshes to Part objects. The first method is a simple, direct conversion, without any optimization:< /translate>

```
import Mesh,Part
mesh = Mesh.createTorus()
shape = Part.Shape()
shape.makeShapeFromMesh(mesh.Topology,0.05) # the second arg is the tolerance for sewing
solid = Part.makeSolid(shape)
Part.show(solid)
```

<translate> The second method offers the possibility to consider mesh facets coplanar when the angle between them is under a certain value. This allows to build much simpler shapes: (let's assume our document contains one Mesh object) < /translate>

```

# let's assume our document contains one Mesh object
import Mesh,Part,MeshPart
faces = []
mesh = App.ActiveDocument.ActiveObject.Mesh
segments = mesh.getPlanes(0.00001) # use rather strict tolerance here

for i in segments:
    if len(i) > 0:
        # a segment can have inner holes
        wires = MeshPart.wireFromSegment(mesh, i)
        # we assume that the exterior boundary is that one with the biggest bounding box
        if len(wires) > 0:
            ext=None
            max_length=0
            for i in wires:
                if i.BoundingBox.DiagonalLength > max_length:
                    max_length = i.BoundingBox.DiagonalLength
                    ext = i

            wires.remove(ext)
            # all interior wires mark a hole and must reverse their orientation, otherwise Part.Face fails
            for i in wires:
                i.reverse()

            # make sure that the exterior wires comes as first in the list
            wires.insert(0, ext)
            faces.append(Part.Face(wires))

shell=Part.Compound(faces)
Part.show(shell)
#solid = Part.Solid(Part.Shell(faces))
#Part.show(solid)

```

<translate>

< previous: Topological data scripting (/wiki/index.php?title=Topological_data_scripting)

Index next: Scenegraph > (</wiki/index.php?title=Scenegraph>)
(/wiki/index.php?title=Online_Help_Toc)

</translate>

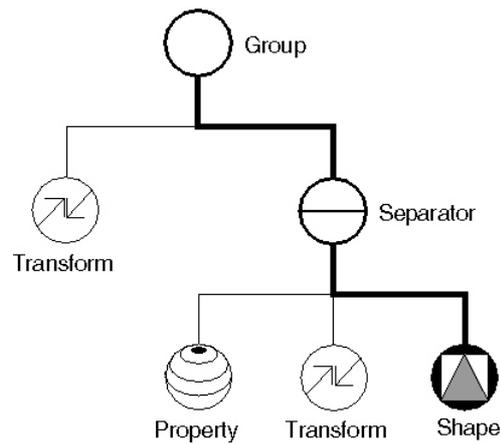
< translate> FreeCAD is basically a collage of different powerful libraries, the most important being openCascade (http://en.wikipedia.org/wiki/Open_CASCADE), for managing and constructing geometry, Coin3d (<http://en.wikipedia.org/wiki/Coin3D>) to display that geometry, and Qt ([http://en.wikipedia.org/wiki/Qt_\(toolkit\)](http://en.wikipedia.org/wiki/Qt_(toolkit))) to put all this in a nice Graphical User Interface.

The geometry that appears in the 3D views of FreeCAD are rendered by the Coin3D library. Coin3D is an implementation of the OpenInventor (http://en.wikipedia.org/wiki/Open_Inventor) standard. The openCascade software also provides the same functionality, but it was decided, at the very beginnings of FreeCAD, not to use the built-in openCascade viewer and rather switch to the more performant coin3D software. A good way to learn about that library is the book Open Inventor Mentor (http://www-evasion.imag.fr/Membres/Francois.Faure/doc/inventorMentor/sgi_html/).

OpenInventor (http://en.wikipedia.org/wiki/Open_Inventor) is actually a 3D scene description language. The scene described in openInventor is then rendered in OpenGL on your screen. Coin3D takes care of doing this, so the programmer doesn't need to deal with complex openGL calls, he just has to provide it with valid OpenInventor code. The big advantage is that openInventor is a very well-known and well documented standard.

One of the big jobs FreeCAD does for you is basically to translate openCascade geometry information into openInventor language.

OpenInventor describes a 3D scene in the form of a scenegraph (http://en.wikipedia.org/wiki/Scene_graph), like the one below:



(/wiki/index.php?

— Path

title=File:Scenegraph.gif) image from Inventor mentor (http://www-evasion.imag.fr/~Francois.Faure/doc/inventorMentor/sgi_html/index.html)

An openInventor scenegraph describes everything that makes part of a 3D scene, such as geometry, colors, materials, lights, etc, and organizes all that data in a convenient and clear structure. Everything can be grouped into sub-structures, allowing you to organize your scene contents pretty much the way you like. Here is an example of an openInventor file:< /translate>

```
#Inventor V2.0 ascii

Separator {
  RotationXYZ {
    axis Z
    angle 0
  }
  Transform {
    translation 0 0 0.5
  }
  Separator {
    Material {
      diffuseColor 0.05 0.05 0.05
    }
    Transform {
      rotation 1 0 0 1.5708
      scaleFactor 0.2 0.5 0.2
    }
    Cylinder {
    }
  }
}
```

<translate>

As you can see, the structure is very simple. You use separators to organize your data into blocks, a bit like you would organize your files into folders. Each statement affects what comes next, for example the first two items of our root separator are a rotation and a translation, both will affect the next item, which is a separator. In that separator, a material is defined, and another transformation. Our cylinder will therefore be affected by both transformations, the one who was applied directly to it and the one that was applied to its parent separator.

We also have many other types of elements to organize our scene, such as groups, switches or annotations. We can define very complex materials for our objects, with color, textures, shading modes and transparency. We can also define lights, cameras, and even movement. It is even possible to embed pieces of scripting in openInventor files, to define more complex behaviours.

If you are interested in learning more about openInventor, head directly to its most famous reference, the Inventor mentor (http://www-evasion.imag.fr/~Francois.Faure/doc/inventorMentor/sgi_html/index.html).

In FreeCAD, normally, we don't need to interact directly with the openInventor scenegraph. Every object in a FreeCAD document, being a mesh, a part shape or anything else, gets automatically converted to openInventor code and inserted in the main scenegraph that you see in a 3D view. That scenegraph gets updated continuously when you do modifications, add or remove objects to the document. In fact, every object (in App space) has a view provider (a corresponding object in Gui space), responsible for issuing openInventor code.

But there are many advantages to be able to access the scenegraph directly. For example, we can temporarily change the appearance of an object, or we can add objects to the scene that have no real existence in the FreeCAD document, such as construction geometry, helpers, graphical hints or tools such as manipulators or on-screen information.

FreeCAD itself features several tools to see or modify openInventor code. For example, the following python code will show the openInventor representation of a selected object:< /translate>

```
obj = FreeCAD.ActiveDocument.ActiveObject
viewprovider = obj.ViewObject
print viewprovider.toString()
```

<translate> But we also have a python module that allows complete access to anything managed by Coin3D, such as our FreeCAD scenegraph. So, read on to Pivy (</wiki/index.php?title=Pivy>).

< previous: [Mesh to Part \(/wiki/index.php?title=Mesh_to_Part\)](/wiki/index.php?title=Mesh_to_Part)
 Index ([/wiki/index.php? next: Pivy > \(/wiki/index.php?title=Pivy\)](/wiki/index.php?next=Pivy)
[title=Online_Help_Toc](/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate> Pivy (<http://pivy.coin3d.org/>) is a python binding library for Coin3d (<http://www.coin3d.org/>), the 3D-rendering library used FreeCAD. When imported in a running python interpreter, it allows to dialog directly with any running Coin3d scenegraphs (</wiki/index.php?title=Scenegraph>), such as the FreeCAD 3D views, or even to create new ones. Pivy is bundled in standard FreeCAD installation.

The coin library is divided into several pieces, coin itself, for manipulating scenegraphs and bindings for several GUI systems, such as windows or, like in our case, qt. Those modules are available to pivy too, depending if they are present on the system. The coin module is always present, and it is what we will use anyway, since we won't need to care about anchoring our 3D display in any interface, it is already done by FreeCAD itself. All we need to do is this:< /translate>

```
from pivy import coin
```

<translate>

Accessing and modifying the scenegraph

We saw in the Scenegraph (</wiki/index.php?title=Scenegraph>) page how a typical Coin scene is organized. Everything that appears in a FreeCAD 3D view is a coin scenegraph, organized the same way. We have one root node, and all objects on the screen are its children.

FreeCAD has an easy way to access the root node of a 3D view scenegraph:< /translate>

```
sg = FreeCADGui.ActiveDocument.ActiveView.getSceneGraph()
print sg
```

<translate> This will return the root node:< /translate>

```
<pivy.coin.SoSelection; proxy of <Swig Object of type 'SoSelection *' at 0x360cb60 >
```

<translate> We can inspect the immediate children of our

```
scene:< /translate>
```

```
for node in sg.getChildren():
    print node
```

<translate> Some of those nodes, such as SoSeparators or SoGroups, can have children themselves. The complete list of the available coin objects can be found in the official coin documentation (<http://doc.coin3d.org/Coin/classes.html>).

Let's try to add something to our scenegraph now. We'll add a nice red cube:< /translate>

```
col = coin.SoBaseColor()
col.rgb=(1,0,0)
cub = coin.SoCube()
myCustomNode = coin.SoSeparator()
myCustomNode.addChild(col)
myCustomNode.addChild(cub)
sg.addChild(myCustomNode)
```

<translate> and here is our (nice) red cube. Now, let's try this:< /translate>

```
col.rgb=(1,1,0)
```

<translate> See? everything is still accessible and modifiable on-the-fly. No need to recompute or redraw anything, coin takes care of everything. You can add stuff to your scenegraph, change properties, hide stuff, show temporary objects, anything. Of course, this only concerns the display in the 3D view. That display gets recomputed by FreeCAD on file open, and when an object needs recomputing. So, if you change the aspect of an existing FreeCAD object, those changes will be lost if the object gets recomputed or when you reopen the file.

A key to work with scenegraphs in your scripts is to be able to access certain properties of the nodes you added when needed. For example, if we wanted to move our cube, we would have added a SoTranslation node to our custom node, and it would have looked like this:< /translate>

```
col = coin.SoBaseColor()
col.rgb=(1,0,0)
trans = coin.SoTranslation()
trans.translation.setValue([0,0,0])
cub = coin.SoCube()
myCustomNode = coin.SoSeparator()
myCustomNode.addChild(col)
myCustomNode.addChild(trans)
myCustomNode.addChild(cub)
sg.addChild(myCustomNode)
```

<translate> Remember that in an openInventor scenegraph, the order is important. A node affects what comes next, so you can say something like: color red, cube, color yellow, sphere, and you will get a red cube and a yellow sphere. If we added the translation now to our existing custom node, it would come after the cube, and not affect it. If we had inserted it when creating it, like here above, we could now do:< /translate>

```
trans.translation.setValue([2,0,0])
```

<translate> And our cube would jump 2 units to the right. Finally, removing something is done with:< /translate>

```
sg.removeChild(myCustomNode)
```

<translate>

Using callback mechanisms

A callback mechanism (http://en.wikipedia.org/wiki/Callback_%28computer_science%29) is a system that permits a library that you are using, such as our coin library, to call you back, that is, to call a certain function from your currently running python object. This is extremely useful,

because that way coin can notify you if some specific event occurs in the scene. Coin can watch very different things, such as mouse position, clicks of a mouse button, keyboard keys being pressed, and many other things.

FreeCAD features an easy way to use such callbacks:< /translate>

```
class ButtonTest:
    def __init__(self):
        self.view = FreeCADGui.ActiveDocument.ActiveView
        self.callback = self.view.addEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.getMouseEventClick)
    def getMouseEventClick(self,event_cb):
        event = event_cb.getEvent()
        if event.getState() == SoMouseButtonEvent.DOWN:
            print "Alert!!! A mouse button has been improperly clicked!!!"
            self.view.removeEventCallbackSWIG(SoMouseButtonEvent.getClassTypeId(),self.callback)

ButtonTest()
```

<translate> The callback has to be initiated from an object, because that object must still be running when the callback will occur. See also a complete [list \(/wiki/index.php?title=Code_snippets#Observing_mouse_events_in_the_3D_viewer_via_Python\)](/wiki/index.php?title=Code_snippets#Observing_mouse_events_in_the_3D_viewer_via_Python) of possible events and their parameters, or the official coin documentation (<http://doc.coin3d.org/Coin/classes.html>).

Documentation

Unfortunately pivy itself still doesn't have a proper documentation, but since it is an accurate translation of coin, you can safely use the coin documentation as reference, and use python style instead of c++ style (for example `SoFile::getClassTypeId()` would in pivy be `SoFile.getClassId()`)

< previous: [Scenegraph \(/wiki/index.php?title=Scenegraph\)](/wiki/index.php?title=Scenegraph) Index [\(/wiki/index.php?title=PySide\)](/wiki/index.php?title=PySide) next: [PySide > \(/wiki/index.php?title=PySide\)](/wiki/index.php?title=PySide) [title=Online_Help_Toc](/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate>

PySide

PySide (<http://en.wikipedia.org/wiki/PySide>) is a Python binding of the cross-platform GUI toolkit Qt. FreeCAD uses PySide for all GUI (Graphic User Intercase) purposes. PySide evolved from the PyQt package which was previously used by FreeCAD for it's GUI. See [Differences Between PySide and PyQt \(http://qt-project.org/wiki/Differences_Between_PySide_and_PyQt\)](http://qt-project.org/wiki/Differences_Between_PySide_and_PyQt) for more information on the differences.

Users of FreeCAD often achieve everything using the built-in interface. But for users who want to customise their operations then the Python interface exists which is documented in the [Python Scripting Tutorial \(/wiki/index.php?title=Python_scripting_tutorial\)](/wiki/index.php?title=Python_scripting_tutorial). The Python interface for FreeCAD had great flexibility and power. For it's user interaction Python with FreeCAD uses PySide, which is what is documented on this page.

Python offers the 'print' statement which gives the code:< /translate>

```
print 'Hello World'
```

<translate> With Python's print statement you have only limited control of the appearance and behaviour. PySide supplies the missing control and also handles environments (such as the FreeCAD macro file environment) where the built-in facilities of Python are not enough.

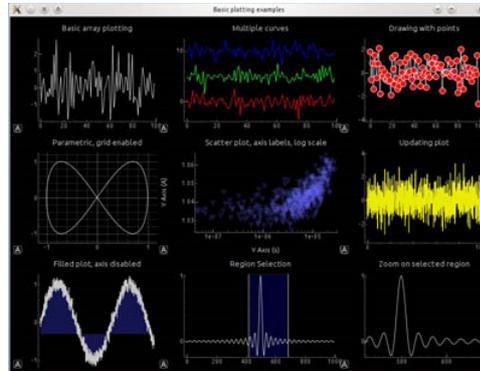
PySide's abilities range from:



(/wiki/index.php?

title=File:PySideScreenSnapshot1.jpg)

to:



(/wiki/index.php?

title=File:PySideScreenSnapshot2.jpg)

PySide is described in the following 3 pages which should follow on one from each other:

- [Beginner PySide Examples \(/wiki/index.php?title=PySide_Beginner_Examples\)](#) (Hello World, announcements, enter text, enter number)
- [Medium PySide Examples \(/wiki/index.php?title=PySide_Medium_Examples\)](#) (window sizing, hiding widgets, popup menus, mouse position, mouse events)
- [Advanced PySide Examples \(/wiki/index.php?title=PySide_Advanced_Examples\)](#) (widgets etc.)

They divide the subject matter into 3 parts, differentiated by level of exposure to PySide, Python and the FreeCAD internals. The first page has overview and background material giving a description of PySide and how it is put together while the second and third pages are mostly code examples at different levels.

The intention is that the associated pages will provide simple Python code to run PySide so that the user working on a problem can easily copy the code, paste it into their own work, adapt it as necessary and return to their problem solving with FreeCAD. Hopefully they don't have to go chasing off across the internet looking for answers to PySide questions. At the same time this page is not intended to replace the various comprehensive PySide tutorials and reference sites available on the web.

< [previous: Pivy \(/wiki/index.php?title=Pivy\)](#) [Index](#)
 next: [Scripted objects > \(/wiki/index.php?title=Scripted_objects\)](#)
[\(/wiki/index.php?title=Online_Help_Toc\)](#)
 </translate>

Besides the standard object types such as annotations, meshes and parts objects, FreeCAD also offers the amazing possibility to build 100% python-scripted objects, called Python Features. Those objects will behave exactly as any other FreeCAD object, and are saved and restored automatically on file save/load.

One particularity must be understood, those objects are saved in FreeCAD FcStd files with python's json (<http://docs.python.org/2/library/json.html>) module. That module turns a python object as a string, allowing it to be added to the saved file. On load, the json module uses that string to

recreate the original object, provided it has access to the source code that created the object. This means that if you save such a custom object and open it on a machine where the python code that generated the object is not present, the object won't be recreated. If you distribute such objects to others, you will need to distribute the python script that created it together.

Python Features follow the same rule as all FreeCAD features: they are separated into App and GUI parts. The app part, the Document Object, defines the geometry of our object, while its GUI part, the View Provider Object, defines how the object will be drawn on screen. The View Provider Object, as any other FreeCAD feature, is only available when you run FreeCAD in its own GUI. There are several properties and methods available to build your object. Properties must be of any of the predefined properties types that FreeCAD offers, and will appear in the property view window, so they can be edited by the user. This way, FeaturePython objects are truly and totally parametric. you can define properties for the Object and its ViewObject separately.

Hint: In former versions we used Python's cPickle (<http://docs.python.org/release/2.5/lib/module-cPickle.html>) module. However, this module executes arbitrary code and thus causes a security problem. Thus, we moved to Python's json module.

Basic example

The following sample can be found in the `src/Mod/TemplatePyMod/FeaturePython.py` (<https://github.com/FreeCAD/FreeCAD/blob/master/src/Mod/TemplatePyMod/Feat> file, together with several other examples:

```

'''Examples for a feature class and its view provider.'''

import FreeCAD, FreeCADGui
from pivy import coin

class Box:
    def __init__(self, obj):
        '''Add some custom properties to our box feature'''
        obj.addProperty("App::PropertyLength", "Length", "Box", "Length of the box").Length=1.0
        obj.addProperty("App::PropertyLength", "Width", "Box", "Width of the box").Width=1.0
        obj.addProperty("App::PropertyLength", "Height", "Box", "Height of the box").Height=1.0
        obj.Proxy = self

    def onChanged(self, fp, prop):
        '''Do something when a property has changed'''
        FreeCAD.Console.PrintMessage("Change property: " + str(prop) + "\n")

    def execute(self, fp):
        '''Do something when doing a recomputation, this method is mandatory'''
        FreeCAD.Console.PrintMessage("Recompute Python Box feature\n")

class ViewProviderBox:
    def __init__(self, obj):
        '''Set this object to the proxy object of the actual view provider'''
        obj.addProperty("App::PropertyColor", "Color", "Box", "Color of the box").Color=(1.0,0.0,
0.0)
        obj.Proxy = self

    def attach(self, obj):
        '''Setup the scene sub-graph of the view provider, this method is mandatory'''
        self.shaded = coin.SoGroup()
        self.wireframe = coin.SoGroup()
        self.scale = coin.SoScale()
        self.color = coin.SoBaseColor()

        data=coin.SoCube()
        self.shaded.addChild(self.scale)
        self.shaded.addChild(self.color)
        self.shaded.addChild(data)
        obj.addDisplayMode(self.shaded, "Shaded");
        style=coin.SoDrawStyle()
        style.style = coin.SoDrawStyle.LINES
        self.wireframe.addChild(style)
        self.wireframe.addChild(self.scale)
        self.wireframe.addChild(self.color)
        self.wireframe.addChild(data)
        obj.addDisplayMode(self.wireframe, "Wireframe");
        self.onChanged(obj, "Color")

    def updateData(self, fp, prop):
        '''If a property of the handled feature has changed we have the chance to handle this
here'''
        # fp is the handled feature, prop is the name of the property that has changed
        l = fp.getPropertyByName("Length")
        w = fp.getPropertyByName("Width")
        h = fp.getPropertyByName("Height")
        self.scale.scaleFactor.setValue(float(l),float(w),float(h))
        pass

    def getDisplayModes(self,obj):
        '''Return a list of display modes.'''
        modes=[]
        modes.append("Shaded")
        modes.append("Wireframe")
        return modes

    def getDefaultDisplayMode(self):
        '''Return the name of the default display mode. It must be defined in getDisplayModes.
...
        return "Shaded"

    def setDisplayMode(self,mode):
        '''Map the display mode defined in attach with those defined in getDisplayModes.\
Since they have the same names nothing needs to be done. This method is option
al'''
        return mode

    def onChanged(self, vp, prop):
        '''Here we can do something when a single property got changed'''
        FreeCAD.Console.PrintMessage("Change property: " + str(prop) + "\n")
        if prop == "Color":
            c = vp.getPropertyByName("Color")
            self.color.rgb.setValue(c[0],c[1],c[2])

    def getIcon(self):
        '''Return the icon in XPM format which will appear in the tree view. This method is\
optional and if not defined a default icon is shown.'''

```



```

App::PropertyBool
App::PropertyBoolList
App::PropertyFloat
App::PropertyFloatList
App::PropertyFloatConstraint
App::PropertyQuantity
App::PropertyQuantityConstraint
App::PropertyAngle
App::PropertyDistance
App::PropertyLength
App::PropertySpeed
App::PropertyAcceleration
App::PropertyForce
App::PropertyPressure
App::PropertyInteger
App::PropertyIntegerConstraint
App::PropertyPercent
App::PropertyEnumeration
App::PropertyIntegerList
App::PropertyIntegerSet
App::PropertyMap
App::PropertyString
App::PropertyUUID
App::PropertyFont
App::PropertyStringList
App::PropertyLink
App::PropertyLinkSub
App::PropertyLinkList
App::PropertyLinkSubList
App::PropertyMatrix
App::PropertyVector
App::PropertyVectorList
App::PropertyPlacement
App::PropertyPlacementLink
App::PropertyColor
App::PropertyColorList
App::PropertyMaterial
App::PropertyPath
App::PropertyFile
App::PropertyFileIncluded
App::PropertyPythonObject
Part::PropertyPartShape
Part::PropertyGeometryList
Part::PropertyShapeHistory
Part::PropertyFilletEdges
Sketcher::PropertyConstraintList

```

When adding properties to your custom objects, take care of this:

- Do not use characters "<" or ">" in the properties descriptions (that would break the xml pieces in the .fcstd file)
- Properties are stored alphabetically in a .fcstd file. If you have a shape in your properties, any property whose name comes after "Shape" in alphabetic order, will be loaded AFTER the shape, which can cause strange behaviours.

Property Type

By default the properties can be updated. It is possible to make the properties read-only, for instance in the case one wants to show the result of a method. It is also possible to hide the property. The property type can be set using

```
obj.setEditorMode("MyPropertyName", mode)
```

where mode is a short int that can be set to:

```

0 -- default mode, read and write
1 -- read-only
2 -- hidden

```

The EditorModes are not set at FreeCAD file reload. This could be done by the `__setstate__` function. See <http://forum.freecadweb.org/viewtopic.php?f=18&t=13460&start=10#p108072> (<http://forum.freecadweb.org/viewtopic.php?f=18&t=13460&start=10#p108072>). By using the `setEditorMode` the properties are only read only in PropertyEditor. They could still be changed from

python. To really make them read only the setting has to be passed directly inside the `addProperty` function. See <http://forum.freecadweb.org/viewtopic.php?f=18&t=13460&start=20#p109709> (<http://forum.freecadweb.org/viewtopic.php?f=18&t=13460&start=20#p109709>) for an example.

Other more complex example

This example makes use of the Part Module (/wiki/index.php?title=Part_Module) to create an octahedron, then creates its coin representation with pivy.

First is the Document object itself:

```
import FreeCAD, FreeCADGui, Part
import pivy
from pivy import coin

class Octahedron:
    def __init__(self, obj):
        "Add some custom properties to our box feature"
        obj.addProperty("App::PropertyLength", "Length", "Octahedron", "Length of the octahedron").Length=1.0
        obj.addProperty("App::PropertyLength", "Width", "Octahedron", "Width of the octahedron").Width=1.0
        obj.addProperty("App::PropertyLength", "Height", "Octahedron", "Height of the octahedron").Height=1.0
        obj.addProperty("Part::PropertyPartShape", "Shape", "Octahedron", "Shape of the octahedron")
        obj.Proxy = self

    def execute(self, fp):
        # Define six vertices for the shape
        v1 = FreeCAD.Vector(0,0,0)
        v2 = FreeCAD.Vector(fp.Length,0,0)
        v3 = FreeCAD.Vector(0,fp.Width,0)
        v4 = FreeCAD.Vector(fp.Length,fp.Width,0)
        v5 = FreeCAD.Vector(fp.Length/2,fp.Width/2,fp.Height/2)
        v6 = FreeCAD.Vector(fp.Length/2,fp.Width/2,-fp.Height/2)

        # Make the wires/faces
        f1 = self.make_face(v1,v2,v5)
        f2 = self.make_face(v2,v4,v5)
        f3 = self.make_face(v4,v3,v5)
        f4 = self.make_face(v3,v1,v5)
        f5 = self.make_face(v2,v1,v6)
        f6 = self.make_face(v4,v2,v6)
        f7 = self.make_face(v3,v4,v6)
        f8 = self.make_face(v1,v3,v6)
        shell=Part.makeShell([f1,f2,f3,f4,f5,f6,f7,f8])
        solid=Part.makeSolid(shell)
        fp.Shape = solid

    # helper method to create the faces
    def make_face(self,v1,v2,v3):
        wire = Part.makePolygon([v1,v2,v3,v1])
        face = Part.Face(wire)
        return face
```

Then, we have the view provider object, responsible for showing the object in the 3D scene:

```

class ViewProviderOctahedron:
    def __init__(self, obj):
        "Set this object to the proxy object of the actual view provider"
        obj.addProperty("App::PropertyColor", "Color", "Octahedron", "Color of the octahedron").Color=(1.0,0.0,0.0)
        obj.Proxy = self

    def attach(self, obj):
        "Setup the scene sub-graph of the view provider, this method is mandatory"
        self.shaded = coin.SoGroup()
        self.wireframe = coin.SoGroup()
        self.scale = coin.SoScale()
        self.color = coin.SoBaseColor()

        self.data=coin.SoCoordinate3()
        self.face=coin.SoIndexedLineSet()

        self.shaded.addChild(self.scale)
        self.shaded.addChild(self.color)
        self.shaded.addChild(self.data)
        self.shaded.addChild(self.face)
        obj.addDisplayMode(self.shaded, "Shaded");
        style=coin.SoDrawStyle()
        style.style = coin.SoDrawStyle.LINES
        self.wireframe.addChild(style)
        self.wireframe.addChild(self.scale)
        self.wireframe.addChild(self.color)
        self.wireframe.addChild(self.data)
        self.wireframe.addChild(self.face)
        obj.addDisplayMode(self.wireframe, "Wireframe");
        self.onChanged(obj, "Color")

    def updateData(self, fp, prop):
        "If a property of the handled feature has changed we have the chance to handle this here"
        # fp is the handled feature, prop is the name of the property that has changed
        if prop == "Shape":
            s = fp.getPropertyByName("Shape")
            self.data.point.setNum(6)
            cnt=0
            for i in s.Vertices:
                self.data.point.set1Value(cnt,i.X,i.Y,i.Z)
                cnt=cnt+1

            self.face.coordIndex.set1Value(0,0)
            self.face.coordIndex.set1Value(1,1)
            self.face.coordIndex.set1Value(2,2)
            self.face.coordIndex.set1Value(3,-1)

            self.face.coordIndex.set1Value(4,1)
            self.face.coordIndex.set1Value(5,3)
            self.face.coordIndex.set1Value(6,2)
            self.face.coordIndex.set1Value(7,-1)

            self.face.coordIndex.set1Value(8,3)
            self.face.coordIndex.set1Value(9,4)
            self.face.coordIndex.set1Value(10,2)
            self.face.coordIndex.set1Value(11,-1)

            self.face.coordIndex.set1Value(12,4)
            self.face.coordIndex.set1Value(13,0)
            self.face.coordIndex.set1Value(14,2)
            self.face.coordIndex.set1Value(15,-1)

            self.face.coordIndex.set1Value(16,1)
            self.face.coordIndex.set1Value(17,0)
            self.face.coordIndex.set1Value(18,5)
            self.face.coordIndex.set1Value(19,-1)

            self.face.coordIndex.set1Value(20,3)
            self.face.coordIndex.set1Value(21,1)
            self.face.coordIndex.set1Value(22,5)
            self.face.coordIndex.set1Value(23,-1)

            self.face.coordIndex.set1Value(24,4)
            self.face.coordIndex.set1Value(25,3)
            self.face.coordIndex.set1Value(26,5)
            self.face.coordIndex.set1Value(27,-1)

            self.face.coordIndex.set1Value(28,0)
            self.face.coordIndex.set1Value(29,4)
            self.face.coordIndex.set1Value(30,5)
            self.face.coordIndex.set1Value(31,-1)

    def getDisplayModes(self, obj):
        "Return a list of display modes."
        modes=[]
        modes.append("Shaded")

```


Simply, you create a SoFCSelection node, then you add your geometry nodes to it, then you add it to your main node, instead of adding your geometry nodes directly.

Working with simple shapes

If your parametric object simply outputs a shape, you don't need to use a view provider object. The shape will be displayed using FreeCAD's standard shape representation:

```
import FreeCAD as App
import FreeCADGui
import FreeCAD
import Part
class Line:
    def __init__(self, obj):
        '''App two point properties'''
        obj.addProperty("App::PropertyVector", "p1", "Line", "Start point")
        obj.addProperty("App::PropertyVector", "p2", "Line", "End point").p2=FreeCAD.Vector(1,0,0
    )
        obj.Proxy = self

    def execute(self, fp):
        '''Print a short message when doing a recomputation, this method is mandatory'''
        fp.Shape = Part.makeLine(fp.p1,fp.p2)

a=FreeCAD.ActiveDocument.addObject("Part::FeaturePython", "Line")
Line(a)
a.ViewObject.Proxy=0 # just set it to something different from None (this assignment is needed
to run an internal notification)
FreeCAD.ActiveDocument.recompute()
```

Same code with use **ViewProviderLine**

```
import FreeCAD as App
import FreeCADGui
import FreeCAD
import Part

class Line:
    def __init__(self, obj):
        '''App two point properties'''
        obj.addProperty("App::PropertyVector", "p1", "Line", "Start point")
        obj.addProperty("App::PropertyVector", "p2", "Line", "End point").p2=FreeCAD.Vector(100,
0,0)
        obj.Proxy = self

    def execute(self, fp):
        '''Print a short message when doing a recomputation, this method is mandatory'''
        fp.Shape = Part.makeLine(fp.p1,fp.p2)

class ViewProviderLine:
    def __init__(self, obj):
        ''' Set this object to the proxy object of the actual view provider'''
        obj.Proxy = self

    def getDefaultDisplayMode(self):
        ''' Return the name of the default display mode. It must be defined in getDisplayModes.
        ...
        return "Flat Lines"

a=FreeCAD.ActiveDocument.addObject("Part::FeaturePython", "Line")
Line(a)
ViewProviderLine(a.ViewObject)
App.ActiveDocument.recompute()
```

Further informations

There are a few very interesting forum threads about scripted objects:

- <http://forum.freecadweb.org/viewtopic.php?f=22&t=13740>
(<http://forum.freecadweb.org/viewtopic.php?f=22&t=13740>)

- <http://forum.freecadweb.org/viewtopic.php?t=12139>
(<http://forum.freecadweb.org/viewtopic.php?t=12139>)

the main() function, and we make a list of its objects. Then, as we choosed only to care about Part geometry, we check if the Type property of each object contains "Part", then we tessellate it.

The tessellation produce a list of vertices and a list of faces defined by vertices indexes. This is perfect, since it is exactly the same way as blender defines meshes. So, our task is ridiculously simple, we just add both lists contents to the verts and faces of a blender mesh. When everything is done, we just redraw the screen, and that's it!

Of course this script is very simple (in fact I made a more advanced here (http://yorik.orgfree.com/scripts/import_freecad.py)), you might want to extend it, for example importing mesh objects too, or importing Part geometry that has no faces, or import other file formats that FreeCAD can read. You might also want to export geometry to a FreeCAD document, which can be done the same way. You might also want to build a dialog, so the user can choose what to import, etc... The beauty of all this actually lies in the fact that you let FreeCAD do the ground work while presenting its results in the program of your choice.

Using FreeCAD with GUI

From version 4.2 on Qt has the intriguing ability to embed Qt-GUI-dependent plugins into non-Qt host applications and share the host's event loop.

Especially, for FreeCAD this means that it can be imported from within another application with its whole user interface where the host application has full control over FreeCAD, then.

The whole python code to achieve that has only two lines

</translate>

```
import FreeCADGui
FreeCADGui.showMainWindow()
```

<translate>

If the host application is based on Qt then this solution should work on all platforms which Qt supports. However, the host should link the same Qt version as FreeCAD because otherwise you could run into unexpected runtime errors.

For non-Qt applications, however, there are a few limitations you must be aware of. This solution probably doesn't work together with all other toolkits. For Windows it works as long as the host application is directly based on Win32 or any other toolkit that internally uses the Win32 API such as wxWidgets, MFC or WinForms. In order to get it working under X11 the host application must link the "glib" library.

Note, for any console application this solution of course doesn't work because there is no event loop running.

< previous: Scripted objects (/wiki/index.php?title=Scripted_objects)

Index next: Code snippets > (/wiki/index.php?title=Code_snippets) (/wiki/index.php?title=Online_Help_Toc)

</translate>

This page contains examples, pieces, chunks of FreeCAD python code collected from users experiences and discussions on the forums. Read and use it as a start for your own scripts...

A typical InitGui.py file

 (/wiki/index.php?title=File:Base_ExampleCommandModel.png) Tutorial
Topic
Python
Level
Beginner
Time to complete
Author
FreeCAD version
Example File(s)

Every module must contain, besides your main module file, an `InitGui.py` file, responsible for inserting the module in the main Gui. This is an example of a simple one.

```
class ScriptWorkbench (Workbench):
    MenuText = "Scripts"
    def Initialize(self):
        import Scripts # assuming Scripts.py is your module
        list = ["Script_Cmd"] # That list must contain command names, that can be defined in S
cripts.py
        self.appendToolbar("My Scripts",list)

Gui.addWorkbench(ScriptWorkbench())
```

A typical module file

This is an example of a main module file, containing everything your module does. It is the `Scripts.py` file invoked by the previous example. You can have all your custom commands here.

```
import FreeCAD, FreeCADGui

class ScriptCmd:
    def Activated(self):
        # Here you write what your ScriptCmd does...
        FreeCAD.Console.PrintMessage('Hello, World!')
    def GetResources(self):
        return {'Pixmap' : 'path_to_an_icon/myicon.png', 'MenuText': 'Short text', 'ToolTip': '
More detailed text'}

FreeCADGui.addCommand('Script_Cmd', ScriptCmd())
```

Import a new filetype

Making an importer for a new filetype in FreeCAD is easy. FreeCAD doesn't consider that you import data in an opened document, but rather that you simply can directly open the new filetype. So what you need to do is to add the new file extension to FreeCAD's list of known extensions, and write the code that will read the file and create the FreeCAD objects you want:

This line must be added to the `InitGui.py` file to add the new file extension to the list:

```
# Assumes Import_Ext.py is the file that has the code for opening and reading .ext files
FreeCAD.addImportType("Your new File Type (*.ext)","Import_Ext")
```

Then in the `Import_Ext.py` file:

```
def open(filename):
    doc=App.newDocument()
    # here you do all what is needed with filename, read, classify data, create corresponding FreeCAD objects
    doc.recompute()
```

To export your document to some new filetype works the same way, except that you use:

```
FreeCAD.addExportType("Your new File Type (*.ext)","Export_Ext")
```

Adding a line

A line simply has 2 points.

```
import Part,PartGui
doc=App.activeDocument()
# add a line element to the document and set its points
l=Part.Line()
l.StartPoint=(0.0,0.0,0.0)
l.EndPoint=(1.0,1.0,1.0)
doc.addObject("Part::Feature","Line").Shape=l.toShape()
doc.recompute()
```

Adding a polygon

A polygon is simply a set of connected line segments (a polyline in AutoCAD). It doesn't need to be closed.

```
import Part,PartGui
doc=App.activeDocument()
n=list()
# create a 3D vector, set its coordinates and add it to the list
v=App.Vector(0,0,0)
n.append(v)
v=App.Vector(10,0,0)
n.append(v)
#... repeat for all nodes
# Create a polygon object and set its nodes
p=doc.addObject("Part::Polygon","Polygon")
p.Nodes=n
doc.recompute()
```

Adding and removing an object to a group

```
doc=App.activeDocument()
grp=doc.addObject("App::DocumentObjectGroup", "Group")
lin=doc.addObject("Part::Feature", "Line")
grp.addObject(lin) # adds the lin object to the group grp
grp.removeObject(lin) # removes the lin object from the group grp
```

Note: You can even add other groups to a group...

Adding a Mesh

```
import Mesh
doc=App.activeDocument()
# create a new empty mesh
m = Mesh.Mesh()
# build up box out of 12 facets
m.addFacet(0.0,0.0,0.0, 0.0,0.0,1.0, 0.0,1.0,1.0)
m.addFacet(0.0,0.0,0.0, 0.0,1.0,1.0, 0.0,1.0,0.0)
m.addFacet(0.0,0.0,0.0, 1.0,0.0,0.0, 1.0,0.0,1.0)
m.addFacet(0.0,0.0,0.0, 1.0,0.0,1.0, 0.0,0.0,1.0)
m.addFacet(0.0,0.0,0.0, 0.0,1.0,0.0, 1.0,1.0,0.0)
m.addFacet(0.0,0.0,0.0, 1.0,1.0,0.0, 1.0,0.0,0.0)
m.addFacet(0.0,1.0,0.0, 0.0,1.0,1.0, 1.0,1.0,1.0)
m.addFacet(0.0,1.0,0.0, 1.0,1.0,1.0, 1.0,1.0,0.0)
m.addFacet(0.0,1.0,1.0, 0.0,0.0,1.0, 1.0,0.0,1.0)
m.addFacet(0.0,1.0,1.0, 1.0,0.0,1.0, 1.0,1.0,1.0)
m.addFacet(1.0,1.0,0.0, 1.0,1.0,1.0, 1.0,0.0,1.0)
m.addFacet(1.0,1.0,0.0, 1.0,0.0,1.0, 1.0,0.0,0.0)
# scale to a edge length of 100
m.scale(100.0)
# add the mesh to the active document
me=doc.addObject("Mesh::Feature","Cube")
me.Mesh=m
```

Adding an arc or a circle

```
import Part
doc = App.activeDocument()
c = Part.Circle()
c.Radius=10.0
f = doc.addObject("Part::Feature", "Circle") # create a document with a circle feature
f.Shape = c.toShape() # Assign the circle shape to the shape property
doc.recompute()
```

Accessing and changing representation of an object

Each object in a FreeCAD document has an associated view representation object that stores all the parameters that define how the object appear, like color, linewidth, etc...

```
gad=Gui.activeDocument() # access the active document containing all
                          # view representations of the features in the
                          # corresponding App document

v=gad.getObject("Cube") # access the view representation to the Mesh feature 'Cube'
v.ShapeColor           # prints the color to the console
v.ShapeColor=(1.0,1.0,1.0) # sets the shape color to white
```

Observing mouse events in the 3D viewer via Python

The Inventor framework allows to add one or more callback nodes to the scenegraph of the viewer. By default in FreeCAD one callback node is installed per viewer which allows to add global or static C++ functions. In the appropriate Python binding some methods are provided to make use of this technique from within Python code.

```
App.newDocument()
v=Gui.activeDocument().activeView()

#This class logs any mouse button events. As the registered callback function fires twice for
'down' and
#'up' events we need a boolean flag to handle this.
class ViewObserver:
    def logPosition(self, info):
        down = (info["State"] == "DOWN")
        pos = info["Position"]
        if (down):
            FreeCAD.Console.PrintMessage("Clicked on position: (" +str(pos[0])+", "+str(pos[1])+
            ")\n")

o = ViewObserver()
c = v.addEventCallback("SoMouseButtonEvent",o.logPosition)
```

Now, pick somewhere on the area in the 3D viewer and observe the messages in the output window. To finish the observation just call

```
v.removeEventCallback("SoMouseButtonEvent",c)
```

The following event types are supported

- SoEvent -- all kind of events
- SoButtonEvent -- all mouse button and key events
- SoLocation2Event -- 2D movement events (normally mouse movements)
- SoMotion3Event -- 3D movement events (normally spaceball)
- SoKeyboardEvent -- key down and up events
- SoMouseButtonEvent -- mouse button down and up events
- SoSpaceballButtonEvent -- spaceball button down and up events

The Python function that can be registered with `addEventCallback()` expects a dictionary. Depending on the watched event the dictionary can contain different keys.

For all events it has the keys:

- Type -- the name of the event type i.e. SoMouseEvent, SoLocation2Event, ...

- Time -- the current time as string
- Position -- a tuple of two integers, mouse position
- ShiftDown -- a boolean, true if Shift was pressed otherwise false
- CtrlDown -- a boolean, true if Ctrl was pressed otherwise false
- AltDown -- a boolean, true if Alt was pressed otherwise false

For all button events, i.e. keyboard, mouse or spaceball events

- State -- A string 'UP' if the button was up, 'DOWN' if it was down or 'UNKNOWN' for all other cases

For keyboard events:

- Key -- a character of the pressed key

For mouse button event

- Button -- The pressed button, could be BUTTON1, ..., BUTTON5 or ANY

For spaceball events:

- Button -- The pressed button, could be BUTTON1, ..., BUTTON7 or ANY

And finally motion events:

- Translation -- a tuple of three floats
- Rotation -- a quaternion for the rotation, i.e. a tuple of four floats

Display keys pressed and Events command

This macro displays in the report view the keys pressed and all events command

```
App.newDocument()
v=Gui.activeDocument().activeView()
class ViewObserver:
    def logPosition(self, info):
        try:
            down = (info["Key"])
            FreeCAD.Console.PrintMessage(str(down)+"\n") # here the character pressed
            FreeCAD.Console.PrintMessage(str(info)+"\n") # list all events command
            FreeCAD.Console.PrintMessage("_____ "+"")
        except Exception:
            None

o = ViewObserver()
c = v.addEventCallback("SoEvent",o.logPosition)

#v.removeEventCallback("SoEvent",c) # remove ViewObserver
```

Manipulate the scenegraph in Python

It is also possible to get and change the scenegraph in Python, with the 'pivy' module -- a Python binding for Coin.

```
from pivy.coin import * # load the pivy module
view = Gui.ActiveDocument.ActiveView # get the active viewer
root = view.getSceneGraph() # the root is an SoSeparator node
root.addChild(SoCube())
view.fitAll()
```

The Python API of pivy is created by using the tool SWIG. As we use in FreeCAD some self-written nodes you cannot create them directly in Python. However, it is possible to create a node by its internal name. An instance of the type 'SoFCSelection' can be created with

```
type = SoType.fromName("SoFCSelection")
node = type.createInstance()
```

Adding and removing objects to/from the scenegraph

Adding new nodes to the scenegraph can be done this way. Take care of always adding a SoSeparator to contain the geometry, coordinates and material info of a same object. The following example adds a red line from (0,0,0) to (10,0,0):

```
from pivy import coin
sg = Gui.ActiveDocument.ActiveView.getSceneGraph()
co = coin.SoCoordinate3()
pts = [[0,0,0],[10,0,0]]
co.point.setValues(0,len(pts),pts)
ma = coin.SoBaseColor()
ma.rgb = (1,0,0)
li = coin.SoLineSet()
li.numVertices.setValue(2)
no = coin.SoSeparator()
no.addChild(co)
no.addChild(ma)
no.addChild(li)
sg.addChild(no)
```

To remove it, simply issue:

```
sg.removeChild(no)
```

Adding custom widgets to the interface

You can create custom widgets with Qt designer, transform them into a python script, and then load them into the FreeCAD interface with PyQt4.

The python code produced by the Ui python compiler (the tool that converts qt-designer .ui files into python code) generally looks like this (it is simple, you can also code it directly in python):

```
class myWidget_Ui(object):
    def setupUi(self, myWidget):
        myWidget.setObjectName("my Nice New Widget")
        myWidget.resize(QtCore.QSize(QtCore.QRect(0,0,300,100).size()).expandedTo(myWidget.minimumSizeHint())) # sets size of the widget

        self.label = QtGui.QLabel(myWidget) # creates a label
        self.label.setGeometry(QtCore.QRect(50,50,200,24)) # sets its size
        self.label.setObjectName("label") # sets its name, so it can be found by name

    def retranslateUi(self, draftToolBar): # built-in QT function that manages translations of widgets
        myWidget.setWindowTitle(QtGui.QApplication.translate("myWidget", "My Widget", None, QtGui.QApplication.UnicodeUTF8))
        self.label.setText(QtGui.QApplication.translate("myWidget", "Welcome to my new widget!", None, QtGui.QApplication.UnicodeUTF8))
```

Then, all you need to do is to create a reference to the FreeCAD Qt window, insert a custom widget into it, and "transform" this widget into yours with the Ui code we just made:

```
app = QtGui.QApp
FCmw = app.activeWindow() # the active qt window, = the freecad window since we are inside it
myNewFreeCADWidget = QtGui.QDockWidget() # create a new dckwidget
myNewFreeCADWidget.ui = myWidget_Ui() # load the Ui script
myNewFreeCADWidget.ui.setupUi(myNewFreeCADWidget) # setup the ui
FCmw.addDockWidget(QtCore.Qt.RightDockWidgetArea,myNewFreeCADWidget) # add the widget to the main window
```

Adding a Tab to the Combo View

The following code allows you to add a tab to the FreeCAD ComboView, besides the "Project" and "Tasks" tabs. It also uses the uic module to load an ui file directly in that tab.

```

# create new Tab in ComboView
from PySide import QtGui,QtCore
#from PySide import uic

def getMainWindow():
    "returns the main window"
    # using QtGui.qApp.activeWindow() isn't very reliable because if another
    # widget than the mainwindow is active (e.g. a dialog) the wrong widget is
    # returned
    toplevel = QtGui.qApp.topLevelWidgets()
    for i in toplevel:
        if i.metaObject().className() == "Gui::MainWindow":
            return i
    raise Exception("No main window found")

def getComboView(mw):
    dw=mw.findChildren(QtGui.QDockWidget)
    for i in dw:
        if str(i.objectName()) == "Combo View":
            return i.findChild(QtGui.QTabWidget)
        elif str(i.objectName()) == "Python Console":
            return i.findChild(QtGui.QTabWidget)
    raise Exception ("No tab widget found")

mw = getMainWindow()
tab = getComboView(getMainWindow())
tab2=QtGui.QDialog()
tab.addTab(tab2,"A Special Tab")

#uic.loadUi("/myTaskPanelforTabs.ui",tab2)
tab2.show()
#tab.removeTab(2)

```

Enable or disable a window

```

from PySide import QtGui
mw=FreeCADGui.getMainWindow()
dws=mw.findChildren(QtGui.QDockWidget)

# objectName may be :
# "Report view"
# "Tree view"
# "Property view"
# "Selection view"
# "Combo View"
# "Python console"
# "draftToolbar"

for i in dws:
    if i.objectName() == "Report view":
        dw=i
        break

va=dw.toggleViewAction()
va.setChecked(True)      # True or False
dw.setVisible(True)     # True or False

```

Opening a custom webpage

```

import WebGui
WebGui.openBrowser("http://www.example.com")

```

Getting the HTML contents of an opened webpage

```

from PyQt4 import QtGui,QtWebKit
a = QtGui.qApp
mw = a.activeWindow()
v = mw.findChild(QtWebKit.QWebFrame)
html = unicode(v.toHtml())
print html

```

Retrieve and use the coordinates of 3 selected points or objects

```

# -*- coding: utf-8 -*-
# the line above to put the accentuated in the remarks
# If this line is missing, an error will be returned
# extract and use the coordinates of 3 objects selected
import Part, FreeCAD, math, PartGui, FreeCADGui
from FreeCAD import Base, Console
sel = FreeCADGui.Selection.getSelection() # " sel " contains the items selected
if len(sel)!=3 :
    # If there are no 3 objects selected, an error is displayed in the report view
    # The \r and \n at the end of line mean return and the newline CR + LF.
    Console.PrintError("Select 3 points exactly\r\n")
else :
    points=[]
    for obj in sel:
        points.append(obj.Shape.BoundingBox.Center)

    for pt in points:
        # display of the coordinates in the report view
        Console.PrintMessage(str(pt.x)+"\r\n")
        Console.PrintMessage(str(pt.y)+"\r\n")
        Console.PrintMessage(str(pt.z)+"\r\n")

    Console.PrintMessage(str(pt[1]) + "\r\n")

```

List all objects

```

# -*- coding: utf-8 -*-
import FreeCAD,Draft
# List all objects of the document
doc = FreeCAD.ActiveDocument
objs = FreeCAD.ActiveDocument.Objects
#App.Console.PrintMessage(str(objs) + "\n")
#App.Console.PrintMessage(str(len(FreeCAD.ActiveDocument.Objects)) + " Objects" + "\n")

for obj in objs:
    a = obj.Name # list the Name of the object (
not modifiable)
    b = obj.Label # list the Label of the object (
modifiable)
    try:
        c = obj.LabelText # list the LabeText of the text (
modifiable)
        App.Console.PrintMessage(str(a) + " " + str(b) + " " + str(c) + "\n") # Displays the Name
the Label and the text
    except:
        App.Console.PrintMessage(str(a) + " " + str(b) + "\n") # Displays the Name and the Label
of the object

#doc.removeObject("Box") # Clears the designated object

```

List the dimension give the name of object

```

for edge in FreeCAD.ActiveDocument.MyObjectName.Shape.Edges: # replace "MyObjectName" for list
print edge.Length

```

Function resident with the mouse click action

Here with **SelObserver** on a object select

```

# -*- coding: utf-8 -*-
# causes an action to the mouse click on an object
# This function remains resident (in memory) with the function "addObserver(s)"
# "removeObserver(s) # Uninstalls the resident function
class SelObserver:
    def addSelection(self,doc,obj,sub,pnt):          # Selection object
    #def setPreselection(self,doc,obj,sub):         # Preselection object
        App.Console.PrintMessage("addSelection"+ "\n")
        App.Console.PrintMessage(str(doc)+ "\n")   # Name of the document
        App.Console.PrintMessage(str(obj)+ "\n")   # Name of the object
        App.Console.PrintMessage(str(sub)+ "\n")   # The part of the object name
        App.Console.PrintMessage(str(pnt)+ "\n")   # Coordinates of the object
        App.Console.PrintMessage("_____"+" "\n")

    def removeSelection(self,doc,obj,sub):          # Delete the selected object
        App.Console.PrintMessage("removeSelection"+ "\n")
    def setSelection(self,doc):                    # Selection in ComboView
        App.Console.PrintMessage("setSelection"+ "\n")
    def clearSelection(self,doc):                  # If click on the screen, clear the
selection
        App.Console.PrintMessage("clearSelection"+ "\n") # If click on another object, clear
the previous object
s =SelObserver()
FreeCADGui.Selection.addObserver(s)              # install the function mode resident
#FreeCADGui.Selection.removeObserver(s)          # Uninstall the resident function

```

Other example with **ViewObserver** on a object select or view

```

App.newDocument()
v=Gui.activeDocument().activeView()

#This class logs any mouse button events. As the registered callback function fires twice for
'down' and
#'up' events we need a boolean flag to handle this.
class ViewObserver:
    def __init__(self, view):
        self.view = view

    def logPosition(self, info):
        down = (info["State"] == "DOWN")
        pos = info["Position"]
        if (down):
            FreeCAD.Console.PrintMessage("Clicked on position: (" +str(pos[0])+", "+str(pos[1])+
")\n")
            pnt = self.view.getPoint(pos)
            FreeCAD.Console.PrintMessage("World coordinates: " + str(pnt) + "\n")
            info = self.view.getObjectInfo(pos)
            FreeCAD.Console.PrintMessage("Object info: " + str(info) + "\n")

o = ViewObserver(v)
c = v.addEventCallback("SoMouseButtonEvent",o.logPosition)

```

List the components of an object

```

# -*- coding: utf-8 -*-
# This function list the components of an object
# and extract this object its XYZ coordinates,
# its edges and their lengths center of mass and coordinates
# its faces and their center of mass
# its faces and their surfaces and coordinates
# 8/05/2014

import Draft,Part
def detail():
    sel = FreeCADGui.Selection.getSelection() # Select an object
    if len(sel) != 0: # If there is a selection then
        Vertx=[]
        Edges=[]
        Faces=[]
        compt_V=0
        compt_E=0
        compt_F=0
        pas =0
        perimetre = 0.0
        EdgesLong = []

        # Displays the "Name" and the "Label" of the selection
        App.Console.PrintMessage("Selection > " + str(sel[0].Name) + " " + str(sel[0].Label)
+"\\n"+"\\n")

        for j in enumerate(sel[0].Shape.Edges): # Search t
            he "Edges" and their lengths
            compt_E+=1
            Edges.append("Edge%d" % (j[0]+1))
            EdgesLong.append(str(sel[0].Shape.Edges[compt_E-1].Length))
            perimetre += (sel[0].Shape.Edges[compt_E-1].Length) # calculat
            es the perimeter

            # Displays the "Edge" and its length
            App.Console.PrintMessage("Edge"+str(compt_E)+" Length > "+str(sel[0].Shape.Edges[c
            ompt_E-1].Length)+"\\n")

            # Displays the "Edge" and its center mass
            App.Console.PrintMessage("Edge"+str(compt_E)+" Center > "+str(sel[0].Shape.Edges[c
            ompt_E-1].CenterOfMass)+"\\n")

            num = sel[0].Shape.Edges[compt_E-1].Vertexes[0]
            Vertx.append("X1: "+str(num.Point.x))
            Vertx.append("Y1: "+str(num.Point.y))
            Vertx.append("Z1: "+str(num.Point.z))
            # Displays the coordinates 1
            App.Console.PrintMessage("X1: "+str(num.Point[0])+" Y1: "+str(num.Point[1])+" Z1:
            "+str(num.Point[2])+"\\n")

            try:
                num = sel[0].Shape.Edges[compt_E-1].Vertexes[1]
                Vertx.append("X2: "+str(num.Point.x))
                Vertx.append("Y2: "+str(num.Point.y))
                Vertx.append("Z2: "+str(num.Point.z))
            except:
                Vertx.append("-")
                Vertx.append("-")
                Vertx.append("-")
            # Displays the coordinates 2
            App.Console.PrintMessage("X2: "+str(num.Point[0])+" Y2: "+str(num.Point[1])+" Z2:
            "+str(num.Point[2])+"\\n")

            App.Console.PrintMessage("\\n")
            App.Console.PrintMessage("Perimeter of the form : "+str(perimetre)+"\\n")

            App.Console.PrintMessage("\\n")
            FacesSurf = []
            for j in enumerate(sel[0].Shape.Faces): # Search
            the "Faces" and their surface
            compt_F+=1
            Faces.append("Face%d" % (j[0]+1))
            FacesSurf.append(str(sel[0].Shape.Faces[compt_F-1].Area))

            # Displays 'Face' and its surface
            App.Console.PrintMessage("Face"+str(compt_F)+" > Surface "+str(sel[0].Shape.Faces
            [compt_F-1].Area)+"\\n")

            # Displays 'Face' and its CenterOfMass
            App.Console.PrintMessage("Face"+str(compt_F)+" > Center "+str(sel[0].Shape.Faces
            [compt_F-1].CenterOfMass)+"\\n")

            # Displays 'Face' and its Coordinates
            FacesCoor = []
            fco = 0
            for f0 in sel[0].Shape.Faces[compt_F-1].Vertexes: # Search
            the Vertexes of the face

```

```

        fco += 1
        FacesCoor.append("X"+str(fco)+": "+str(f0.Point.x))
        FacesCoor.append("Y"+str(fco)+": "+str(f0.Point.y))
        FacesCoor.append("Z"+str(fco)+": "+str(f0.Point.z))

        # Displays 'Face' and its Coordinates
        App.Console.PrintMessage("Face"+str(compt_F)+" > Coordinate"+str(FacesCoor)+"\n")

        # Displays 'Face' and its Volume
        App.Console.PrintMessage("Face"+str(compt_F)+" > Volume "+str(sel[0].Shape.Faces
[compt_F-1].Volume)+"\n")
        App.Console.PrintMessage("\n")

        # Displays the total surface of the form
        App.Console.PrintMessage("Surface of the form : "+str(sel[0].Shape.Area)+"\n")

        # Displays the total Volume of the form
        App.Console.PrintMessage("Volume of the form : "+str(sel[0].Shape.Volume)+"\n")

detail()

```

List the PropertiesList

```

import FreeCADGui
from FreeCAD import Console
o = App.ActiveDocument.ActiveObject
op = o.PropertiesList
for p in op:
    Console.PrintMessage("Property: "+ str(p)+ " Value: " + str(o.getPropertyByName(p))+"\r\n"
)

```

Adding one Property "Comment"

```

import Draft
obj = FreeCADGui.Selection.getSelection()[0]
obj.addProperty("App::PropertyString", "GComment", "Draft", "Font name").GComment = "Comment here
"
App.activeDocument().recompute()

```

Search and data extraction

Examples of research and decoding information on an object.

Each section is independently and is separated by "#####" can be copied directly into the Python console, or in a macro or use this macro. The description of the macro in the commentary.

Displaying it in the "View Report" window (View > Views > View report)

```

# -*- coding: utf-8 -*-
from __future__ import unicode_literals

# Exemples de recherche et de decodage d'informations sur un objet
# Chaque section peut etre copiee directement dans la console Python ou dans une macro ou utilisez la macro tel quel
# Certaines commandes se repetent seul l'approche est differente
# L'affichage se fait dans la Vue rapport : Menu Affichage > Vues > Vue rapport
#
# Examples of research and decoding information on an object
# Each section can be copied directly into the Python console, or in a macro or uses this macro
# Certain commands as repeat alone approach is different
# Displayed on Report view : Menu View > Views > report view
#
# rev:30/08/2014:29/09/2014:17/09/2015

from FreeCAD import Base
import DraftVecUtils, Draft, Part

mydoc = FreeCAD.activeDocument().Name # Name of active Document
App.Console.PrintMessage("Active docu : "+(mydoc)+"\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
object_Label = sel[0].Label # Label of the object (modifiable)
App.Console.PrintMessage("object_Label : "+(object_Label)+"\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
App.Console.PrintMessage("sel : "+str(sel[0])+"\n\n") # sel[0] first object selected
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
object_Name = sel[0].Name # Name of the object (not modifiable)
App.Console.PrintMessage("object_Name : "+str(object_Name)+"\n\n")
#####

try:
    SubElement = FreeCADGui.Selection.getSelectionEx() # sub element name with
    getSelectionEx()
    element_ = SubElement[0].SubElementNames[0] # name of 1 element selected
    App.Console.PrintMessage("elementSel : "+str(element_)+"\n\n")
except:
    App.Console.PrintMessage("Oops"+ "\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
App.Console.PrintMessage("sel : "+str(sel[0])+"\n\n") # sel[0] first object selected
#####

SubElement = FreeCADGui.Selection.getSelectionEx() # sub element name with
getSelectionEx()
App.Console.PrintMessage("SubElement : "+str(SubElement[0])+"\n\n") # name of sub element
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
i = 0
for j in enumerate(sel[0].Shape.Edges): # list all Edges
    i += 1
    App.Console.PrintMessage("Edges n : "+str(i)+"\n")
    a = sel[0].Shape.Edges[j[0]].Vertexes[0]
    App.Console.PrintMessage("X1 : "+str(a.Point.x)+"\n") # coordinate XYZ first point
    App.Console.PrintMessage("Y1 : "+str(a.Point.y)+"\n")
    App.Console.PrintMessage("Z1 : "+str(a.Point.z)+"\n")
    try:
        a = sel[0].Shape.Edges[j[0]].Vertexes[1]
        App.Console.PrintMessage("X2 : "+str(a.Point.x)+"\n") # coordinate XYZ second point
        App.Console.PrintMessage("Y2 : "+str(a.Point.y)+"\n")
        App.Console.PrintMessage("Z2 : "+str(a.Point.z)+"\n")
    except:
        App.Console.PrintMessage("Oops"+ "\n\n")
App.Console.PrintMessage("\n")

```

```
#####
try:
    SubElement = FreeCADGui.Selection.getSelectionEx()
    # sub element name with getSelectionEx()
    subElementName = Gui.Selection.getSelectionEx()[0].SubElementNames[0]
    # sub element name with getSelectionEx()
    App.Console.PrintMessage("subElementName : "+str(subElementName)+"\n\n")

    subObjectLength = Gui.Selection.getSelectionEx()[0].SubObjects[0].Length
    # sub element Length
    App.Console.PrintMessage("subObjectLength: "+str(subObjectLength)+"\n\n")

    subObjectX1 = Gui.Selection.getSelectionEx()[0].SubObjects[0].Vertexes[0].Point.x
    # sub element coordinate X1
    App.Console.PrintMessage("subObject_X1 : "+str(subObjectX1)+"\n")
    subObjectY1 = Gui.Selection.getSelectionEx()[0].SubObjects[0].Vertexes[0].Point.y
    # sub element coordinate Y1
    App.Console.PrintMessage("subObject_Y1 : "+str(subObjectY1)+"\n")
    subObjectZ1 = Gui.Selection.getSelectionEx()[0].SubObjects[0].Vertexes[0].Point.z
    # sub element coordinate Z1
    App.Console.PrintMessage("subObject_Z1 : "+str(subObjectZ1)+"\n\n")

    subObjectX2 = Gui.Selection.getSelectionEx()[0].SubObjects[0].Vertexes[1].Point.x
    # sub element coordinate X2
    App.Console.PrintMessage("subObject_X2 : "+str(subObjectX2)+"\n")
    subObjectY2 = Gui.Selection.getSelectionEx()[0].SubObjects[0].Vertexes[1].Point.y
    # sub element coordinate Y2
    App.Console.PrintMessage("subObject_Y2 : "+str(subObjectY2)+"\n")
    subObjectZ2 = Gui.Selection.getSelectionEx()[0].SubObjects[0].Vertexes[1].Point.z
    # sub element coordinate Z2
    App.Console.PrintMessage("subObject_Z2 : "+str(subObjectZ2)+"\n\n")

    subObjectBoundingBox = Gui.Selection.getSelectionEx()[0].SubObjects[0].BoundingBox
    # sub element BoundingBox coordinates
    App.Console.PrintMessage("subObjectBBBox : "+str(subObjectBoundingBox)+"\n")

    subObjectBoundingBoxCenter = Gui.Selection.getSelectionEx()[0].SubObjects[0].BoundingBox.Center
    # sub element BoundingBoxCenter
    App.Console.PrintMessage("subObjectBBBoxCe: "+str(subObjectBoundingBoxCenter)+"\n")

    surfaceFace = Gui.Selection.getSelectionEx()[0].SubObjects[0].Area
    # Area of the face selected
    App.Console.PrintMessage("surfaceFace : "+str(surfaceFace)+"\n\n")
except:
    App.Console.PrintMessage("Oops "+str("\n\n"))
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
surface = sel[0].Shape.Area # Area object complete
App.Console.PrintMessage("surfaceObject : "+str(surface)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
CenterOfMass = sel[0].Shape.CenterOfMass # Center of Mass of
the object
App.Console.PrintMessage("CenterOfMass : "+str(CenterOfMass)+"\n")
App.Console.PrintMessage("CenterOfMassX : "+str(CenterOfMass[0])+"\n") # coordinates [0]=X
[1]=Y [2]=Z
App.Console.PrintMessage("CenterOfMassY : "+str(CenterOfMass[1])+"\n")
App.Console.PrintMessage("CenterOfMassZ : "+str(CenterOfMass[2])+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
for j in enumerate(sel[0].Shape.Faces): # List all faces of
the object
    App.Console.PrintMessage("Face : "+str("Face%d" % (j[0]+1))+"\n")
App.Console.PrintMessage("\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
volume_ = sel[0].Shape.Volume # Volume of the object
App.Console.PrintMessage("volume_ : "+str(volume_)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
boundingBox_ = sel[0].Shape.BoundingBox # BoundingBox of the object
App.Console.PrintMessage("boundingBox_ : "+str(boundingBox_)+"\n")

```

```

boundBoxLX = boundBox_.XLength          # Length x boundBox
rectangle
boundBoxLY = boundBox_.YLength          # Length y boundBox
rectangle
boundBoxLZ = boundBox_.ZLength          # Length z boundBox
rectangle
boundBoxDiag= boundBox_.DiagonalLength  # Diagonal Length bo
undBox rectangle

App.Console.PrintMessage("boundBoxLX   : "+str(boundBoxLX)+"\n")
App.Console.PrintMessage("boundBoxLY   : "+str(boundBoxLY)+"\n")
App.Console.PrintMessage("boundBoxLZ   : "+str(boundBoxLZ)+"\n")
App.Console.PrintMessage("boundBoxDiag : "+str(boundBoxDiag)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
pl = sel[0].Shape.Placement              # Placement Vector X
YZ and Yaw-Pitch-Roll
App.Console.PrintMessage("Placement    : "+str(pl)+"\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
pl = sel[0].Shape.Placement.Base         # Placement Vector X
YZ
App.Console.PrintMessage("PlacementBase : "+str(pl)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
Yaw = sel[0].Shape.Placement.Rotation.toEuler()[0] # decode angle Euler
Yaw
App.Console.PrintMessage("Yaw          : "+str(Yaw)+"\n")
Pitch = sel[0].Shape.Placement.Rotation.toEuler()[1] # decode angle Euler
Pitch
App.Console.PrintMessage("Pitch        : "+str(Pitch)+"\n")
Roll = sel[0].Shape.Placement.Rotation.toEuler()[2] # decode angle Euler
Yaw
App.Console.PrintMessage("Roll         : "+str(Roll)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
oripl_X = sel[0].Placement.Base[0]       # decode Placement X
oripl_Y = sel[0].Placement.Base[1]       # decode Placement Y
oripl_Z = sel[0].Placement.Base[2]       # decode Placement Z

App.Console.PrintMessage("oripl_X     : "+str(oripl_X)+"\n")
App.Console.PrintMessage("oripl_Y     : "+str(oripl_Y)+"\n")
App.Console.PrintMessage("oripl_Z     : "+str(oripl_Z)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
rotation = sel[0].Placement.Rotation     # decode Placement R
otation
App.Console.PrintMessage("rotation     : "+str(rotation)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
pl = sel[0].Shape.Placement.Rotation     # decode Placement R
otation other method
App.Console.PrintMessage("Placement Rot : "+str(pl)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
pl = sel[0].Shape.Placement.Rotation.Angle # decode Placement R
otation Angle
App.Console.PrintMessage("Placement Rot Angle : "+str(pl)+"\n\n")
#####

sel = FreeCADGui.Selection.getSelection() # select object with
getSelection()
Rot_0 = sel[0].Placement.Rotation.Q[0]   # decode Placement R
otation 0
App.Console.PrintMessage("Rot_0       : "+str(Rot_0)+ " rad , "+str(180 * Rot_0 / 3.1416)+"
deg "+"")

Rot_1 = sel[0].Placement.Rotation.Q[1]   # decode Placement R
otation 1
App.Console.PrintMessage("Rot_1       : "+str(Rot_1)+ " rad , "+str(180 * Rot_1 / 3.1416)+"
deg "+"")

Rot_2 = sel[0].Placement.Rotation.Q[2]   # decode Placement R

```

```

otation 2
App.Console.PrintMessage("Rot_2      : "+str(Rot_2)+ " rad ,  "+str(180 * Rot_2 / 3.1416)+"
deg "+"\\n")

Rot_3 = sel[0].Placement.Rotation.Q[3]                                # decode Placement R
otation 3
App.Console.PrintMessage("Rot_3      : "+str(Rot_3)+"\\n\\n")
#####

```

Manual search of an element with label

```

# Extract the coordinate X,Y,Z and Angle giving the label
App.Console.PrintMessage("Base.x      : "+str(FreeCAD.ActiveDocument.getObjectsByLabel("Cylind
re")[0].Placement.Base.x)+"\\n")
App.Console.PrintMessage("Base.y      : "+str(FreeCAD.ActiveDocument.getObjectsByLabel("Cylind
re")[0].Placement.Base.y)+"\\n")
App.Console.PrintMessage("Base.z      : "+str(FreeCAD.ActiveDocument.getObjectsByLabel("Cylind
re")[0].Placement.Base.z)+"\\n")
App.Console.PrintMessage("Base.Angle  : "+str(FreeCAD.ActiveDocument.getObjectsByLabel("Cylind
re")[0].Placement.Rotation.Angle)+"\\n\\n")
#####

```

PS: Usually the angles are given in Radian to convert :

1. angle in Degrees to Radians :

- Angle in radian = $\pi * (\text{angle in degree}) / 180$
- Angle in radian = `math.radians(angle in degree)`

2. angle in Radians to Degrees :

- Angle in degree = $180 * (\text{angle in radian}) / \pi$
- Angle in degree = `math.degrees(angle in radian)`

Cartesian coordinates

This code displays the Cartesian coordinates of the selected item.

Change the value of "numberOfPoints" if you want a different number of points (precision)

```

numberOfPoints = 100                                             # Decomposition n
umber (or precision you can change)
selectedEdge = FreeCADGui.Selection.getSelectionEx()[0].SubObjects[0].copy() # select one elem
ent
points = selectedEdge.discretize(numberOfPoints)                # discretize the
element
i=0
for p in points:                                                # list and displa
y the coordinates
    i+=1
    print i, " X", p.x, " Y", p.y, " Z", p.z

```

Other method display on "Int" and "Float"

```

import Part
from FreeCAD import Base

c=Part.makeCylinder(2,10)      # create the circle
Part.show(c)                  # display the shape

# slice accepts two arguments:
#+ the normal of the cross section plane
#+ the distance from the origin to the cross section plane. Here you have to find a value so t
hat the plane intersects your object
s=c.slice(Base.Vector(0,1,0),0) #

# here the result is a single wire
# depending on the source object this can be several wires
s=s[0]

# if you only need the vertexes of the shape you can use
v=[]
for i in s.Vertexes:
    v.append(i.Point)

# but you can also sub-sample the section to have a certain number of points (int) ...
p1=s.discretize(20)
ii=0
for i in p1:
    ii+=1
    print i                                # Vector()
    print ii, ": X:", i.x, " Y:", i.y, " Z:", i.z    # Vector decode
Draft.makeWire(p1,closed=False,face=False,support=None) # to see the difference accuracy (20)

## uncomment to use
#import Draft
#Draft.downgrade(App.ActiveDocument.ActiveObject,delete=True) # first transform the DWire in
Wire "downgrade"
#Draft.downgrade(App.ActiveDocument.ActiveObject,delete=True) # second split the Wire in sing
le objects "downgrade"
#
##Draft.upgrade(FreeCADGui.Selection.getSelection(),delete=True) # to attach lines contiguous
SELECTED use "upgrade"

# ... or define a sampling distance (float)
p2=s.discretize(0.5)
ii=0
for i in p2:
    ii+=1
    print i                                # Vector()
    print ii, ": X:", i.x, " Y:", i.y, " Z:", i.z    # Vector decode
Draft.makeWire(p2,closed=False,face=False,support=None) # to see the difference accuracy (0.5
)

## uncomment to use
#import Draft
#Draft.downgrade(App.ActiveDocument.ActiveObject,delete=True) # first transform the DWire in
Wire "downgrade"
#Draft.downgrade(App.ActiveDocument.ActiveObject,delete=True) # second split the Wire in sing
le objects "downgrade"
#
##Draft.upgrade(FreeCADGui.Selection.getSelection(),delete=True) # to attach lines contiguous
SELECTED use "upgrade"

```

Select all objects in the document

```

import FreeCAD
for obj in FreeCAD.ActiveDocument.Objects:
    print obj.Name                    # display the object Name
    objName = obj.Name
    obj = App.ActiveDocument.getObject(objName)
    Gui.Selection.addSelection(obj)    # select the object

```

Selecting a face of an object

```

# select one face of the object
import FreeCAD, Draft
App=FreeCAD
nameObject = "Box"                    # objet
faceSelect = "Face3"                  # face to selection
loch=App.ActiveDocument.getObject(nameObject) # objet
Gui.Selection.clearSelection()         # clear all selection
Gui.Selection.addSelection(loch,faceSelect) # select the face specified
s = Gui.Selection.getSelectionEx()
#Draft.makeFacebinder(s)              #

```

Create one object to the position of the Camera

```

# create one object of the position to camera with "getCameraOrientation()"
# the object is still facing the screen
import Draft

plan = FreeCADGui.ActiveDocument.ActiveView.getCameraOrientation()
plan = str(plan)
##### extract data
a = ""
for i in plan:
    if i in ("0123456789e.- "):
        a+=i
a = a.strip(" ")
a = a.split(" ")
##### extract data

#print a
#print a[0]
#print a[1]
#print a[2]
#print a[3]

xP = float(a[0])
yP = float(a[1])
zP = float(a[2])
qP = float(a[3])

pl = FreeCAD.Placement()
pl.Rotation.Q = (xP,yP,zP,qP) # rotation of object
pl.Base = FreeCAD.Vector(0.0,0.0,0.0) # here coordinates XYZ of Object
rec = Draft.makeRectangle(length=10.0,height=10.0,placement=pl,face=False,support=None) # create rectangle
#rec = Draft.makeCircle(radius=5,placement=pl,face=False,support=None) # create circle
print rec.Name

```

here same code simplified

```

import Draft
pl = FreeCAD.Placement()
pl.Rotation = FreeCADGui.ActiveDocument.ActiveView.getCameraOrientation()
pl.Base = FreeCAD.Vector(0.0,0.0,0.0)
rec = Draft.makeRectangle(length=10.0,height=10.0,placement=pl,face=False,support=None)

```

Find normal vector on the surface

This example show how to find normal vector on the surface by find the u,v parameters of one point on the surface and use u,v parameters to find normal vector

```

def normal(self):
    ss=FreeCADGui.Selection.getSelectionEx()[0].SubObjects[0].copy()#SubObjects[0] is the edge
    list
    points = ss.discretize(3.0)#points on the surface edge,
        #this example just use points on the edge for example.
        #However point is not necessary on the edge, it can be anywhere on the surface.
    face=FreeCADGui.Selection.getSelectionEx()[0].SubObjects[1]
    for pp in points:
        pt=FreeCAD.Base.Vector(pp.x,pp.y,pp.z)#a point on the surface edge
        uv=face.Surface.parameter(pt)# find the surface u,v parameter of a point on the surface
    edge
        u=uv[0]
        v=uv[1]
        normal=face.normalAt(u,v)#use u,v to find normal vector
        print normal
        line=Part.makeLine((pp.x,pp.y,pp.z), (normal.x,normal.y,normal.z))
        Part.show(line)

```

< [previous: Embedding FreeCAD \(/wiki/index.php?title=Embedding_FreeCAD\)](#)

[next: Line drawing function > \(/wiki/index.php?title=Line_drawing_function\)](#)

[Index \(/wiki/index.php?title=Online_Help_Toc\)](#)

< translate> This page shows how advanced functionality can easily be built in Python. In this exercise, we will be building a new tool that draws a line.

This tool can then be linked to a FreeCAD command, and that command can be called by any element of the interface, like a menu item or a toolbar button.

The main script

First we will write a script containing all our functionality. Then, we will save this in a file, and import it in FreeCAD, so all classes and functions we write will be available to FreeCAD. So, launch your favorite text editor, and type the following lines:< /translate>

```
import FreeCADGui, Part
from pivy.coin import *

class line:
    "this class will create a line after the user clicked 2 points on the screen"
    def __init__(self):
        self.view = FreeCADGui.ActiveDocument.ActiveView
        self.stack = []
        self.callback = self.view.addEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.getpoint)

    def getpoint(self,event_cb):
        event = event_cb.getEvent()
        if event.getState() == SoMouseButtonEvent.DOWN:
            pos = event.getPosition()
            point = self.view.getPoint(pos[0],pos[1])
            self.stack.append(point)
            if len(self.stack) == 2:
                l = Part.Line(self.stack[0],self.stack[1])
                shape = l.toShape()
                Part.show(shape)
                self.view.removeEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.callback)
```

<translate>

Detailed explanation

</translate>

```
import Part, FreeCADGui
from pivy.coin import *
```

<translate> In Python, when you want to use functions from another module, you need to import it. In our case, we will need functions from the Part Module (/wiki/index.php?title=Part_Module), for creating the line, and from the Gui module (FreeCADGui), for accessing the 3D view. We also need the complete contents of the coin library, so we can use directly all coin objects like SoMouseButtonEvent, etc...< /translate>

```
class line:
```

<translate> Here we define our main class. Why do we use a class and not a function? The reason is that we need our tool to stay "alive" while we are waiting for the user to click on the screen. A function ends when its task has been done, but an object (a class defines an object) stays alive until it is destroyed.< /translate>

```
"this class will create a line after the user clicked 2 points on the screen"
```

<translate> In Python, every class or function can have a description string. This is particularly useful in FreeCAD, because when you'll call that class in the interpreter, the description string will be displayed as a tooltip.< /translate>

```
def __init__(self):
```

<translate> Python classes can always contain an __init__ function, which is executed when the class is called to create an object. So, we will put here everything we want to happen when our line tool begins.< /translate>

```
self.view = FreeCADGui.ActiveDocument.ActiveView
```

<translate> In a class, you usually want to append *self*. before a variable name, so it will be easily accessible to all functions inside and outside that class. Here, we will use *self.view* to access and manipulate the active 3D view. </translate>

```
self.stack = []
```

<translate> Here we create an empty list that will contain the 3D points sent by the *getpoint* function. </translate>

```
self.callback = self.view.addEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.getpoint)
```

<translate> This is the important part: Since it is actually a coin3D (<http://www.coin3d.org/>) scene, the FreeCAD uses coin callback mechanism, that allows a function to be called everytime a certain scene event happens. In our case, we are creating a callback for *SoMouseButtonEvent* (http://doc.coin3d.org/Coin/group__events.html) events, and we bind it to the *getpoint* function. Now, everytime a mouse button is pressed or released, the *getpoint* function will be executed.

Note that there is also an alternative to *addEventCallbackPivy()* called *addEventCallback()* which dispenses the use of *pivy*. But since *pivy* is a very efficient and natural way to access any part of the coin scene, it is much better to use it as much as you can! </translate>

```
def getpoint(self,event_cb):
```

<translate> Now we define the *getpoint* function, that will be executed when a mouse button is pressed in a 3D view. This function will receive an argument, that we will call *event_cb*. From this event callback we can access the event object, which contains several pieces of information (mode info [here](/wiki/index.php?title=Code_snippets#Observing_mouse_events_in_the_3D_viewer_via_Python)). </translate>

```
if event.getState() == SoMouseButtonEvent.DOWN:
```

<translate> The *getpoint* function will be called when a mouse button is pressed or released. But we want to pick a 3D point only when pressed (otherwise we would get two 3D points very close to each other). So we must check for that here. </translate>

```
pos = event.getPosition()
```

<translate> Here we get the screen coordinates of the mouse cursor</translate>

```
point = self.view.getPoint(pos[0],pos[1])
```

<translate> This function gives us a FreeCAD vector (*x,y,z*) containing the 3D point that lies on the focal plane, just under our mouse cursor. If you are in camera view, imagine a ray coming from the camera, passing through the mouse cursor, and hitting the focal plane. There is our 3D point. If we are in orthogonal view, the ray is parallel to the view direction.</translate>

```
self.stack.append(point)
```

<translate> We add our new point to the stack</translate>

```
if len(self.stack) == 2:
```

<translate> Do we have enough points already? if yes, then let's draw the line!</translate>

```
l = Part.Line(self.stack[0],self.stack[1])
```

<translate> Here we use the function *Line()* from the *Part* Module (/wiki/index.php?title=Part_Module) that creates a line from two FreeCAD vectors. Everything we create and modify inside the *Part* module, stays in

the Part module. So, until now, we created a Line Part. It is not bound to any object of our active document, so nothing appears on the screen.< /translate>

```
shape = l.toShape()
```

<translate> The FreeCAD document can only accept shapes from the Part module. Shapes are the most generic type of the Part module. So, we must convert our line to a shape before adding it to the document.< /translate>

```
Part.show(shape)
```

<translate> The Part module has a very handy show() function that creates a new object in the document and binds a shape to it. We could also have created a new object in the document first, then bound the shape to it manually.< /translate>

```
self.view.removeEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.callback)
```

<translate> Since we are done with our line, let's remove the callback mechanism, that consumes precious CPU cycles.

Testing & Using the script

Now, let's save our script to some place where the FreeCAD python interpreter will find it. When importing modules, the interpreter will look in the following places: the python installation paths, the FreeCAD bin directory, and all FreeCAD modules directories. So, the best solution is to create a new directory in one of the FreeCAD Mod directories (/wiki/index.php?title=Installing_more_workbenches), and to save our script in it. For example, let's make a "MyScripts" directory, and save our script as "exercise.py".

Now, everything is ready, let's start FreeCAD, create a new document, and, in the python interpreter, issue: < /translate>

```
import exercise
```

<translate> If no error message appear, that means our exercise script has been loaded. We can now check its contents with:< /translate>

```
dir(exercise)
```

<translate> The command dir() is a built-in python command that lists the contents of a module. We can see that our line() class is there, waiting for us. Now let's test it: < /translate>

```
exercise.line()
```

<translate> Then, click two times in the 3D view, and bingo, here is our line! To do it again, just type exercise.line() again, and again, and again... Feels great, no?

Registering the script in the FreeCAD interface

Now, for our new line tool to be really cool, it should have a button on the interface, so we don't need to type all that stuff everytime. The easiest way is to transform our new MyScripts directory into a full FreeCAD workbench. It is easy, all that is needed is to put a file called **InitGui.py** inside your MyScripts directory. The InitGui.py will contain the instructions to create a new workbench, and add our new tool to it. Besides that we will also need to transform a bit our exercise code, so the line() tool is recognized as an official FreeCAD command. Let's start by making an InitGui.py file, and write the following code in it: < /translate>

```
class MyWorkbench (Workbench):
    MenuText = "MyScripts"
    def Initialize(self):
        import exercise
        commandslist = ["line"]
        self.appendToolBar("My Scripts",commandslist)
Gui.addWorkbench(MyWorkbench())
```

<translate> By now, you should already understand the above script by yourself, I think: We create a new class that we call MyWorkbench, we give it a title (MenuText), and we define an Initialize() function that will be executed when the workbench is loaded into FreeCAD. In that function, we load in the contents of our exercise file, and append the FreeCAD commands found inside to a command list. Then, we make a toolbar called "My Scripts" and we assign our commands list to it. Currently, of course, we have only one tool, so our command list contains only one element. Then, once our workbench is ready, we add it to the main interface.

But this still won't work, because a FreeCAD command must be formatted in a certain way to work. So we will need to transform a bit our line() tool. Our new exercise.py script will now look like this:< /translate>

```
import FreeCADGui, Part
from pivy.coin import *
class line:
    "this class will create a line after the user clicked 2 points on the screen"
    def Activated(self):
        self.view = FreeCADGui.ActiveDocument.ActiveView
        self.stack = []
        self.callback = self.view.addEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.getpoint)
    def getpoint(self,event_cb):
        event = event_cb.getEvent()
        if event.getState() == SoMouseButtonEvent.DOWN:
            pos = event.getPosition()
            point = self.view.getPoint(pos[0],pos[1])
            self.stack.append(point)
            if len(self.stack) == 2:
                l = Part.Line(self.stack[0],self.stack[1])
                shape = l.toShape()
                Part.show(shape)
                self.view.removeEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.callback)
    def GetResources(self):
        return {'Pixmap' : 'path_to_an_icon/line_icon.png', 'MenuText': 'Line', 'ToolTip': 'Create a line by clicking 2 points on the screen'}
FreeCADGui.addCommand('line', line())
```

<translate> What we did here is transform our __init__() function into an Activated() function, because when FreeCAD commands are run, they automatically execute the Activated() function. We also added a GetResources() function, that informs FreeCAD where it can find an icon for the tool, and what will be the name and tooltip of our tool. Any jpg, png or svg image will work as an icon, it can be any size, but it is best to use a size that is close to the final aspect, like 16x16, 24x24 or 32x32. Then, we add the line() class as an official FreeCAD command with the addCommand() method.

That's it, we now just need to restart FreeCAD and we'll have a nice new workbench with our brand new line tool!

So you want more?

If you liked this exercise, why not try to improve this little tool? There are many things that can be done, like for example:

- Add user feedback: until now we did a very bare tool, the user might be a bit lost when using it. So we could add some feedback, telling him what to do next. For example, you could issue messages to the FreeCAD console. Have a look in the FreeCAD.Console module
- Add a possibility to type the 3D points coordinates manually. Look at the python input() function, for example
- Add the possibility to add more than 2 points

- Add events for other things: Now we just check for Mouse button events, what if we would also do something when the mouse is moved, like displaying current coordinates?
- Give a name to the created object

Don't hesitate to write your questions or ideas on the forum (<http://forum.freecadweb.org/>)!

< previous: Code snippets (/wiki/index.php?title=Code_snippets)
 next: Dialog creation > (/wiki/index.php?title=Dialog_creation)
 Index (/wiki/index.php?title=Online_Help_Toc)
 </translate>

< translate> In this page we will show how to build a simple Qt Dialog with Qt Designer (<http://qt-project.org/doc/qt-4.8/designer-manual.html>), Qt's official tool for designing interfaces, then convert it to python code, then use it inside FreeCAD. I'll assume in the example that you know how to edit and run python scripts already, and that you can do simple things in a terminal window such as navigate, etc. You must also have, of course, pyqt installed.

Designing the dialog

In CAD applications, designing a good UI (User Interface) is very important. About everything the user will do will be through some piece of interface: reading dialog boxes, pressing buttons, choosing between icons, etc. So it is very important to think carefully to what you want to do, how you want the user to behave, and how will be the workflow of your action.

There are a couple of concepts to know when designing interface:

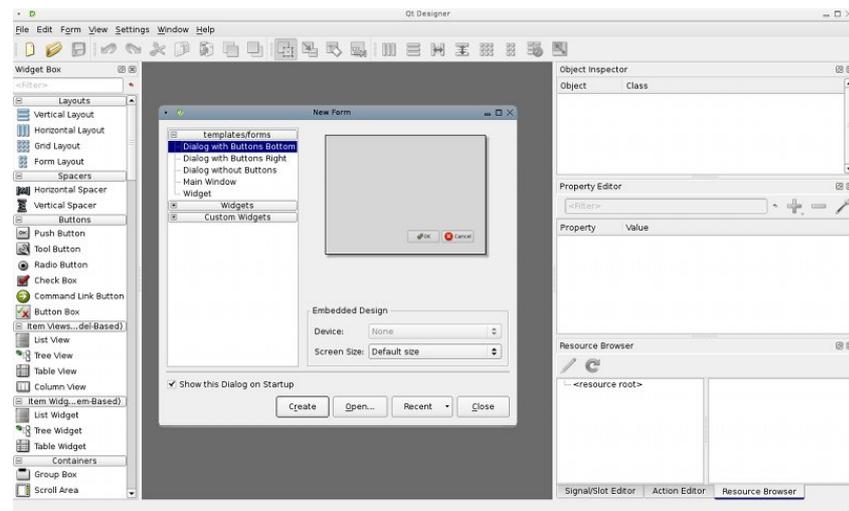
- Modal/non-modal dialogs (http://en.wikipedia.org/wiki/Modal_window): A modal dialog appears in front of your screen, stopping the action of the main window, forcing the user to respond to the dialog, while a non-modal dialog doesn't stop you from working on the main window. In some case the first is better, in other cases not.
- Identifying what is required and what is optional: Make sure the user knows what he must do. Label everything with proper description, use tooltips, etc.
- Separating commands from parameters: This is usually done with buttons and text input fields. The user knows that clicking a button will produce an action while changing a value inside a text field will change a parameter somewhere. Nowadays, though, users usually know well what is a button, what is an input field, etc. The interface toolkit we are using, Qt, is a state-of-the-art toolkit, and we won't have to worry much about making things clear, since they will already be very clear by themselves.

So, now that we have well defined what we will do, it's time to open the qt designer. Let's design a very simple dialog, like this:



(</wiki/index.php?title=File:Qttestdialog.jpg>)

We will then use this dialog in FreeCAD to produce a nice rectangular plane. You might find it not very useful to produce nice rectangular planes, but it will be easy to change it later to do more complex things. When you open it, Qt Designer looks like this:



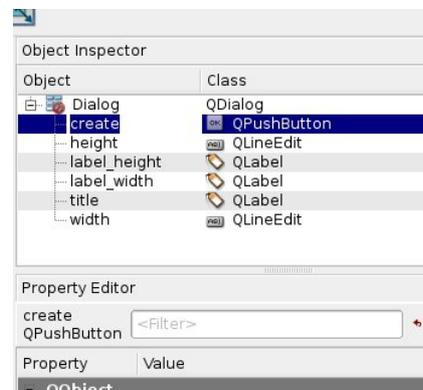
(/wiki/index.php?title=File:Qt designer-screenshot.jpg)

It is very simple to use. On the left bar you have elements that can be dragged on your widget. On the right side you have properties panels displaying all kinds of editable properties of selected elements. So, begin with creating a new widget. Select "Dialog without buttons", since we don't want the default Ok/Cancel buttons. Then, drag on your widget **3 labels**, one for the title, one for writing "Height" and one for writing "Width". Labels are simple texts that appear on your widget, just to inform the user. If you select a label, on the right side will appear several properties that you can change if you want, such as font style, height, etc.

Then, add **2 LineEdits**, which are text fields that the user can fill in, one for the height and one for the width. Here too, we can edit properties. For example, why not set a default value? For example 1.00 for each. This way, when the user will see the dialog, both values will be filled already and if he is satisfied he can directly press the button, saving precious time. Then, add a **PushButton**, which is the button the user will need to press after he filled the 2 fields.

Note that I choosed here very simple controls, but Qt has many more options, for example you could use Spinboxes instead of LineEdits, etc... Have a look at what is available, you will surely have other ideas.

That's about all we need to do in Qt Designer. One last thing, though, let's rename all our elements with easier names, so it will be easier to identify them in our scripts:



(/wiki/index.php?

title=File:Qtpropeditor.jpg) </translate>< translate>

Converting our dialog to python

</translate>< translate> Now, let's save our widget somewhere. It will be saved as an .ui file, that we will easily convert to python script with pyuic. On windows, the pyuic program is bundled with pyqt (to be verified), on linux you probably will need to install it separately from your package manager (on debian-based systems, it is part of the pyqt4-dev-tools package). To do the conversion, you'll need to open a terminal window (or a command prompt window on windows), navigate to where you saved your .ui file, and issue:< /translate>

```
pyuic mywidget.ui > mywidget.py
```

<translate> Into Windows pyuic.py are located in "C:\Python27\Lib\site-packages\PyQt4\uic\pyuic.py" For create batch file "compQt4.bat:< /translate>

```
@"C:\Python27\python" "C:\Python27\Lib\site-packages\PyQt4\uic\pyuic.py" -x %1.ui > %1.py
```

<translate> In the console Dos type without extension< /translate>

```
compQt4 myUiFile
```

<translate> Into Linux : to do< /translate>

<translate> Since FreeCAD progressively moved away from PyQt after version 0.13, in favour of PySide (<http://qt-project.org/wiki/PySide>) (Choice your PySide install building PySide (<http://pyside.readthedocs.org/en/latest/building/>)), to make the file based on PySide now you have to use:

</translate>

```
pyside-uic mywidget.ui -o mywidget.py
```

<translate> Into Windows uic.py are located in "C:\Python27\Lib\site-packages\PySide\scripts\uic.py" For create batch file "compSide.bat":

```
@"C:\Python27\python" "C:\Python27\Lib\site-packages\PySide\scripts\uic.py" %1.ui > %1.py
```

In the console Dos type without extension

```
compSide myUiFile
```

Into Linux : to do

On some systems the program is called pyuic4 instead of pyuic. This will simply convert the .ui file into a python script. If we open the mywidget.py file, its contents are very easy to understand:< /translate>

```
from PySide import QtCore, QtGui

class Ui_Dialog(object):
    def setupUi(self, Dialog):
        Dialog.setObjectName("Dialog")
        Dialog.resize(187, 178)
        self.title = QtGui.QLabel(Dialog)
        self.title.setGeometry(QtCore.QRect(10, 10, 271, 16))
        self.title.setObjectName("title")
        self.label_width = QtGui.QLabel(Dialog)
        ...

        self.retranslateUi(Dialog)
        QtCore.QMetaObject.connectSlotsByName(Dialog)

    def retranslateUi(self, Dialog):
        Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None, QtGui.QApplication.UnicodeUTF8))
        self.title.setText(QtGui.QApplication.translate("Dialog", "Plane-0-Matic", None, QtGui.QApplication.UnicodeUTF8))
        ...
```

<translate> As you see it has a very simple structure: a class named Ui_Dialog is created, that stores the interface elements of our widget. That class has two methods, one for setting up the widget, and one for translating its contents, that is part of the general Qt mechanism for

translating interface elements. The setup method simply creates, one by one, the widgets as we defined them in Qt Designer, and sets their options as we decided earlier. Then, the whole interface gets translated, and finally, the slots get connected (we'll talk about that later).

We can now create a new widget, and use this class to create its interface. We can already see our widget in action, by putting our mywidget.py file in a place where FreeCAD will find it (in the FreeCAD bin directory, or in any of the Mod subdirectories), and, in the FreeCAD python interpreter, issue:< /translate>

```
from PySide import QtGui
import mywidget
d = QtGui.QWidget()
d.ui = mywidget.Ui_Dialog()
d.ui.setupUi(d)
d.show()
```

<translate> And our dialog will appear! Note that our python interpreter is still working, we have a non-modal dialog. So, to close it, we can (apart from clicking its close icon, of course) issue:< /translate>

```
d.hide()
```

<translate>

Making our dialog do something

Now that we can show and hide our dialog, we just need to add one last part: To make it do something! If you play a bit with Qt designer, you'll quickly discover a whole section called "signals and slots". Basically, it works like this: elements on your widgets (in Qt terminology, those elements are themselves widgets) can send signals. Those signals differ according to the widget type. For example, a button can send a signal when it is pressed and when it is released. Those signals can be connected to slots, which can be special functionality of other widgets (for example a dialog has a "close" slot to which you can connect the signal from a close button), or can be custom functions. The PyQt Reference Documentation (<http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/classes.html>) lists all the qt widgets, what they can do, what signals they can send, etc...

What we will do here, is to create a new function that will create a plane based on height and width, and to connect that function to the pressed signal emitted by our "Create!" button. So, let's begin with importing our FreeCAD modules, by putting the following line at the top of the script, where we already import QtCore and QtGui: < /translate>

```
import FreeCAD, Part
```

<translate> Then, let's add a new function to our Ui_Dialog class:< /translate>

```
def createPlane(self):
    try:
        # first we check if valid numbers have been entered
        w = float(self.width.text())
        h = float(self.height.text())
    except ValueError:
        print "Error! Width and Height values must be valid numbers!"
    else:
        # create a face from 4 points
        p1 = FreeCAD.Vector(0,0,0)
        p2 = FreeCAD.Vector(w,0,0)
        p3 = FreeCAD.Vector(w,h,0)
        p4 = FreeCAD.Vector(0,h,0)
        pointslist = [p1,p2,p3,p4,p1]
        mywire = Part.makePolygon(pointslist)
        myface = Part.Face(mywire)
        Part.show(myface)
        self.hide()
```

<translate> Then, we need to inform Qt to connect the button to the function, by placing the following line just before QtCore.QMetaObject.connectSlotsByName(Dialog): < /translate>

```
QtCore.QObject.connect(self.create,QtCore.SIGNAL("pressed()"),self.createPlane)
```

<translate> This, as you see, connects the pressed() signal of our create object (the "Create!" button), to a slot named createPlane, which we just defined. That's it! Now, as a final touch, we can add a little function to create the dialog, it will be easier to call. Outside the Ui_Dialog class, let's add this code: < /translate>

```
class plane():
    def __init__(self):
        self.d = QtGui.QWidget()
        self.ui = Ui_Dialog()
        self.ui.setupUi(self.d)
        self.d.show()
```

<translate> (Python reminder: the __init__ method of a class is automatically executed whenever a new object is created!) Then, from FreeCAD, we only need to do: < /translate>

```
import mywidget
myDialog = mywidget.plane()
```

<translate> That's all Folks... Now you can try all kinds of things, like for example inserting your widget in the FreeCAD interface (see the Code snippets (/wiki/index.php?title=Code_snippets) page), or making much more advanced custom tools, by using other elements on your widget.

The complete script

This is the complete script, for reference: < /translate>

```

# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'mywidget.ui'
#
# Created: Mon Jun 1 19:09:10 2009
# by: PyQt4 UI code generator 4.4.4
# Modified for PySide 16:02:2015
# WARNING! All changes made in this file will be lost!

from PySide import QtCore, QtGui
import FreeCAD, Part

class Ui_Dialog(object):
    def setupUi(self, Dialog):
        Dialog.setObjectName("Dialog")
        Dialog.resize(187, 178)
        self.title = QtGui.QLabel(Dialog)
        self.title.setGeometry(QtCore.QRect(10, 10, 271, 16))
        self.title.setObjectName("title")
        self.label_width = QtGui.QLabel(Dialog)
        self.label_width.setGeometry(QtCore.QRect(10, 50, 57, 16))
        self.label_width.setObjectName("label_width")
        self.label_height = QtGui.QLabel(Dialog)
        self.label_height.setGeometry(QtCore.QRect(10, 90, 57, 16))
        self.label_height.setObjectName("label_height")
        self.width = QtGui.QLineEdit(Dialog)
        self.width.setGeometry(QtCore.QRect(60, 40, 111, 26))
        self.width.setObjectName("width")
        self.height = QtGui.QLineEdit(Dialog)
        self.height.setGeometry(QtCore.QRect(60, 80, 111, 26))
        self.height.setObjectName("height")
        self.create = QtGui.QPushButton(Dialog)
        self.create.setGeometry(QtCore.QRect(50, 140, 83, 26))
        self.create.setObjectName("create")

        self.retranslateUi(Dialog)
        QtCore.QObject.connect(self.create, QtCore.SIGNAL("pressed()"), self.createPlane)
        QtCore.QMetaObject.connectSlotsByName(Dialog)

    def retranslateUi(self, Dialog):
        Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None, QtGui.QApplication.UnicodeUTF8))
        self.title.setText(QtGui.QApplication.translate("Dialog", "Plane-0-Matic", None, QtGui.QApplication.UnicodeUTF8))
        self.label_width.setText(QtGui.QApplication.translate("Dialog", "Width", None, QtGui.QApplication.UnicodeUTF8))
        self.label_height.setText(QtGui.QApplication.translate("Dialog", "Height", None, QtGui.QApplication.UnicodeUTF8))
        self.create.setText(QtGui.QApplication.translate("Dialog", "Create!", None, QtGui.QApplication.UnicodeUTF8))

    def createPlane(self):
        try:
            # first we check if valid numbers have been entered
            w = float(self.width.text())
            h = float(self.height.text())
        except ValueError:
            print "Error! Width and Height values must be valid numbers!"
        else:
            # create a face from 4 points
            p1 = FreeCAD.Vector(0,0,0)
            p2 = FreeCAD.Vector(w,0,0)
            p3 = FreeCAD.Vector(w,h,0)
            p4 = FreeCAD.Vector(0,h,0)
            pointslist = [p1,p2,p3,p4,p1]
            mywire = Part.makePolygon(pointslist)
            myface = Part.Face(mywire)
            Part.show(myface)

class plane():
    def __init__(self):
        self.d = QtGui.QWidget()
        self.ui = Ui_Dialog()
        self.ui.setupUi(self.d)
        self.d.show()

```

<translate>

Creation of a dialog with buttons

Method 1

An example of a dialog box complete with its connections. < /translate>

```

# -*- coding: utf-8 -*-
# Create by flachyjoe

from PySide import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
        return s

try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig, _encoding)
except AttributeError:
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig)

class Ui_MainWindow(object):

    def __init__(self, MainWindow):
        self.window = MainWindow

        MainWindow.setObjectName(_fromUtf8("MainWindow"))
        MainWindow.resize(400, 300)
        self.centralWidget = QtGui.QWidget(MainWindow)
        self.centralWidget.setObjectName(_fromUtf8("centralWidget"))

        self.pushButton = QtGui.QPushButton(self.centralWidget)
        self.pushButton.setGeometry(QtCore.QRect(30, 170, 93, 28))
        self.pushButton.setObjectName(_fromUtf8("pushButton"))
        self.pushButton.clicked.connect(self.on_pushButton_clicked) #connection pushButton

        self.lineEdit = QtGui.QLineEdit(self.centralWidget)
        self.lineEdit.setGeometry(QtCore.QRect(30, 40, 211, 22))
        self.lineEdit.setObjectName(_fromUtf8("lineEdit"))
        self.lineEdit.returnPressed.connect(self.on_lineEdit_clicked) #connection lineEdit

        self.checkBox = QtGui.QCheckBox(self.centralWidget)
        self.checkBox.setGeometry(QtCore.QRect(30, 90, 81, 20))
        self.checkBox.setChecked(True)
        self.checkBox.setObjectName(_fromUtf8("checkBoxON"))
        self.checkBox.clicked.connect(self.on_checkBox_clicked) #connection checkBox

        self.radioButton = QtGui.QRadioButton(self.centralWidget)
        self.radioButton.setGeometry(QtCore.QRect(30, 130, 95, 20))
        self.radioButton.setObjectName(_fromUtf8("radioButton"))
        self.radioButton.clicked.connect(self.on_radioButton_clicked) #connection radioButton

        MainWindow.setCentralWidget(self.centralWidget)

        self.menuBar = QtGui.QMenuBar(MainWindow)
        self.menuBar.setGeometry(QtCore.QRect(0, 0, 400, 26))
        self.menuBar.setObjectName(_fromUtf8("menuBar"))
        MainWindow.setMenuBar(self.menuBar)

        self.mainToolBar = QtGui.QToolBar(MainWindow)
        self.mainToolBar.setObjectName(_fromUtf8("mainToolBar"))
        MainWindow.addToolBar(QtCore.Qt.TopToolBarArea, self.mainToolBar)

        self.statusBar = QtGui.QStatusBar(MainWindow)
        self.statusBar.setObjectName(_fromUtf8("statusBar"))
        MainWindow.setStatusBar(self.statusBar)

        self.retranslateUi(MainWindow)

    def retranslateUi(self, MainWindow):
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow", None))
        self.pushButton.setText(_translate("MainWindow", "OK", None))
        self.lineEdit.setText(_translate("MainWindow", "tyty", None))
        self.checkBox.setText(_translate("MainWindow", "CheckBox", None))
        self.radioButton.setText(_translate("MainWindow", "RadioButton", None))

    def on_checkBox_clicked(self):
        if self.checkBox.checkState()==0:
            App.Console.PrintMessage(str(self.checkBox.checkState())+" CheckBox KO\r\n")
        else:
            App.Console.PrintMessage(str(self.checkBox.checkState())+" CheckBox OK\r\n")
# App.Console.PrintMessage(str(self.lineEdit.setText("tititi"))+" LineEdit\r\n") #write
text to the lineEdit window !
# str(self.lineEdit.setText("tititi")) #écrit le texte dans la fenêtre lineEdit
App.Console.PrintMessage(str(self.lineEdit.displayText())+" LineEdit\r\n")

    def on_radioButton_clicked(self):
        if self.radioButton.isChecked():

```

```

        App.Console.PrintMessage(str(self.radioButton.isChecked())+" Radio OK\r\n")
    else:
        App.Console.PrintMessage(str(self.radioButton.isChecked())+" Radio KO\r\n")

    def on_lineEdit_clicked(self):
#        if self.lineEdit.textChanged():
            App.Console.PrintMessage(str(self.lineEdit.displayText())+" LineEdit Display\r\n"
)

    def on_pushButton_clicked(self):
        App.Console.PrintMessage("Terminé\r\n")
        self.window.hide()

MainWindow = QtGui.QMainWindow()
ui = Ui_MainWindow(MainWindow)
MainWindow.show()

```

<translate> Here the same window but with an icon on each button.

Download associated icons (Click rigth "Copy the image below ...")



(/wiki/index.php?title=File:Icône01.png)



(/wiki/index.php?title=File:Icône02.png)



(/wiki/index.php?title=File:Icône03.png)

</translate>

```

# -*- coding: utf-8 -*-

from PySide import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
        return s

try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig, _encoding)
except AttributeError:
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig)

class Ui_MainWindow(object):

    def __init__(self, MainWindow):
        self.window = MainWindow
        path = FreeCAD.ConfigGet("UserAppData")
#         path = FreeCAD.ConfigGet("AppHomePath")

        MainWindow.setObjectName(_fromUtf8("MainWindow"))
        MainWindow.resize(400, 300)
        self.centralWidget = QtGui.QWidget(MainWindow)
        self.centralWidget.setObjectName(_fromUtf8("centralWidget"))

        self.pushButton = QtGui.QPushButton(self.centralWidget)
        self.pushButton.setGeometry(QtCore.QRect(30, 170, 93, 28))
        self.pushButton.setObjectName(_fromUtf8("pushButton"))
        self.pushButton.clicked.connect(self.on_pushButton_clicked) #connection pushButton

        self.lineEdit = QtGui.QLineEdit(self.centralWidget)
        self.lineEdit.setGeometry(QtCore.QRect(30, 40, 211, 22))
        self.lineEdit.setObjectName(_fromUtf8("lineEdit"))
        self.lineEdit.returnPressed.connect(self.on_lineEdit_clicked) #connection lineEdit

        self.checkBox = QtGui.QCheckBox(self.centralWidget)
        self.checkBox.setGeometry(QtCore.QRect(30, 90, 100, 20))
        self.checkBox.setChecked(True)
        self.checkBox.setObjectName(_fromUtf8("checkBoxON"))
        self.checkBox.clicked.connect(self.on_checkBox_clicked) #connection checkBox

        self.radioButton = QtGui.QRadioButton(self.centralWidget)
        self.radioButton.setGeometry(QtCore.QRect(30, 130, 95, 20))
        self.radioButton.setObjectName(_fromUtf8("radioButton"))
        self.radioButton.clicked.connect(self.on_radioButton_clicked) #connection radioButton

        MainWindow.setCentralWidget(self.centralWidget)

        self.menuBar = QtGui.QMenuBar(MainWindow)
        self.menuBar.setGeometry(QtCore.QRect(0, 0, 400, 26))
        self.menuBar.setObjectName(_fromUtf8("menuBar"))
        MainWindow.setMenuBar(self.menuBar)

        self.mainToolBar = QtGui.QToolBar(MainWindow)
        self.mainToolBar.setObjectName(_fromUtf8("mainToolBar"))
        MainWindow.addToolBar(QtCore.Qt.TopToolBarArea, self.mainToolBar)

        self.statusBar = QtGui.QStatusBar(MainWindow)
        self.statusBar.setObjectName(_fromUtf8("statusBar"))
        MainWindow.setStatusBar(self.statusBar)

        self.retranslateUi(MainWindow)

        # Affiche un icone sur le bouton PushButton
        # self.image_01 = "C:\Program Files\FreeCAD0.13\Icône01.png" # adapt the icon name
        self.image_01 = path+"Icône01.png" # adapt the name of the icon
        icon01 = QtGui.QIcon()
        icon01.addPixmap(QtGui.QPixmap(self.image_01),QtGui.QIcon.Normal, QtGui.QIcon.Off)
        self.pushButton.setIcon(icon01)
        self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the
        direction of the button

        # Affiche un icone sur le bouton RadioButton
        # self.image_02 = "C:\Program Files\FreeCAD0.13\Icône02.png" # adapt the name of the i
        con
        self.image_02 = path+"Icône02.png" # adapter le nom de l'icone
        icon02 = QtGui.QIcon()
        icon02.addPixmap(QtGui.QPixmap(self.image_02),QtGui.QIcon.Normal, QtGui.QIcon.Off)
        self.radioButton.setIcon(icon02)
        # self.radioButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses
        the direction of the button

```

```

# Affiche un icone sur le bouton CheckBox
# self.image_03 = "C:\Program Files\FreeCAD0.13\Icône03.png" # the name of the icon
self.image_03 = path+"Icône03.png" # adapter le nom de l'icone
icon03 = QtGui.QIcon()
icon03.addPixmap(QtGui.QPixmap(self.image_03),QtGui.QIcon.Normal, QtGui.QIcon.Off)
self.checkBox.setIcon(icon03)
# self.checkBox.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the
direction of the button

def retranslateUi(self, MainWindow):
    MainWindow.setWindowTitle(_translate("MainWindow", "FreeCAD", None))
    self.pushButton.setText(_translate("MainWindow", "OK", None))
    self.lineEdit.setText(_translate("MainWindow", "tyty", None))
    self.checkBox.setText(_translate("MainWindow", "CheckBox", None))
    self.radioButton.setText(_translate("MainWindow", "RadioButton", None))

def on_checkBox_clicked(self):
    if self.checkBox.checkState()==0:
        App.Console.PrintMessage(str(self.checkBox.checkState())+" CheckBox KO\r\n")
    else:
        App.Console.PrintMessage(str(self.checkBox.checkState())+" CheckBox OK\r\n")
        # App.Console.PrintMessage(str(self.lineEdit.setText("tititi"))+" LineEdit\r\n") #
write text to the lineEdit window !
        # str(self.lineEdit.setText("tititi")) #écrit le texte dans la fenêtre lineEdit
        App.Console.PrintMessage(str(self.lineEdit.displayText())+" LineEdit\r\n")

def on_radioButton_clicked(self):
    if self.radioButton.isChecked():
        App.Console.PrintMessage(str(self.radioButton.isChecked())+" Radio OK\r\n")
    else:
        App.Console.PrintMessage(str(self.radioButton.isChecked())+" Radio KO\r\n")

def on_lineEdit_clicked(self):
    # if self.lineEdit.textChanged():
        App.Console.PrintMessage(str(self.lineEdit.displayText())+" LineEdit Display\r\n")

def on_pushButton_clicked(self):
    App.Console.PrintMessage("Terminé\r\n")
    self.window.hide()

MainWindow = QtGui.QMainWindow()
ui = Ui_MainWindow(MainWindow)
MainWindow.show()

```

<translate> Here the code to display the icon on the **pushButton**, change the name for another button, (**radioButton**, **checkBox**) and the path to the icon. </translate>

```

# Affiche un icône sur le bouton PushButton
# self.image_01 = "C:\Program Files\FreeCAD0.13\icone01.png" # the name of the icon
self.image_01 = path+"icone01.png" # the name of the icon
icon01 = QtGui.QIcon()
icon01.addPixmap(QtGui.QPixmap(self.image_01),QtGui.QIcon.Normal, QtGui.QIcon.Off)
self.pushButton.setIcon(icon01)
self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the
direction of the button

```

<translate> The command **UserAppData** gives the user path **AppHomePath** gives the installation path of FreeCAD</translate>

```

# path = FreeCAD.ConfigGet("UserAppData")
path = FreeCAD.ConfigGet("AppHomePath")

```

<translate> This command reverses the horizontal button, right to left.</translate>

```

self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the directio
n of the button

```

<translate>

Method 2

Another method to display a window, here by creating a file **QtForm.py** which contains the header program (module called with **import QtForm**), and a second module that contains the code window all these accessories, and your code (the calling module).

This method requires two separate files, but allows to shorten your program using the file ' ' QtForm.py ' ' import. Then distribute the two files together, they are inseparable.

The file **QtForm.py** < /translate>

```
# -*- coding: utf-8 -*-
# Create by flachyjoie
from PySide import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
        return s

try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig, _encoding)
except AttributeError:
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig)

class Form(object):
    def __init__(self, title, width, height):
        self.window = QtGui.QMainWindow()
        self.title=title
        self.window.setObjectName(_fromUtf8(title))
        self.window.setWindowTitle(_translate(self.title, self.title, None))
        self.window.resize(width, height)

    def show(self):
        self.createUI()
        self.retranslateUI()
        self.window.show()

    def setText(self, control, text):
        control.setText(_translate(self.title, text, None))
```

<translate> The appellant, file that contains the window and your code.

The file **my_file.py**

The connections are to do, a good exercise.< /translate>

```
# -*- coding: utf-8 -*-
# Create by flachyjoie
from PySide import QtCore, QtGui
import QtForm

class myForm(QtForm.Form):
    def createUI(self):
        self.centralWidget = QtGui.QWidget(self.window)
        self.window.setCentralWidget(self.centralWidget)

        self.pushButton = QtGui.QPushButton(self.centralWidget)
        self.pushButton.setGeometry(QtCore.QRect(30, 170, 93, 28))
        self.pushButton.clicked.connect(self.on_pushButton_clicked)

        self.lineEdit = QtGui.QLineEdit(self.centralWidget)
        self.lineEdit.setGeometry(QtCore.QRect(30, 40, 211, 22))

        self.checkBox = QtGui.QCheckBox(self.centralWidget)
        self.checkBox.setGeometry(QtCore.QRect(30, 90, 81, 20))
        self.checkBox.setChecked(True)

        self.radioButton = QtGui.QRadioButton(self.centralWidget)
        self.radioButton.setGeometry(QtCore.QRect(30, 130, 95, 20))

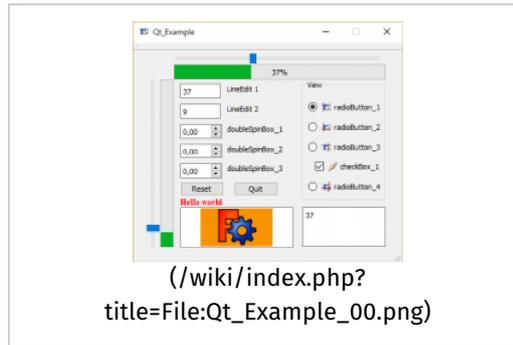
    def retranslateUI(self):
        self.setText(self.pushButton, "Fermer")
        self.setText(self.lineEdit, "essai de texte")
        self.setText(self.checkBox, "CheckBox")
        self.setText(self.radioButton, "RadioButton")

    def on_pushButton_clicked(self):
        self.window.hide()

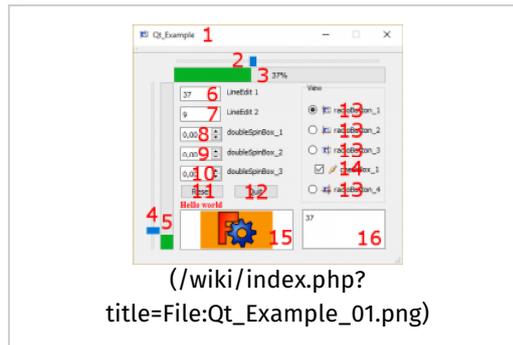
myWindow=myForm("Fenetre de test",400,300)
myWindow.show()
```

<translate>

Other example



Qt example 1



Qt example details

Are treated :

1. icon for window
2. horizontalSlider
3. progressBar horizontal
4. verticalSlider
5. progressBar vertical
6. lineEdit
7. lineEdit
8. doubleSpinBox
9. doubleSpinBox
10. doubleSpinBox
11. button
12. button
13. radioButton with icons
14. checkBox with icon checked and unchecked
15. textEdit
16. graphicsView with 2 graphs

The code page and the icons Qt_Example (/wiki/index.php?title=Qt_Example)

</translate>< translate>

Use QFileDialog for write the file

Complete code:< /translate>

```

# -*- coding: utf-8 -*-
import PySide
from PySide import QtGui ,QtCore
from PySide.QtGui import *
from PySide.QtCore import *
path = FreeCAD.ConfigGet("UserAppData")

try:
    SaveName = QFileDialog.getSaveFileName(None,QString.fromLocal8Bit("Save a file txt"),path,
        "*.txt") # PyQt4
    #
    on windows" "here the filter (extension)"
except Exception:
    SaveName, Filter = PySide.QtGui.QFileDialog.getSaveFileName(None, "Save a file txt", path,
        "*.txt") # PySide
    #
    on windows" "here the filter (extension)"
if SaveName == "":
    le are not selected then Abord process
    App.Console.PrintMessage("Process aborted"+"\\n")
else:
    le are selected or created then
    App.Console.PrintMessage("Registration of "+SaveName+"\\n")
    to Report view (Menu > View > Report view checked)
    try:
    ..
        file = open(SaveName, 'w')
    selected to write (w)
        try:
    ted to write ...
            # here your code
            print "here your code"
            file.write(str(1)+"\\n")
    er convert in text with (str())
            file.write("FreeCAD the best")
    text with (" ")
        except Exception:
    ted to write
            App.Console.PrintError("Error write file "+"\\n")
    .. display the text in red (PrintError)
        finally:
    ted to write ... or not the file is closed
            file.close()
    ted to write ... or not the file is closed
        except Exception:
            App.Console.PrintError("Error Open file "+SaveName+"\\n")
    ay the text in red (PrintError)

```

<translate>

Use QFileDialog for read the file

Complete code:< /translate>

```

# -*- coding: utf-8 -*-
import PySide
from PySide import QtGui ,QtCore
from PySide.QtGui import *
from PySide.QtCore import *
path = FreeCAD.ConfigGet("UserAppData")

OpenName = ""
try:
    OpenName = QFileDialog.getOpenFileName(None,QString.fromLocal8Bit("Read a file txt"),path,
        "*.txt") # PyQt4
#
# on windows" "here the filter (extension)"
except Exception:
    OpenName, Filter = PySide.QtGui.QFileDialog.getOpenFileName(None, "Read a file txt", path,
        "*.txt") #PySide
#
# on windows" "here the filter (extension)"
if OpenName == "":
    # if the name fi
    # le are not selected then Abord process
    App.Console.PrintMessage("Process aborted"+"\\n")
else:
    App.Console.PrintMessage("Read "+OpenName+"\\n")
    # text displayed
    # to Report view (Menu > View > Report view checked)
    try:
        # detect error t
    o read file
        file = open(OpenName, "r")
        # open the file
    selected to read (r) # (rb is binary)
        try:
            # detect error .
        ..
            # here your code
            print "here your code"
            op = OpenName.split("/")
            # decode the pat
        h
            op2 = op[-1].split(".")
            # decode the fil
        e name
            nomF = op2[0]
            # the file name
        are isolated
            App.Console.PrintMessage(str(nomF)+"\\n")
            # the file name
        are displayed
            for ligne in file:
                # read the file
                X = ligne.rstrip('\\n\\r') #.split()
                # decode the lin
            e
                print X
                # print the line
            in report view other method
                # (Menu > Edit >
            preferences... > Output window > Redirect internal Python output (and errors) to report view
            checked)
            except Exception:
                # if error detec
            ted to read
                App.Console.PrintError("Error read file "+"\\n")
                # detect error .
            .. display the text in red (PrintError)
            finally:
                # if error detec
            ted to read ... or not error the file is closed
                file.close()
                # if error detec
            ted to read ... or not error the file is closed
            except Exception:
                # if one error d
            etected to read file
                App.Console.PrintError("Error in Open the file "+OpenName+"\\n")
                # if one error d
            etected ... display the text in red (PrintError)

```

<translate>

Use QColorDialog for get the color

Complete code:< /translate>

```
# -*- coding: utf-8 -*-
# https://deptinfo-ensip.univ-poitiers.fr/ENS/pyside-docs/PySide/QtGui/QColor.html
import PySide
from PySide import QtGui ,QtCore
from PySide.QtGui import *
from PySide.QtCore import *
path = FreeCAD.ConfigGet("UserAppData")

couleur = QtGui.QColorDialog.getColor()
if couleur.isValid():
    red = int(str(couleur.name())[1:3],16) # decode hexadecimal to int()
    green = int(str(couleur.name())[3:5],16) # decode hexadecimal to int()
    blue = int(str(couleur.name())[5:7],16) # decode hexadecimal to int()

    print couleur #
    print "hexadecimal ",couleur.name() # color format hexadecimal mode 16
    print "Red color ",red # color format decimal
    print "Green color ",green # color format decimal
    print "Blue color ",blue # color format decimal
```

<translate>

Some useful commands

</translate>

```

# Here the code to display the icon on the '''pushButton''',
# change the name to another button, ('''radioButton, checkBox''') as well as the path to the
icon,

# Displays an icon on the button PushButton
# self.image_01 = "C:\Program Files\FreeCAD0.13\icone01.png" # he name of the icon
self.image_01 = path+"icone01.png" # the name of the icon
icon01 = QtGui.QIcon()
icon01.addPixmap(QtGui.QPixmap(self.image_01),QtGui.QIcon.Normal, QtGui.QIcon.Off)
self.pushButton.setIcon(icon01)
self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the d
irection of the button

# path = FreeCAD.ConfigGet("UserAppData") # gives the user path
path = FreeCAD.ConfigGet("AppHomePath") # gives the installation path of FreeCAD

# This command reverses the horizontal button, right to left
self.pushButton.setLayoutDirection(QtCore.Qt.RightToLeft) # This command reverses the horizont
al button

# Displays an info button
self.pushButton.setToolTip(_translate("MainWindow", "Quitter la fonction", None)) # Displays a
n info button

# This function gives a color button
self.pushButton.setStyleSheet("background-color: red") # This function gives a color button

# This function gives a color to the text of the button
self.pushButton.setStyleSheet("color : #ff0000") # This function gives a color to the text of
the button

# combinaison des deux, bouton et texte
self.pushButton.setStyleSheet("color : #ff0000; background-color : #0000ff;" ) # combination
of the two, button, and text

# replace the icon in the main window
MainWindow.setWindowIcon(QtGui.QIcon('C:\Program Files\FreeCAD0.13\View-C3P.png'))

# connects a QLineEdit on execute
self.lineEdit.returnPressed.connect(self.execute) # connects a QLineEdit on "def execute" after
validation on enter
# self.lineEdit.textChanged.connect(self.execute) # connects a QLineEdit on "def execute" with
each keystroke on the keyboard

# display text in a QLineEdit
self.lineEdit.setText(str(val_X)) # Displays the value in the QLineEdit (convert to string)

# extract the string contained in a QLineEdit
val_X = self.lineEdit.text() # extract the (string) string contained in QLineEdit
val_X = float(val_X0) # converted the string to an floating
val_X = int(val_X0) # convert the string to an integer

# This code allows you to change the font and its attributes
font = QtGui.QFont()
font.setFamily("Times New Roman")
font.setPointSize(10)
font.setWeight(10)
font.setBold(True) # same result with tags "<b>your text</b>" (in quotes)
self.label_6.setFont(font)
self.label_6.setObjectName("label_6")
self.label_6.setStyleSheet("color : #ff0000") # This function gives a color to the text
self.label_6.setText(_translate("MainWindow", "Select a view", None))

```

<translate>

By using the characters with accents, where you get the error :

Several solutions are possible.

UnicodeDecodeError: 'utf8' codec can't decode bytes in position 0-2: invalid data< /translate>

```

# conversion from a QLineEdit
App.activeDocument().CopyRight.Text = str(unicode(self.lineEdit_20.text() , 'ISO-8859-1').enco
de('UTF-8'))
DESIGNED_BY = unicode(self.lineEdit_01.text(), 'ISO-8859-1').encode('UTF-8')

```

<translate> or with the procedure< /translate>

```

def utf8(unio):
    return unicode(unio).encode('UTF8')

```

UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in

position 9: ordinal not in range(128)

```
# conversion
a = u"Nom de l'élément : "
f.write(''a.encode('iso-8859-1')''+str(element_)+"\n")
```

<translate> or with the procedure< /translate>

```
def iso8859(encoder):
    return unicode(encoder).encode('iso-8859-1')
```

<translate> or< /translate>

```
iso8859(unichr(176))
```

<translate> or < /translate>

```
unichr(ord(176))
```

<translate> or < /translate>

```
uniteSs = "mm"+iso8859(unichr(178))
print unicode(uniteSs, 'iso8859')
```

<translate>

< previous: [Line drawing function \(/wiki/index.php?title=Line_drawing_function\)](#)

Index next: [Licence > \(/wiki/index.php?title=Licence\)](#)

[\(/wiki/index.php?title=Online_Help_Toc\)](#)

</translate>

Developing applications for FreeCAD

<translate>

Statement of the main developer

I know that the discussion on the "*right*" licence for open source occupied a significant portion of internet bandwidth and so is here the reason why, in my opinion, FreeCAD should have this one.

I chose the LGPL (<http://en.wikipedia.org/wiki/LGPL>) for the project and I know the pro and cons about the LGPL and will give you some reasons for that decision.

FreeCAD is a mixture of a library and an application, so the GPL would be a little bit strong for that. It would prevent writing commercial modules for FreeCAD because it would prevent linking with the FreeCAD base libs. You may ask why commercial modules at all? Therefore Linux is good example. Would Linux be so successful when the GNU C Library would be GPL and therefore prevent linking against non-GPL applications? And although I love the freedom of Linux, I also want to be able to use the very good NVIDIA 3D graphic driver. I understand and accept the reason NVIDIA does not wish to give away driver code. We all work for companies and need payment or at least food. So for me, a coexistence of open source and closed source software is not a bad thing, when it obeys the rules of the LGPL. I would like to see someone writing a Catia import/export processor for FreeCAD and distribute it for free or for some money. I don't like to force him to give away more than he wants to. That wouldn't be good neither for him nor for FreeCAD.

Nevertheless this decision is made only for the core system of FreeCAD. Every writer of an application module may make his own decision.

Licences used in FreeCAD

FreeCAD uses two different licenses, one for the application itself, and one for the documentation:

Lesser General Public Licence, version 2 or superior (LGPL2+) (<http://en.wikipedia.org/wiki/LGPL>)

For the core libs as stated in the .h and .cpp files in src/App src/Gui src/Base and all modules (/wiki/index.php?title=Workbenches) in src/Mod and for the executable as stated in the .h and .cpp files in src/main. The icons and other graphic parts are also LGPL.

Open Publication Licence (http://en.wikipedia.org/wiki/Open_Publication_License)

For the documentation on <http://www.freecadweb.org> (<http://www.freecadweb.org>) when not marked differently by the author

See FreeCAD's debian copyright file (<http://sourceforge.net/p/freecad/code/ci/master/tree/package/debian/copyright>) for more details about the licenses used by the different components found in FreeCAD

Impact of the licences

Private users

Private users can use FreeCAD free of charge and can do basically whatever they want to do with it: use it, copy it, modify it, redistribute it. They are always master of their data, they are not forced to update FreeCAD, change their usage of FreeCAD. Using FreeCAD doesn't bind them to any kind of contract or obligation.

Professional users

Can use FreeCAD freely, for any kind of private or professional work. They can customize the application as they wish. They can write open or closed source extensions to FreeCAD. They are always master of their data, they are not forced to update FreeCAD, change their usage of FreeCAD. Using FreeCAD doesn't bind them to any kind of contract or obligation.

Open Source developers

Can use FreeCAD as the groundwork for own extension modules for special purposes. They can choose either the GPL or the LGPL to allow the use of their work in proprietary software or not.

Commercial developers

Commercial developers can use FreeCAD as the groundwork for their own extension modules for special purposes and are not forced to make their modules open source. They can use all modules which use the LGPL. They are allowed to distribute FreeCAD along with their proprietary software. They will get the support of the author(s) as long as it is not a one way street.

OpenCasCade License side effects (for FreeCAD version 0.13 and older)

The following is no more applicable since version 0.14, since both FreeCAD and OpenCasCade are now fully LGPL.

Up to Version 0.13 FreeCAD is delivered as GPL2+, although the source itself is under LGPL2+. That's because of linkage of Coin3D (GPL2) and PyQt(GPL). Starting with 0.14 we will be completely GPL free. PyQt will be replaced by PySide, and Coin3D was re-licensed under BSD. One problem, we still have to face, license-wise, the OCTPL (Open CASCADE Technology Public License)

(<http://www.opencascade.org/getocc/license/>). Its a License mostly LGPL similar, with certain changes. On of the originators, Roman Lygin, elaborated on the License on his Blog (<http://opencascade.blogspot.de/2008/12/license-to-kill-license-to-use.html>). The home-brew OCTPL license leads to all kind of side effects for FreeCAD, which where widely discussed on different forums and mailing lists, e.g. on OpenCasCade forum itself (http://www.opencascade.org/org/forum/thread_15859/?forum=3). I will link here some articles for the biggest problems.

GPL2/GPL3/OCTLP incompatibility

We first discovered the problem by a discussion on the FSF (<http://www.fsf.org/>) high priority project discussion list (<https://groups.google.com/forum/#!topic/polignu/XRergtws80>). It was about a library we look at, which was licensed with GPL3. Since we linked back then with Coin3D, with GPL2 only, we was not able to adopt that lib. Also the OCTPL is considered GPL incompatible (<http://www.opencascade.org/occt/faq/>). This Libre Graphics World article "LibreDWG drama: the end or the new beginning?" (<http://libregraphicsworld.org/blog/entry/libredwg-drama-the-end-or-the-new-beginning>) shows up the drama of LibreDWG project not acceptably in FreeCAD or LibreCAD.

Debian

The incompatibility of the OCTPL was discussed on the debian legal list (<http://lists.debian.org/debian-legal/2009/10/msg00000.html>) and lead to a bug report on the FreeCAD package (<http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=617613>) which prevent (ignor-tag) the transition from debian-testing to the main distribution. But its also mentioned thats a FreeCAD, which is free of GPL code and libs, would be acceptably. With a re-licensed Coin3D V4 and a substituted PyQt we will hopefully reach GPL free with the 0.14 release.

Fedora/RedHat non-free

In the Fedora project OpenCasCade is listed "non-free". This means basically it won't make it into Fedora or RedHat. This means also FreeCAD won't make it into Fedora/RedHat until OCC is changing its license. Here the links to the license evaluation:

- Discussion on the Fedora-legal-list (<http://lists.fedoraproject.org/pipermail/legal/2011-September/001713.html>)
- License review entry in the RedHat bug tracker (https://bugzilla.redhat.com/show_bug.cgi?id=458974#c10)

The main problem they have AFIK is that the OCC license demand non discriminatory support fees if you want to do paid support. It has nothing to do with "free" or OpenSource, its all about RedHat's business model!

< previous: Dialog creation (/wiki/index.php?title=Dialog_creation)
 Index next: Tracker > (/wiki/index.php?title=Tracker)
 (/wiki/index.php?title=Online_Help_Toc)
 </translate>

< translate> The adress of our bug tracker is:

<http://www.freecadweb.org/tracker> (<http://www.freecadweb.org/tracker>)

There you can report bugs, submit feature requests, patches, or request to merge your branch if you developed something using git. The tracker is divided into modules, so please be specific and file your request in the appropriate subsection. In case of doubt, leave it in the "FreeCAD" section.

Please before creating tickets, always first discuss bugs in the Help forum (<http://forum.freecadweb.org/viewforum.php?f=3>) and feature requests in the Open discussion forum (<http://forum.freecadweb.org/viewforum.php?f=8>).

Reporting bugs

If you think you might have found a bug, you are welcome to report it there so long as you have first discussed the matter in the appropriate forum. But before reporting a bug, please check the following items:

- Make sure your bug is really a bug, that is, something that should be working and that is not working. If you are not sure, don't hesitate to explain your problem on the forum (<http://forum.freecadweb.org/>) and ask what to do.
- Before submitting anything, read the frequently asked questions (</wiki/index.php?title=FAQ>), do a search on the forum (<http://forum.freecadweb.org/>), and make sure the same bug hasn't been reported before, by doing a search on the bug tracker.
- Describe as clearly as possible the problem, and how it can be reproduced. If we can not verify the bug, we might not be able to fix it.
- Include all the information from the "Copy to Clipboard" button in the Help (menu) -> About FreeCAD dialogue and do so from either the Part or PartDesign workbench so that your data will include your OCC or OCE version.
- Please file one separate report for each bug.
- If you are on a linux system and your bug causes a crash in FreeCAD, you can try running a debug backtrace: From a terminal run *gdb freecad* (assuming package gdb is installed), then, inside gdb, type *run*. FreeCAD will then run. After the crash happens, type *bt*, to get the full backtrace. Include that backtrace in your bug report.

Requesting features

If you want something to appear in FreeCAD that is not implemented yet, it is not a bug but a feature request. You can also submit it on the same tracker (file it as feature request instead of bug), but keep in mind there are no guarantees that your wish will be fulfilled.

Submitting patches

In case you have programmed a bug fix, an extension or something else that can be of public use in FreeCAD, create a patch using the Git diff tool and submit it on the same tracker (file it as patch).

Requesting merge

If you have created a git branch containing changes that you would like to see merged into the FreeCAD code, you can ask there to have your branch reviewed and merged if the FreeCAD developers are OK with it. You must first publish your branch to a public git repository (github,bitbucket, sourceforge...) and then give the URL of your branch in your merge request.

```
< previous: Licence (/wiki/index.php?title=Licence)      Index
next:      CompileOnWindows          >      (/wiki/index.php?
title=CompileOnWindows)
          (/wiki/index.php?title=Online_Help_Toc)
</translate>
```

This article explains step by step **how to compile FreeCAD on Windows**.

See also [Compile on Windows with Visual Studio 2013 \(/wiki/index.php?title=Compile_on_Windows_with_VS2013\)](/wiki/index.php?title=Compile_on_Windows_with_VS2013)

Prerequisites

Required programs

- [Git \(http://git-scm.com/\)](http://git-scm.com/) There are a number of alternatives such as GitCola, Tortoise Git, and others.
- [CMake \(http://www.cmake.org/cmake/resources/software.html\)](http://www.cmake.org/cmake/resources/software.html) version 2.x.x or Cmake 3.x.x
- Python >2.5 (This is only required if NOT using the Libpack. The Libpack comes with a minimal Python(2.7.x) suitable for compiling and running FreeCAD)

Source Code

Using Git (Preferred)

To create a local tracking branch and download the source code you need to open a terminal(command prompt) and cd to the directory you want the source, then type:

```
git clone https://github.com/FreeCAD/FreeCAD.git free-cad-code
```

Compiler

On Windows, the default compiler is M\$ Visual Studio, be it the Express or Full 2008, 2012, or 2013 versions. You will also need to install the Windows Platform SDK to get several required libraries (e.g. Windows.h), though they may not be required with M\$ compilers (either full or express).

For those who want to avoid installing the huge Visual Studio for the mere purpose of having a compiler, see [CompileOnWindows - Reducing Disk Footprint \(/wiki/index.php?title=CompileOnWindows_-_Reducing_Disk_Footprint\)](/wiki/index.php?title=CompileOnWindows_-_Reducing_Disk_Footprint).

Note

Though it may be possible to use Cygwin or MinGW gcc it's not tested or ported so far.

Third Party Libraries

You will need all of the Third Party Libraries (/wiki/index.php?title=Third_Party_Libraries) to successfully compile FreeCAD. If you use the M\$ compilers it is recommended to install a FreeCAD LibPack (<http://sourceforge.net/projects/free-cad/files/FreeCAD%20LibPack/>), which provides all of the required libraries to build FreeCAD in Windows. You will need the Libpack for your architecture and compiler. FreeCAD currently supplies Libpack Version11 for x32 and x64, for VS9 2008, VS11 2012, and VS12 2013.

Optional programs

- [NSIS \(http://sourceforge.net/projects/nsis/\)](http://sourceforge.net/projects/nsis/) Windows installer (note: formerly, WiX (<http://wixtoolset.org/>) installer was used - now under transition to NSIS) - if you want to make msi installer

System Path Configuration

Inside your system path be sure to set the correct paths to the following programs:

- `git` (not `tortoiseGit`, but `git.exe`) This is necessary for Cmake to properly update the "About FreeCAD" information in the `version.h` file which allows FreeCAD to report the proper version in About FreeCAD from the help menu.
- Optionally you can include the Libpack in your system path. This is useful if you plan to build multiple configurations/versions of FreeCAD, you will need to copy less files as explained later in the build process.

To add to your system path:

- Start menu -> Right click on Computer -> Properties -> Advanced system settings
- Advanced tab -> Environment Variables...
- Add the `PATH/TO/GIT` to the **PATH**
- It should be separated from the others with a semicolon `;`

Configuration with CMake

The switch to CMake

Warning

Since FreeCAD version 0.9 we have stopped providing `.vcproj` files.

Currently, FreeCAD uses the CMake build system to generate build and make files that can be used between different operating systems and compilers. If you want build former versions of FreeCAD (0.8 and older) see Building older versions later in this article.

We switched because it became more and more painful to maintain project files for 30+ build targets and x compilers. CMake gives us the possibility to support alternative IDEs, like Code::Blocks, Qt Creator and Eclipse CDT. The main compiler is still M\$ VC9 Express, though. But we plan for the future a build process on Windows without proprietary compiler software.

CMake

The first step to build FreeCAD with CMake is to configure the environment. There are two ways to do it:

- Using the LibPack
- Installing all the needed libraries and let CMake find them

The following process will assume you are using the LipPack. The second option may be discussed in Options for the Build Process.

Configure CMake using GUI

- Open the CMake GUI
- Specify the source folder
- Specify the build folder
- Click **Configure**
- Specify the generator according to the IDE that you'll use.

This will begin configuration and should fail because the location of `FREECAD_LIBPACK_DIR` is unset.

- Expand the `FREECAD` category and set `FREECAD_LIBPACK_DIR` to the correct location
- Check `FREECAD_USE_EXTERNAL_PIVY`
- Optionally Check `FREECAD_USE_FREETYPE` this is required to use the Draft WB's Shape String functionality

- Click **Configure** again
- There should be no errors
- Click **Generate**
- Close CMake
- Copy **libpack\bin** folder into the new build folder CMake created

Options for the Build Process

The CMake build system gives us a lot more flexibility over the build process. That means we can switch on and off some features or modules. It's in a way like the Linux kernel build. You have a lot of switches to determine the build process.

Here is the description of some of these switches. They will most likely change a lot in the future because we want to increase the build flexibility a lot more.

Link table

Variable name	Description	Default
FREECAD_LIBPACK_USE	Switch the usage of the FreeCAD LibPack on or off	On Win32 on, otherwise off
FREECAD_LIBPACK_DIR	Directory where the LibPack is	FreeCAD SOURCE dir
FREECAD_BUILD_GUI	Build FreeCAD with all Gui related modules	ON
FREECAD_BUILD_CAM	Build the CAM module, experimental!	OFF
FREECAD_BUILD_INSTALLER	Create the project files for the Windows installer.	OFF
FREECAD_BUILD_DOXYGEN_DOCU	Create the project files for source code documentation.	OFF
FREECAD_MAINTAINERS_BUILD	Switch on stuff needed only when you do a Release build.	OFF

If you are building with Qt Creator, jump to Building with Qt Creator, otherwise proceed to Building with Visual Studio 9 2008.

Building FreeCAD

Depending on your current setup, the process for building FreeCAD will be slightly different. This is due to the differences in available software and software versions for each operating system.

The following procedure will work for compiling on Windows Vista/7/8, for XP an alternate VS tool set is required for VS 2012 and 2013, which has not been tested successfully with the current Libpacks. To target XP(both x32 and x64) it is recommended to use VS2008 and Libpack FreeCADLibs_11.0_x86_VC9.7z

Building with Visual Studio 12 2013

Make sure to specify **Visual Studio 12 x64**(or the alternate C-Compiler you are using) as the generator in CMake before you continue.

- Start Visual Studio 12 2013 by clicking on the desktop icon created at installation.
- Open the project by:

File -> Open -> Project/Solution

- Open FreeCAD_Trunk.sln from the build folder CMake created
- Switch the Solutions Configuration drop down at the top to **Release X64**

This may take a while depending on your sytem

- Build -> Build Solution
- This will take a long time...

If you don't get any errors you are done. Exit Visual Studio and start FreeCAD by double clicking the FreeCAD icon in the bin folder of the build directory.

Building with Visual Studio 9 2008

Warning

Visual C++ Express 2008 does not support 64-bit compilation. There is a workaround here (<http://jenshuebel.wordpress.com/2009/02/12/visual-c-2008-express-edition-and-64-bit-targets/>)

Make sure to specify **Visual Studio 9 2008** as the generator in CMake before you continue.

- Open **Visual Studio 9 2008** or **Visual C++ Express 2008**
- File -> Open -> Project/Solution
- Open **FreeCAD_Trunk.sln** from the build folder CMake created
- Switch the **Solutions Configuration** dropdown at the top to **Release**
- Build -> Build Solution to build
- Wait until the Build is finished (will take a while)

After Building

- Debug -> Start without Debugging
- Click popup menu under **Executable File Name** and choose **Browse**
- Go to the build\bin folder and choose **FreeCAD.exe**
- You are done!

Building with Qt Creator

Installation and configuration of Qt Creator

- Download and install Qt Creator (<https://qt-project.org/downloads>)
- Tools -> Options -> Text Editor -> Behavior tab:
 - File Encodings -> Default Encodings:

- Set to: **ISO-8859-1 /...csISOLatin1** (Certain characters create errors/warnings with Qt Creator if left set to UTF-8. This seems to fix it.)
- Tools -> Options -> Build & Run:
 - CMake tab
 - Fill Executable box with path to cmake.exe
 - Kits tab
 - Name: MSVC 2008
 - Compiler: Microsoft Visual C++ Compiler 9.0 (x86)
 - Debugger: Auto detected...
 - Qt version: None
 - General tab
 - Uncheck: Always build project before deploying it
 - Uncheck: Always deploy project before running it

Import project and Build

- File -> Open File or Project
- Open **CMakeLists.txt** which is in the top level of the source
- This will start CMake
- Choose build directory and click next
- Set generator to **NMake Generator (MSVC 2008)**
- Click Run CMake. Follow the instructions depicted above to configure CMake to your liking.

Now FreeCAD can be built

- Build -> Build All
- This will take a long time...

Once complete, it can be run: There are 2 green triangles at the bottom left. One is debug. The other is run. Pick whichever you want.

Command line build

Here an example how to build FreeCAD from the Command line:

```
rem @echo off
rem   Build script, uses vcbuild to completetly build FreeCAD

rem update trunc
d:
cd "D:\_Projekte\FreeCAD\FreeCAD_0.9"
"C:\Program Files (x86)\Subversion\bin\svn.exe" update

rem set the aprobiated Variables here or outside in the system

set PATH=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem
set INCLUDE=
set LIB=

rem Register VS Build programmms
call "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"

rem Set Standard include paths
set INCLUDE=%INCLUDE%;%FrameworkSDKDir%\include
set INCLUDE=%INCLUDE%;C:\Program Files\Microsoft SDKs\Windows\v6.0A\Include

rem Set lib Pathes
set LIB=%LIB%;C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib
set LIB=%LIB%;%PROGRAMFILES%\Microsoft Visual Studio\VC98\Lib

rem Start the Visuall Studio build process
"C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\vcpackages\vcbuild.exe" "D:\_Projekte\FreeCAD FreeCAD_0.9_build\FreeCAD_trunk.sln" /useenv
```

Building older versions

Using LibPack

To make it easier to get FreeCAD compiled, we provide a collection of all needed libraries. It's called the LibPack (/wiki/index.php?title=Third_Party_Libraries). You can find it on the download page (http://sourceforge.net/project/showfiles.php?group_id=49159) on sourceforge.

You need to set the following environment variables:

FREECADLIB = "D:\Wherever\LIBPACK"

QTDIR = "%FREECADLIB%"

Add "%FREECADLIB%\bin" and "%FREECADLIB%\dll" to the system *PATH* variable. Keep in mind that you have to replace "%FREECADLIB%" with the path name, since Windows does not recursively replace environment variables.

Directory setup in Visual Studio

Some search path of Visual Studio need to be set. To change them, use the menu *Tools*→*Options*→*Directory*

Includes

Add the following search path to the include path search list:

- %FREECADLIB%\include
- %FREECADLIB%\include\Python
- %FREECADLIB%\include\boost
- %FREECADLIB%\include\xercesc
- %FREECADLIB%\include\OpenCascade
- %FREECADLIB%\include\OpenCV
- %FREECADLIB%\include\Coin
- %FREECADLIB%\include\SoQt
- %FREECADLIB%\include\QT
- %FREECADLIB%\include\QT\Qt3Support
- %FREECADLIB%\include\QT\QtCore
- %FREECADLIB%\include\QT\QtGui
- %FREECADLIB%\include\QT\QtNetwork
- %FREECADLIB%\include\QT\QtOpenGL
- %FREECADLIB%\include\QT\QtSvg
- %FREECADLIB%\include\QT\QtUiTools
- %FREECADLIB%\include\QT\QtXml
- %FREECADLIB%\include\Gts
- %FREECADLIB%\include\zlib

Libs

Add the following search path to the lib path search list:

- %FREECADLIB%\lib

Executables

Add the following search path to the executable path search list:

- %FREECADLIB%\bin

- TortoiseSVN binary installation directory, usually "C:\Program Files\TortoiseSVN\bin", this is needed for a distribution build when *SubWVRev.exe* is used to extract the version number from Subversion.

Python needed

During the compilation some Python scripts get executed. So the Python interpreter has to function on the OS. Use a command box to check it. If the Python library is not properly installed you will get an error message like *Cannot find python.exe*. If you use the LibPack you can also use the *python.exe* in the bin directory.

Special for VC8

When building the project with VC8, you have to change the link information for the WildMagic library, since you need a different version for VC6 and VC8. Both versions are supplied in *LIBPACK/dll*. In the project properties for *AppMesh* change the library name for the *wm.dll* to the VC8 version. Take care to change it in Debug *and* Release configuration.

Compile

After you conform to all prerequisites the compilation is - hopefully - only a mouse click in VC

After Compiling

To get FreeCAD up and running from the compiler environment you need to copy a few files from the LibPack (/wiki/index.php?title=Third_Party_Libraries) to the *bin* folder where FreeCAD.exe is installed after a successful build:

- *python.exe* and *python_d.exe* from *LIBPACK/bin*
- *python25.dll* and *python25_d.dll* from *LIBPACK/bin*
- *python25.zip* from *LIBPACK/bin*
- make a copy of *Python25.zip* and rename it to *Python25_d.zip*
- *QtCore4.dll* from *LIBPACK/bin*
- *QtGui4.dll* from *LIBPACK/bin*
- *boost_signals-vc80-mt-1_34_1.dll* from *LIBPACK/bin*
- *boost_program_options-vc80-mt-1_34_1.dll* from *LIBPACK/bin*
- *xerces-c_2_8.dll* from *LIBPACK/bin*
- *zlib1.dll* from *LIBPACK/bin*
- *coin2.dll* from *LIBPACK/bin*
- *soqt1.dll* from *LIBPACK/bin*
- *QtOpenGL4.dll* from *LIBPACK/bin*
- *QtNetwork4.dll* from *LIBPACK/bin*
- *QtSvg4.dll* from *LIBPACK/bin*
- *QtXml4.dll* from *LIBPACK/bin*

When using a LibPack (/wiki/index.php?title=Third_Party_Libraries) with a Python version older than 2.5 you have to copy two further files:

- *zlib.pyd* and *zlib_d.pyd* from *LIBPACK/bin/lib*. This is needed by python to open the zipped python library.
- *_sre.pyd* and *_sre_d.pyd* from *LIBPACK/bin/lib*. This is needed by python for the built in help system.

If you don't get it running due to a Python error it is very likely that one of the *zlib*.pyd* files is missing.

Alternatively, you can copy the whole bin folder of libpack into bin folder of the build. This is easier, but takes time and disk space. This can be substituted by making links instead of copying files, see [CompileOnWindows - Reducing Disk Footprint \(/wiki/index.php?title=CompileOnWindows_-_Reducing_Disk_Footprint#avoiding_copying_any_libpack\)](http://wiki/index.php?title=CompileOnWindows_-_Reducing_Disk_Footprint#avoiding_copying_any_libpack)

Additional stuff

If you want to build the source code documentation you need Doxygen (<http://www.stack.nl/~dimitri/doxygen/>).

To create an installer package you need WIX (<http://wix.sourceforge.net/>).

During the compilation some Python scripts get executed. So the Python interpreter has to work properly.

For more details have also a look to *README.Linux* in your sources.

First of all you should build the Qt plugin that provides all custom widgets of FreeCAD we need for the Qt Designer. The sources are located under

```
//src/Tools/plugins/widget/.
```

So far we don't provide a makefile -- but calling

```
qmake plugin.pro
```

creates it. Once that's done, calling *make* will create the library

```
//libFreeCAD_widgets.so/.
```

To make this library known to your *Qt Designer* you have to copy the file to

```
//$QTDIR/plugin/designer/.
```

References

< previous: [Tracker \(/wiki/index.php?title=Tracker\)](http://wiki/index.php?title=Tracker) [Index](#)
 next: [CompileOnUnix > \(/wiki/index.php?title=CompileOnUnix\)](http://wiki/index.php?title=CompileOnUnix)
 ([/wiki/index.php?title=Online_Help_Toc](http://wiki/index.php?title=Online_Help_Toc))

On recent linux distributions, FreeCAD is generally easy to build, since all dependencies are usually provided by the package manager. It basically involves 3 steps:

1. Getting the FreeCAD source code
2. Getting the dependencies (packages FreeCAD depends upon)
3. Configure with "cmake" and Compile with "make"

Below, you'll find detailed explanations of the whole process and particularities you might encounter. If you find anything wrong or out-of-date in the text below (Linux distributions change often), or if you use a distribution which is not listed, please help us correcting it.

Getting the source

Before you can compile FreeCAD, you need the source code. There are 3 ways to get it:

Git

The quickest and best way to get the code is to clone the read-only git repository now hosted on GitHub (you need the git (<http://git-scm.com/>) package installed):

```
git clone https://github.com/FreeCAD/FreeCAD.git free-cad-code
```

This will place a copy of the latest version of the FreeCAD source code in a new directory called "free-cad-code".

Github

The official FreeCAD repository is on Github: github.com/FreeCAD/FreeCAD (<https://github.com/FreeCAD/FreeCAD>)

Source package

Alternatively you can download a source package, but they could be already quite old so it's always better to get the latest sources via git or github.

- Official FreeCAD source packages (distribution-independent):
FreeCAD-0.17_pre.zip
(https://github.com/FreeCAD/FreeCAD/archive/0.17_pre.zip).

Getting the dependencies

To compile FreeCAD under Linux you have to install all libraries mentioned in Third Party Libraries (/wiki/index.php?title=Third_Party_Libraries) first. Please note that the names and availability of the libraries will depend on your distribution. Note that if you don't use the most recent version of your distribution, some of the packages below might be missing from your repositories. In that case, look in the Older and non-conventional distributions section below.

Skip to Compile FreeCAD

Debian and Ubuntu

On Debian-based systems (Debian, Ubuntu, Mint, etc...) it is quite easy to get all needed dependencies installed. Most of the libraries are available via apt-get or synaptic package manager.

- build-essential
- cmake
- python
- python-matplotlib
- libtool

either:

- libcoin60-dev (Debian Wheezy, Wheezy-backports, Ubuntu 13.04 and before)

or:

- libcoin80-dev (Debian unstable(Jesse), testing, Ubuntu 13.10 and forward)
- libsoqt4-dev
- libxerces-c-dev
- libboost-dev
- libboost-filesystem-dev
- libboost-regex-dev
- libboost-program-options-dev
- libboost-signals-dev
- libboost-thread-dev
- libboost-python-dev
- libqt4-dev
- libqt4-opengl-dev

- qt4-dev-tools
- python-dev
- python-pyside
- pyside-tools

either:

- libopencascade-dev (official opencascade version)

or:

- liboce*-dev (opencascade community edition)
- oce-draw
- libeigen3-dev
- libqtwebkit-dev
- libshiboken-dev
- libpyside-dev
- libode-dev
- swig
- libzipios++-dev
- libfreetype6
- libfreetype6-dev

Additional instruction (<http://forum.freecadweb.org/viewtopic.php?f=4&t=5096#p40018>) for libcoin80-dev Debian wheezy-backports, unstable, testing, Ubuntu 13.10 and forward

Note that liboce*-dev includes the following libraries:

- liboce-foundation-dev
- liboce-modeling-dev
- liboce-ocaf-dev
- liboce-visualization-dev
- liboce-ocaf-lite-dev

You may have to install these packages by individual name.

Optionally you can also install these extra packages:

- libsimage-dev (to make Coin to support additional image file formats)
- checkinstall (to register your installed files into your system's package manager, so you can easily uninstall later)
- python-pivy (needed for the 2D Drafting module)
- python-qt4 (needed for the 2D Drafting module)
- doxygen and libcoin60-doc (if you intend to generate source code documentation)
- libspnav-dev (for 3Dconnexion devices support like the Space Navigator or Space Pilot)

Fedora

You need the following packages:

- gcc-c++ (or possibly another C++ compiler?)
- cmake
- doxygen
- swig

- gettext
- dos2unix
- desktop-file-utils
- libXmu-devel
- freeimage-devel
- mesa-libGLU-devel
- OCE-devel
- python
- python-devel
- python-pyside-devel
- pyside-tools
- boost-devel
- tbb-devel
- eigen3-devel
- qt-devel
- qt-webkit-devel
- ode-devel
- xerces-c
- xerces-c-devel
- opencv-devel
- smesh-devel
- coin3-devel (if coin2 is the latest available for your version of Fedora, use packages from <http://www.zultron.com/rpm-repo/> (<http://www.zultron.com/rpm-repo/>))
- soqt-devel
- freetype
- freetype-devel

And optionally:

- libspnav-devel (for 3Dconnexion devices support like the Space Navigator or Space Pilot)
- pivy (https://bugzilla.redhat.com/show_bug.cgi?id=458975 (https://bugzilla.redhat.com/show_bug.cgi?id=458975) Pivy is not mandatory but needed for the Draft module)

Gentoo

Easiest way to check which packages are needed to compile FreeCAD is to check via portage:

```
emerge -pv freecad
```

This should give a nice list of extra packages that you need installed on your system.

OpenSUSE

You need the following packages:

- gcc
- cmake
- OpenCASCADE-devel
- libXerces-c-devel

- python-devel
- libqt4-devel
- libshiboken-devel
- python-pyside-devel
- python-pyside-tools
- Coin-devel
- SoQt-devel
- boost-devel
- libode-devel
- libQtWebKit-devel
- libeigen3-devel
- gcc-fortran
- freetype2
- freetype2-devel

For FreeCAD 0.14 stable and 0.15 unstable you need to add Eigen3 and swig libraries, that don't seem to be in standard repos. You can get them with a one-click install here:

- Eigen3 (http://software.opensuse.org/search?q=eigen3&baseproject=openSUSE%3A12.1&lang=en&exclude_debug=true)
- swig (http://software.opensuse.org/search?q=swig&baseproject=openSUSE%3A12.1&lang=en&exclude_debug=true)

Also, note that the Eigen3 Library from Factory Education was causing problems sometimes, so use the one from the KDE 4.8 Extra repo

Arch Linux

You will need the following libraries from the official repositories:

- boost-libs
- curl
- hicolor-icon-theme
- libspnav
- opencascade
- python2-pivy
- python2-matplotlib
- python2-pyside
- python2-shiboken
- qtwebkit
- shared-mime-info
- xerces-c
- boost
- cmake
- coin
- desktop-file-utils
- eigen
- gcc-fortran
- swig

- xerces-c

Also, make sure to check the AUR for any missing packages that are not on the repositories, currently:

- python2-pyside-tools

Older and non-conventional distributions

On other distributions, we have very few feedback from users, so it might be harder to find the required packages. Try first locating the required libraries mentioned in Third Party Libraries (/wiki/index.php?title=Third_Party_Libraries). Beware that some of them might have a slightly different package name in your distribution (such as name, libname, name-dev, name-devel, etc...).

You also need the GNU gcc compiler (http://en.wikipedia.org/wiki/GNU_Compiler_Collection) version equal or above 3.0.0. g++ is also needed because FreeCAD is completely written in C++. During the compilation some Python scripts get executed. So the Python interpreter has to work properly. To avoid any linker problems during the build process it is also a good idea to have the library paths either in your `LD_LIBRARY_PATH` variable or in your `ld.so.conf` file. This is normally already the case in recent distributions.

For more details have also a look to *README.Linux* in your sources.

Below is additional help for a couple of libraries that might not be present in your distribution repositories

OpenCASCADE community edition (OCE)

OpenCasCade has recently been forked into a Community edition (<http://github.com/tpaviot/oce>), which is much, much easier to build. FreeCAD can use any version installed on your system, either the "official" edition or the community edition. The OCE website contains detailed build instructions.

OpenCASCADE official version

Note: You are advised to use the OpenCasCade community edition above, which is easier to build, but this one works too. Not all Linux distributions have an official OpenCASCADE package in their repositories. You have to check for yourself if one is available for your distribution. At least from Debian Lenny and Ubuntu Intrepid an official .deb package is provided. For older Debian or Ubuntu releases you may get unofficial packages from here (<http://lyre.mit.edu/~powell/opencascade>). To build your own private .deb packages follow these steps:

```
wget http://lyre.mit.edu/~powell/opencascade/opencascade_6.2.0.orig.tar.gz
wget http://lyre.mit.edu/~powell/opencascade/opencascade_6.2.0-7.dsc
wget http://lyre.mit.edu/~powell/opencascade/opencascade_6.2.0-7.diff.gz

dpkg-source -x opencascade_6.2.0-7.dsc

# Install OCC build-deps
sudo apt-get install build-essential devscripts debhelper autoconf automake libtool bison libx11-dev tc18.4-dev tk8.4-dev libgl1-mesa-dev libglu1-mesa-dev java-gcj-compat-dev libxmu-dev

#Build Opencascade packages. This takes hours and requires
# at least 8 GB of free disk space
cd opencascade-6.2.0 ; debuild

# Install the resulting library debs
sudo dpkg -i libopencascade6.2-0_6.2.0-7_i386.deb
libopencascade6.2-dev_6.2.0-7_i386.deb
```

Alternatively, you can download and compile the latest version from [opencascade.org](http://www.opencascade.org) (<http://www.opencascade.org>):

Install the package normally, be aware that the installer is a java program that requires the official java runtime edition from Sun (package name: sun-java6-jre), not the open-source java (gij) that is bundled with Ubuntu. Install it if needed:

```
sudo apt-get remove gij
sudo apt-get install sun-java6-jre
```

Be careful, if you use gij java with other things like a browser plugin, they won't work anymore. If the installer doesn't work, try:

```
java -cp path_to_file_setup.jar <-Dtemp.dir=path_to_tmp_directory> run
```

Once the package is installed, go into the "ros" directory inside the opencascade dir, and do

```
./configure --with-tcl=/usr/lib/tcl8.4 --with-tk=/usr/lib/tk8.4
```

Now you can build. Go back to the ros folder and do:

```
make
```

It will take a long time, maybe several hours.

When it is done, just install by doing

```
sudo make install
```

The library files will be copied into /usr/local/lib which is fine because there they will be found automatically by any program. Alternatively, you can also do

```
sudo checkinstall
```

which will do the same as make install but create an entry in your package management system so you can easily uninstall later. Now clean up the enormous temporary compilation files by doing

```
make clean
```

Possible error 1: If you are using OCC version 6.2, it is likely that the compiler will stop right after the beginning of the "make" operation. If it happens, edit the "configure" script, locate the CXXFLAGS="\$CXXFLAGS " statement, and replace it by CXXFLAGS="\$CXXFLAGS -ffriend-injection -fpermissive". Then do the configure step again.

Possible error 2: Possibly several modules (WOKSH, WOKLibs, TKWOKTcl, TKViewerTest and TKDraw) will complain that they couldn't find the tcl/tk headers. In that case, since the option is not offered in the configure script, you will have to edit manually the makefile of each of those modules: Go into adm/make and into each of the bad modules folders. Edit the Makefile, and locate the lines CSF_TclLibs_INCLUDES = -I/usr/include and CSF_TclTkLibs_INCLUDES = -I/usr/include and add /tcl8.4 and /tk8.4 to it so they read: CSF_TclLibs_INCLUDES = -I/usr/include/tcl8.4 and CSF_TclTkLibs_INCLUDES = -I/usr/include/tk8.4

SoQt

The SoQt library must be compiled against Qt4, which is the case in most recent distributions. But at the time of writing this article there were only SoQt4 packages for Debian itself available but not for all Ubuntu versions. To get the packages built do the following steps:

```
wget http://ftp.de.debian.org/debian/pool/main/s/soqt/soqt_1.4.1.orig.tar.gz
wget http://ftp.de.debian.org/debian/pool/main/s/soqt/soqt_1.4.1-6.dsc
wget http://ftp.de.debian.org/debian/pool/main/s/soqt/soqt_1.4.1-6.diff.gz
dpkg-source -x soqt_1.4.1-6.dsc
sudo apt-get install doxygen devscripts fakeroot debhelper libqt3-mt-dev qt3-dev-tools libqt4-opengl-dev
cd soqt-1.4.1
debuild
sudo dpkg -i libsoqt4-20_1.4.1-6_i386.deb libsoqt4-dev_1.4.1-6_i386.deb libsoqt-dev-common_1.4.1-6_i386.deb
```

If you are on a 64bit system, you will probably need to change i386 by amd64.

Pivy

Pivy is not needed to build FreeCAD or to run it, but it is needed for the 2D Drafting module to work. If you are not going to use that module, you won't need pivy. By November 2015 the obsolete version of Pivy included with FreeCAD source code will no longer compile on many systems, due to its age. If you cannot find Pivy in your distribution's packages repository or elsewhere, you can compile pivy yourself:

Pivy compilation instructions (/wiki/index.php?title=Extra_python_modules#Pivy)

Compile FreeCAD

Using cMake

cMake is a newer build system which has the big advantage of being common for different target systems (Linux, Windows, MacOSX, etc). FreeCAD is now using the cMake system as its main building system. Compiling with cMake is usually very simple and happens in 2 steps. In the first step, cMake checks that every needed programs and libraries are present on your system and sets up all that's necessary for the subsequent compilation. You are given a few alternatives detailed below, but FreeCAD comes with sensible defaults. The second step is the compiling itself, which produces the FreeCAD executable. Changing any options for cmake away from their default values, is much easier with cmake-gui or other graphical cmake applications than with cmake on the command line, as the graphical applications will give you interactive feed back.

Since FreeCAD is a heavy application, compiling can take a bit of time (about 10 minutes on a fast machine, 30 minutes (or more) on a slow one)

In-source building

If you are unsure then, due to its limitations, do not make an in-source build, create an out-of-source build as explained in the next section. However FreeCAD can be built in-source, which means that all the files resulting from the compilation stay in the same folder as the source code. This is fine if you are just looking at FreeCAD, and want to be able to remove it easily by just deleting that folder. But in case you are planning to compile it often, you are advised to make an out-of-source build, which offers many more advantages. The following commands will compile FreeCAD:

```
$ cd freecad (the folder where you cloned the freecad source)
```

If you want to use your system's copy of Pivy, which you most commonly will, then if not on Linux, set the compiler flag to use the correct pivy (via `FREECAD_USE_EXTERNAL_PIVY=1`). Using external Pivy became the default for Linux, during development of FreeCAD 0.16, so it does not need to be manually set when compiling this version onwards, on Linux. Also, set the build type to Debug if you want a debug build or Release if not. A Release build will run much faster than a Debug build. Sketcher becomes very slow with complex sketches if your FreeCAD is a Debug build. (NOTE: the space and "." after the cmake flags are CRITICAL!):

For a Debug build

```
$ cmake -DFREECAD_USE_EXTERNAL_PIVY=1 -DCMAKE_BUILD_TYPE=Debug .
$ make
```

Or for a Release build

```
$ cmake -DFREECAD_USE_EXTERNAL_PIVY=1 -DCMAKE_BUILD_TYPE=Release .
$ make
```

Your FreeCAD executable will then reside in the "bin" folder, and you can launch it with:

```
$ ./bin/FreeCAD
```

How to repair your source code directory after accidentally running an in-source build.

This is a method, using Git, to repair your source code directory after accidentally running an in-source build.

```
1) delete everything in your source base directory EXCEPT the hidden .git folder
2) In terminal 'git reset --hard HEAD'
//any remnants of an 'in source' build will be gone.
3) delete everything from your 'out of source' build directory and start over again with cmake
and a full new clean build.
```

Out-of-source build

If you intend to follow the fast evolution of FreeCAD, building in a separate folder is much more convenient. Every time you update the source code, cMake will then intelligently distinguish which files have changed, and recompile only what is needed. Out-of-source builds are specially handy when using the Git system, because you can easily try other branches without confusing the build system. To build out-of-source, simply create a build directory, distinct from your FreeCAD source folder, and, from the build folder, point cMake (or if using cmake-gui replace "cmake" in the code below with "cmake-gui") to the source folder:

```
mkdir freecad-build
cd freecad-build
cmake ../freecad (or whatever the path is to your FreeCAD source folder)
make
```

The FreeCAD executable will then reside in the "bin" directory (within your freecad-build directory).

Configuration options

There are a number of experimental or unfinished modules you may have to build if you want to work on them. To do so, you need to set the proper options for the configuration phase. Do it either on the command line, passing `-D <var>:<type>=<value>` options to cMake or using one of the availables gui-frontends (eg for Debian, packages `cmake-qt-gui` or `cmake-curses-gui`). Changing any options for cmake away from their default values, is much easier with `cmake-gui` or other graphical cmake applications than with `cmake` on the command line, as they will give you interactive feed back.

As an example, to configure FreeCAD with the Assembly module built just tick the box in a `cmake gui` application (e.g. `cmake-gui`) or on the command line issue:

```
cmake -D FREECAD_BUILD_ASSEMBLY:BOOL=ON ''path-to-freecad-root''
```

Possible options are listed in FreeCAD's root `CmakeLists.txt` file.

Qt designer plugin

If you want to develop Qt stuff for FreeCAD, you'll need the Qt Designer plugin that provides all custom widgets of FreeCAD. Go to

```
freecad/src/Tools/plugins/widget
```

So far we don't provide a makefile -- but calling

```
qmake plugin.pro
```

creates it. Once that's done, calling

```
make
```

will create the library `libFreeCAD_widgets.so`. To make this library known to Qt Designer you have to copy the file to `$QTDIR/plugin/designer`

Doxygen

If you feel bold enough to dive in the code, you could take advantage to build and consult Doxygen generated FreeCAD's Source documentation (`/wiki/index.php?title=Source_documentation`)

Making a debian package

If you plan to build a Debian package out of the sources you need to install those packages first:

```
dh-make
devscripts

#optional, used for checking if packages are standard-compliant
lintian
```

To build a package open a console, simply go to the FreeCAD directory and call

```
debuild
```

Once the package is built, you can use lintian to check if the package contains errors

```
#replace by the name of the package you just created
lintian your-fresh-new-freecad-package.deb
```

Troubleshooting

Note for 64bit systems

When building FreeCAD for 64-bit there is a known issue with the OpenCASCADE 64-bit package. To get FreeCAD running properly you might need to run the `./configure` script with the additional define `_OCC64` set:

```
./configure CXXFLAGS="-D_OCC64"
```

For Debian based systems this workaround is not needed when using the prebuilt package because there the OpenCASCADE package is built to set internally this define. Now you just need to compile FreeCAD the same way as described above.

Automatic build scripts

Here is all what you need for a complete build of FreeCAD. It's a one-script-approach and works on a fresh installed distro. The commands will ask for root password (for installation of packages) and sometime to acknowledge a fingerprint for an external repository server or https-subversion repository. These scripts should run on 32 and 64 bit versions. They are written for different versions, but are also likely to run on a later version with or without major changes.

If you have such a script for your preferred distro, please send it! We will incorporate it into this article.

Ubuntu

These scripts provide a reliable way to install the correct set of dependencies required to build and run FreeCAD on Ubuntu. They make use of the FreeCAD Ubuntu PPA repositories, and should work on any version of Ubuntu targeted by the PPA. The 'daily' PPA (<https://launchpad.net/~freecad-maintainers/+archive/ubuntu/freecad-daily>) targets recent versions of Ubuntu, and the 'stable' PPA (<https://launchpad.net/~freecad-maintainers/+archive/ubuntu/freecad-stable>) targets all officially supported versions of Ubuntu.

This script installs dependencies for the daily development snapshot of FreeCAD.

```
#!/bin/sh
sudo add-apt-repository --enable-source ppa:freecad-maintainers/freecad-daily && sudo apt-get
update
# Install the dependencies needed to build FreeCAD
sudo apt-get build-dep freecad-daily
# Install the dependencies needed to run FreeCAD (and a build of FreeCAD itself)
sudo apt-get install freecad-daily
```

This script installs dependencies for the latest stable release of FreeCAD. (For Ubuntu 12.04, omit `"--enable-source"` from the `add-apt-repository` command.)

```
#!/bin/sh
sudo add-apt-repository --enable-source ppa:freecad-maintainers/freecad-stable && sudo apt-get
update
# Install the dependencies needed to build FreeCAD
sudo apt-get build-dep freecad
# Install the dependencies needed to run FreeCAD (and a build of FreeCAD itself)
sudo apt-get install freecad
```

(These scripts also install the PPA build of FreeCAD itself, as a side effect. You could then uninstall that while leaving the dependencies in place. However, leaving it installed will enable the package manager to keep the set of dependencies up to date, which is useful if you are following the development for a long time.)

After installing the dependencies, please see the generic instructions for getting the source code, running CMake, and compiling. The following script is an example of one way to do this.

```
#!/bin/sh

# checkout the latest source
git clone https://github.com/FreeCAD/FreeCAD.git freecad

# go to source dir
cd freecad

# open cmake-gui window
cmake-gui .

# build configuration
cmake .

# build FreeCAD
make
```

OpenSUSE 12.2

No external Repositories are needed to compile FreeCAD 0.13 with this release. However, there is an incompatibility with python3-devel which needs to be removed. FreeCAD can be compiled from GIT similar to in OpenSUSE 12.2

```
# install needed packages for development
sudo zypper install gcc cmake OpenCASCADE-devel libXerces-c-devel \
python-devel libqt4-devel python-qt4 Coin-devel SoQt-devel boost-devel \
libode-devel libQtWebKit-devel libeigen3-devel gcc-fortran git swig

# create new dir, and go into it
mkdir FreeCAD-Compiled
cd FreeCAD-Compiled

# get the source
git clone https://github.com/FreeCAD/FreeCAD.git free-cad

# Now you will have subfolder in this location called free-cad. It contains the source

# make another dir for compilation, and go into it
mkdir FreeCAD-Build1
cd FreeCAD-Build1

# build configuration
cmake ../free-cad

# build FreeCAD
make

# test FreeCAD
cd bin
./FreeCAD -t 0
```

Since you are using git, next time you wish to compile you do not have to clone everything, just pull from git and compile once more

```
# go into free-cad dir created earlier
cd free-cad

# pull
git pull

# get back to previous dir
cd ..

# Now repeat last few steps from before.

# make another dir for compilation, and go into it
mkdir FreeCAD-Build2
cd FreeCAD-Build2

# build configuration
cmake ../free-cad

# build FreeCAD
make

# test FreeCAD
cd bin
./FreeCAD -t 0
```

Debian Squeeze

```
# get the needed tools and libs
sudo apt-get install build-essential python libcoin60-dev libsoqt4-dev \
libxerces-c2-dev libboost-dev libboost-date-time-dev libboost-filesystem-dev \
libboost-graph-dev libboost-iostreams-dev libboost-program-options-dev \
libboost-serialization-dev libboost-signals-dev libboost-regex-dev \
libqt4-dev qt4-dev-tools python2.5-dev \
libimage-dev libopencascade-dev \
libsoqt4-dev libode-dev subversion cmake libeigen2-dev python-pivy \
libtool autotools-dev automake gfortran

# checkout the latest source
git clone https://github.com/FreeCAD/FreeCAD.git freecad

# go to source dir
cd freecad

# build configuration
cmake .

# build FreeCAD
make

# test FreeCAD
cd bin
./FreeCAD -t 0
```

Fedora 22/23/24

Posted by user [PrzemoF (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=3666>)] in the forum.

```
#!/bin/bash

#ARCH=x86_64
#ARCH=i686
ARCH=$(arch)

MAIN_DIR=FreeCAD
BUILD_DIR=build

#FEDORA_VERSION=22
FEDORA_VERSION=23
#FEDORA_VERSION=24

echo "Installing packages required to build FreeCAD"
sudo dnf -y install gcc cmake gcc-c++ boost-devel zlib-devel swig eigen3 qt-devel \
shiboken shiboken-devel pyside-tools python-pyside python-pyside-devel xerces-c \
xerces-c-devel OCE-devel smesh graphviz python-pivy python-matplotlib tbb-devel \
freeimage-devel Coin3 Coin3-devel med-devel vtk-devel

cd ~

mkdir $MAIN_DIR || { echo "~/MAIN_DIR already exist. Quitting.."; exit; }

cd $MAIN_DIR

git clone https://github.com/FreeCAD/FreeCAD.git

mkdir $BUILD_DIR || { echo "~/BUILD_DIR already exist. Quitting.."; exit; }

cd $BUILD_DIR

cmake ../FreeCAD && make
```

Updating the source code

FreeCAD development happens fast, everyday or so there are bug fixes or new features. The cmake systems allows you to intelligently update the source code, and only recompile what has changed, making subsequent compilations very fast. Updating the source code with git or subversion is very easy:

```
#Replace with the location where you cloned the source code the first time
cd freecad
#If you are using git
git pull
```

Move into the appropriate build directory and run cmake again (as cmake updates the version number data for the Help menu, ...about FreeCAD), however you do not need to add the path to source code after "cmake", just a space and a dot:

```
#Replace with the location of the build directory
cd ../freecad-build
cmake .
make
```

< [previous: CompileOnWindows \(/wiki/index.php?title=CompileOnWindows\)](#)
[next: CompileOnMac > \(/wiki/index.php?title=CompileOnMac\)](#)
[Index \(/wiki/index.php?title=Online_Help_Toc\)](#)

< translate> This page explains how to compile the latest FreeCAD source code on Mac OS X.

Prerequisites

First of all, you will need to install the following software.

Xcode Development Tools

Unless you want to use the Xcode IDE for FreeCAD development, you will only need to install the Command Line Tools. To do this on 10.9 and later, open Terminal, run the following command, and then click Install in the dialog that comes up.< /translate>

```
xcode-select --install
```

<translate> For other versions of OS X, you can get the package from the Apple developer downloads page (<https://developer.apple.com/downloads/index.action?q=xcode>) (sign in with the same Apple ID you use for other Apple services). Specifically, you will need to download Development Tools 3.2 for OS X 10.6, and Command Line Tools 4.8 for OS X 10.8.

Package Manager

You will want to use a package manager to install prerequisite software, this page gives instructions for two of the common package managers in use for OS X: Homebrew (<http://brew.sh/>) and MacPorts (<https://www.macports.org/>). It's easiest to pick one package manager for your system, and not have multiple package managers installed concurrently. Currently (October 2015), Homebrew has more up-to-date libraries relating to FreeCAD than MacPorts.

Homebrew

To install Homebrew, enter the following in Terminal:< /translate>

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

<translate>

MacPorts

To install MacPorts, follow the instructions from their website (<https://www.macports.org/install.php>)

CMake

FreeCAD uses CMake (<http://www.cmake.org/>) to build the source. Homebrew and MacPorts can install the command line version of CMake, or if you prefer using a GUI application, install the latest version from <http://www.cmake.org/download> (<http://www.cmake.org/download>).

For the command line version of CMake, from a terminal use either Homebrew:< /translate>

```
brew install cmake
```

<translate> or MacPorts: < /translate>

```
sudo port install cmake
```

<translate>

Installing the Dependencies

All of the needed libraries can be installed using either Homebrew or MacPorts.

Homebrew Dependencies

</translate>

```
brew tap homebrew/science
brew tap sanelson/freecad
brew install boost eigen freetype oce python qt pyside pyside-tools xerces-c boost-python
brew install --without-framework --without-soqt sanelson/freecad/coin
brew install --HEAD pivy
brew install --with-oc nglib
```

<translate>

MacPorts Dependencies

</translate>

```
sudo port install boost eigen3 freetype oce py27-pyside-tools xerces Coin
```

<translate>

Getting the source

In this guide, the source and build folders are created in **/Users/username/FreeCAD**, but you can of course use whatever folder you want.< /translate>

```
mkdir ~/FreeCAD
cd ~/FreeCAD
```

<translate> To get the FreeCAD source code, run:< /translate>

```
git clone https://github.com/FreeCAD/FreeCAD_sf_master.git FreeCAD-git
```

<translate>

Building FreeCAD

First, create a new folder for the build: < /translate>

```
mkdir ~/FreeCAD/build
```

<translate> Now you will need to run CMake to generate the build files. Several options will need to be given to CMake, which can be accomplished either with the CMake GUI application, or via the command line.

CMake Options

Name	Value
BUILD_ROBOT	0 (unchecked)
CMAKE_BUILD_TYPE	Release
FREECAD_USE_EXTERNAL_PIVY	1 (checked)
FREETYPE_INCLUDE_DIR_freetype2	/usr/local/include/freetype2 for Homebrew, /opt/local/include/freety for MacPorts
BUILD_FEM_NETGEN	1 (checked)
QT_QMAKE_EXECUTABLE	/opt/local/libexec/qt4/bin/qmake
FREECAD_CREATE_MAC_APP	1 (checked)

CMake GUI

Open the CMake app, and fill in the source and build folder fields. In this case, it would be `/Users/username/FreeCAD/FreeCAD-git` for the source, and `/Users/username/FreeCAD/build` for the build folder.

Next, click the **Configure** button to populate the list of configuration options. This will display a dialog asking you to specify what generator to use. Leave it at the default **Unix Makefiles**. Configuring will fail the first time because there are some options that need to be changed. Note: You will need to check the **Advanced** checkbox to get all of the options.

Set options from the table above, then click **Configure** again and then **Generate**.

CMake command line

Open a terminal, cd in to the build directory that was created above. Run `cmake` with options from the table above, following the formula `-D(Name)=(Value)`, and the path to your FreeCAD source directory as the final argument.

```
$cd ~/FreeCAD/build
$cmake -DBUILD_ROBOT="0" ...options continue... -DFREECAD_CREATE_MAC_APP ="1" ../FreeCAD-git
```

<translate>

Make

Finally, from a terminal run **make** to compile FreeCAD.

```
cd ~/FreeCAD/build
make -j3
```

<translate> The `-j` option specifies how many make processes to run at once. One plus the number of CPU cores is usually a good number to use. However, if compiling fails for some reason, it is useful to rerun make without the `-j` option, so that you can see exactly where the error occurred.

If make finishes without any errors, you can now launch FreeCAD, either from Terminal with `./bin/FreeCAD`, or by double clicking the executable in Finder.

Updating

FreeCAD development happens fast; everyday or so there are bug fixes or new features. To get these changes, run:

```
cd ~/FreeCAD/FreeCAD-git
git pull
```

<translate> And then repeat the compile step above.

Troubleshooting

Fortran

"No CMAKE_Fortran_COMPILER could be found." during configuration - Older versions of FreeCAD will need a fortran compiler installed. With Homebrew, do `brew install gcc` and try configuring again, for Macports, do `sudo port install gcc49` and give cmake the path to Fortran ie `-DCMAKE_Fortran_COMPILER=/opt/local/bin/gfortran-mp-4.9` . Or, preferably use a more current version of FreeCAD source!

OpenGL

See OpenGL on MacOS (/wiki/index.php?title=OpenGL_on_MacOS)

< previous: [CompileOnUnix](/wiki/index.php?title=CompileOnUnix)
 next: [Third Party Libraries](/wiki/index.php?title=Third_Party_Libraries) >

Index (/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate>

Overview

These are libraries which are not changed in the FreeCAD project. They are basically used unchanged as a dynamic link library (*.so or *.dll). If there is a change necessary or a wrapper class is needed, then the code of the wrapper or the changed library code has to be moved to the FreeCAD base package. The used libraries are:

If you are using Windows, consider using LibPack instead of downloading and installing all the stuff on your own.

Links

Link table

Lib name	Version needed	Link to get it
Python	>= 2.5.x	http://www.python.org/ (http://www.python.org/)
OpenCasCade	>= 5.2	http://www.opencascade.org (http://www.opencascade.org)
Qt	>= 4.1.x	http://www.qtsoftware.com (http://www.qtsoftware.com)
Coin3D	>= 2.x	http://www.coin3d.org (http://www.coin3d.org)
SoQt	>= 1.2	http://www.coin3d.org (http://www.coin3d.org)
Xerces-C++	>= 2.7.x < 3.0	http://xml.apache.org/xerces-c/ (http://xml.apache.org/xerces-c/)
Zlib	>= 1.x.x	http://www.zlib.net/ (http://www.zlib.net/)
Boost	>= 1.33.x	http://www.boost.org/ (http://www.boost.org/)
Eigen3	>= 3.0.1	http://eigen.tuxfamily.org/index.php?title=Main_Page (http://eigen.tuxfamily.org/index.php?title=Main_Page)
Shiboken	>= 1.1.2	http://shiboken.readthedocs.org/en/latest/ (http://shiboken.readthedocs.org/en/latest/)
libarea	N/A	https://github.com/danielfalck/libarea (https://github.com/danielfalck/libarea)

Details

Python

Version: 2.5 or higher

License: Python 2.5 license

You can use the source or binary from <http://www.python.org/> (<http://www.python.org/>) or use alternatively ActiveState Python from <http://www.activestate.com/> (<http://www.activestate.com/>) though it is a little bit hard to get the debug libs from ActiveState.

Description

Python is the primary scripting language and is used throughout the application. For example:

- Implement test scripts for testing on:
 - memory leaks
 - ensure presents of functionality after changes
 - post build checks
 - test coverage tests
- Macros and macro recording
- Implement application logic for standard packages
- Implementation of whole workbenches
- Dynamic loading of packages
- Implementing rules for design (Knowledge engineering)
- Doing some fancy Internet stuff like work groups and PDM
- And so on ...

Especially the dynamic package loading of Python is used to load at run time additional functionality and workbenches needed for the actual tasks. For a closer look to Python see: www.python.org Why Python you may ask. There are some reasons: So far I used different scripting languages in my professional life:

- Perl
- Tcl/Tk
- VB
- Java

Python is more OO then Perl and Tcl, the code is not a mess like in Perl and VB. Java isn't a script language in the first place and hard (or impossible) to embed. Python is well documented and easy to embed and extend. It is also well tested and has a strong back hold in the open source community.

Credits

Goes to Guido van Rossum and a lot of people made Python such a success!

OpenCasCade

Version: 5.2 or higher

License: OCTPL

OCC is a full-featured CAD Kernel. Originally, it's developed by Matra Datavision in France for the Strim (Styler) and Euclid Quantum applications and later on made Open Source. It's a really huge library and makes a free CAD application possible in the first place, by providing some packages which would be hard or impossible to implement in an Open Source project:

- A complete STEP compliant geometry kernel
- A topological data model and all needed functions to work on (cut, fuse, extrude, and so on. . .)
- Standard Import- / Export processors like STEP, IGES, VRML
- 3D and 2D viewer with selection support
- A document and project data structure with support for save and restore, external linking of documents, recalculation of design history

(parametric modeling) and a facility to load new data types as an extension package dynamically

To learn more about OpenCasCade take a look at the OpenCasCade page or <http://www.opencascade.org> (<http://www.opencascade.org>).

Qt

Version: 4.1.x or higher

License: GPL v2.0/v3.0 or Commercial (from version 4.5 on also LGPL v2.1)

I don't think I need to tell a lot about Qt. It's one of the most often used GUI toolkits in Open Source projects. For me the most important point to use Qt is the Qt Designer and the possibility to load whole dialog boxes as a (XML) resource and incorporate specialized widgets. In a CAX application the user interaction and dialog boxes are by far the biggest part of the code and a good dialog designer is very important to easily extend FreeCAD with new functionality. Further information and a very good online documentation you'll find on <http://www.qtsoftware.com> (<http://www.qtsoftware.com>).

Coin3D

Version: 2.0 or higher

License: GPL v2.0 or Commercial

Coin is a high-level 3D graphics library with a C++ Application Programming Interface. Coin uses scenegraph data structures to render real-time graphics suitable for mostly all kinds of scientific and engineering visualization applications.

Coin is portable over a wide range of platforms: any UNIX / Linux / *BSD platform, all Microsoft Windows operating system, and Mac OS X.

Coin is built on the industry-standard OpenGL immediate mode rendering library, and adds abstractions for higher-level primitives, provides 3D interactivity, immensely increases programmer convenience and productivity, and contains many complex optimization features for fast rendering that are transparent for the application programmer.

Coin is based on the SGI Open Inventor API. Open Inventor, for those who are not familiar with it, has long since become the de facto standard graphics library for 3D visualization and visual simulation software in the scientific and engineering community. It has proved it's worth over a period of more than 10 years, its maturity contributing to its success as a major building block in thousands of large-scale engineering applications around the world.

We will use OpenInventor as 3D viewer in FreeCAD because the OpenCasCade viewer (AIS and Graphics3D) has serious limitations and performance bottlenecks, especially when it goes in large-scale engineering rendering. Other things like textures or volumetric rendering are not really supported, and so on

Since Version 2.0 Coin uses a different licence model. It's not longer LGPL. They use GPL for open source and a commercial licence for closed source. That means if you want to sell your work based on FreeCAD (extension modules) you need to purchase a Coin licence!

SoQt

Version: 1.2.0 or higher

License: GPL v2.0 or Commercial

SoQt is the Inventor binding to the Qt Gui Toolkit. Unfortunately, it's not longer LGPL so we have to remove it from the code base of FreeCAD and link it as a library. It has the same licence model like Coin. And you have to compile it with your version of Qt.

Xerces-C++

Version: 2.7.0 or higher

License: Apache Software License Version 2.0

Xerces-C++ is a validating XML parser written in a portable subset of C++. Xerces-C++ makes it easy to give your application the ability to read and write XML data. A shared library is provided for parsing, generating, manipulating, and validating XML documents.

Xerces-C++ is faithful to the XML 1.0 recommendation and many associated standards (see Features below).

The parser provides high performance, modularity, and scalability. Source code, samples and API documentation are provided with the parser. For portability, care has been taken to make minimal use of templates, no RTTI, and minimal use of #ifdefs.

The parser is used for saving and restoring parameters in FreeCAD.

zlib

Version: 1.x.x

License: zlib License

zlib is designed to be a free, general-purpose, legally unencumbered -- that is, not covered by any patents -- lossless data-compression library for use on virtually any computer hardware and operating system. The zlib data format is itself portable across platforms. Unlike the LZW compression method used in Unix compress(1) and in the GIF image format, the compression method currently used in zlib essentially never expands the data. (LZW can double or triple the file size in extreme cases.) zlib's memory footprint is also independent of the input data and can be reduced, if necessary, at some cost in compression.

Boost

Version: 1.33.x

License: Boost Software License - Version 1.0

The Boost C++ libraries are a collection of peer-reviewed, open source libraries that extend the functionality of C++. The libraries are licensed under the Boost Software License, designed to allow Boost to be used with both open and closed source projects. Many of Boost's founders are on the C++ standard committee and several Boost libraries have been accepted for incorporation into the Technical Report 1 of C++0x.

The libraries are aimed at a wide range of C++ users and application domains. They range from general-purpose libraries like SmartPtr, to OS Abstractions like FileSystem, to libraries primarily aimed at other library developers and advanced C++ users, like MPL.

In order to ensure efficiency and flexibility, Boost makes extensive use of templates. Boost has been a source of extensive work and research into generic programming and meta-programming in C++.

See: <http://www.boost.org/> (<http://www.boost.org/>) for details.

libarea

Version: N/A

License: New BSD (BSD 3-Clause)

Area is a piece of software created by Dan Heeks for HeeksCNC. It is employed as a library for generation of CAM related operations in the Path Workbench.

LibPack

LibPack is a convenient package with all the above libraries packed together. It is currently available for the Windows platform on the Download (</wiki/index.php?title=Download>) page! If you're working under Linux you don't need a LibPack, instead of you should make use of the package repositories of your Linux distribution.

FreeCADLibs7.x Changelog

- Using QT 4.5.x and Coin 3.1.x

- Eigen template lib for Robot added
- SMESH experimental

< previous: CompileOnMac (/wiki/index.php?title=CompileOnMac)
 next: Third Party Tools > (/wiki/index.php?title=Third_Party_Tools)
 Index (/wiki/index.php?title=Online_Help_Toc)
 </translate>

< translate>

Tool Page

For every serious software development you need tools. Here is a list of tools we use to develop FreeCAD:

Platform independend tools

Qt-Toolkit

The Qt-toolkit is a state of the art, plattform independend user interface design tool. It is contained in the LibPack (/wiki/index.php?title=Third_Party_Libraries) of FreeCAD, but can also be downloaded at Qt project (<http://qt-project.org/downloads>).

InkScape

Great vector drawing programm. Adhers to the SVG standard and is used to draw Icons and Pictures. Get it at www.inkscape.org (<http://www.inkscape.org>).

Doxygen

A very good and stable tool to generate source documentation from the .h and .cpp files.

The Gimp

Not much to say about the Gnu Image Manipulation Program. Besides it can handle .xpm files which is a very convenient way to handle Icons in QT Programms. XPM is basicly C-Code which can be compiled into a programme.

Get the GIMP here: www.gimp.org (<http://www.gimp.org/>)

Tools on Windows

Visual Studio 8 Express

Although VC8 is for C++ development not really a step forward since VisualStudio 6 (IMO a big step back), its a free development system on Windows. For native Win32 applications you need to download the PlatformSDK from M\$.

So the Express edition is hard to find. But you might try this link (<http://msdn.microsoft.com/vstudio/express/visualc/default.aspx>)

CamStudio

Is a Open Source tool to record Screencasts (Webcasts). Its a very good tool to create tutorials by recording them. Its far not so boring as writing documentation.

See camstudio.org (<http://camstudio.org/>) for details.

Tortoise SVN

This is a very great tool. It makes using Subversion (our version control system on sf.net) a real pleasure. You can throught out the explorer integration, easily manage Revisions, check on Diffs, resolve Confilct, make branches, and so on.... The commit dialog itself is a piece of art. It gives you

an overview over your changed files and allows you to put them in the commit or not. That makes it easy to bundle the changes to logical units and give them an clear commit message.

You find ToroiseSVN on tortoisesevn.tigris.org (<http://tortoisesevn.tigris.org/>).

StarUML

A full featured Open Source UML programm. It has a lot of features of the big ones, including reverse engenieng C++ source code....

Download here: staruml.sourceforge.net
(<http://staruml.sourceforge.net/en/>)

Tools on Linux

TODO

< previous: Third Party Libraries (/wiki/index.php?title=Third_Party_Libraries)

next: Start up and Configuration > (/wiki/index.php?title=Start_up_and_Configuration)

Index (/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate> This page shows the different ways to start FreeCAD and the most important configuration features.

Starting FreeCAD from the Command line

FreeCAD can be started normally, by double-clicking on its desktop icon or selecting it from the start menu, but it can also be started directly from the command line. This allows you to change soem of the default startup options.

Command line options

The command line options are subject of frequent changes, therefore it is a good idea to check the current options by typing: < /translate>

```
FreeCAD --help
```

<translate> From the response you can read the possible parameters:< /translate>

```
Usage:
FreeCAD [options] File1 File2 .....
Allowed options:

Generic options:
-v [ --version ]      print version string
-h [ --help ]        print help message
-c [ --console ]     start in console mode
--response-file arg  can be specified with '@name', too
```

```
Configuration:
-l [ --write-log ] arg  write a log file to default location(Run FreeCAD --h to see default
location)
--log-file arg         Unlike to --write-log this allows to log to an arbitrary file
-u [ --user-cfg ] arg  User config file to load/save user settings
-s [ --system-cfg ] arg System config file to load/save system settings
-t [ --run-test ] arg  test level
-M [ --module-path ] arg additional module paths
-P [ --python-path ] arg additional python paths
```

EX: (Windows)

```
"C:\Program Files\FreeCAD 0.14\bin\FreeCAD.exe" -M "N:\FreeCAD\Mod\Draft" -M "N:\FreeCAD\Mod\P
art" -M "N:\FreeCAD\Mod\Drawing" -u "N:\FreeCAD\Config\user.cfg" -s "N:\FreeCAD\Config\system.
cfg"
```

<translate>

Response and config files

FreeCAD can read some of these options from a config file. This file must be in the bin path and must be named FreeCAD.cfg. Be aware that options specified in the command line override the config file!

Some operating system have very low limit of the command line length. The common way to work around those limitations is using response files. A response file is just a configuration file which uses the same syntax as the command line. If the command line specifies a name of response file to use, it's loaded and parsed in addition to the command line:< /translate>

```
FreeCAD @ResponseFile.txt
```

<translate> or: < /translate>

```
FreeCAD --response-file=ResponseFile.txt
```

<translate>

Hidden options

There are a couple of options not visible to the user. These options are e.g. the X-Window parameters parsed by the Windows system:

- -display display, sets the X display (default is \$DISPLAY).
- -geometry geometry, sets the client geometry of the first window that is shown.
- -fn or -font font, defines the application font. The font should be specified using an X logical font description.
- -bg or -background color, sets the default background color and an application palette (light and dark shades are calculated).
- -fg or -foreground color, sets the default foreground color.
- -btn or -button color, sets the default button color.
- -name name, sets the application name.
- -title title, sets the application title.
- -visual TrueColor, forces the application to use a TrueColor visual on an 8-bit display.
- -ncols count, limits the number of colors allocated in the color cube on an 8-bit display, if the application is using the QApplication::ManyColor color specification. If count is 216 then a 6x6x6 color cube is used (i.e. 6 levels of red, 6 of green, and 6 of blue); for other values, a cube approximately proportional to a 2x3x1 cube is used.
- -cmap, causes the application to install a private color map on an 8-bit display.

Running FreeCAD without User Interface

FreeCAD normally starts in GUI mode, but you can also force it to start in console mode by typing: < /translate>

```
FreeCAD -c
```

<translate> from the command line. In console mode, no user interface will be displayed, and you will be presented with a python interpreter prompt. From that python prompt, you have the same functionality as the python interpreter that runs inside the FreeCAD GUI, and normal access to all modules and plugins of FreeCAD, excepted the FreeCADGui module. Be aware that modules that depend on FreeCADGui might also be unavailable.

Running FreeCAD as a python module

FreeCAD can also be used to run as a python module inside other applications that use python or from an external python shell. For that, the host python application must know where your FreeCAD libs reside. The best way to obtain that is to temporarily append FreeCAD's lib path to the `sys.path` variable. The following code typed from any python shell will import FreeCAD and let you run it the same way as in console mode:

</translate>

```
import sys
sys.path.append("path/to/FreeCAD/lib") # change this by your own FreeCAD lib path
import FreeCAD
```

<translate>

Once FreeCAD is loaded, it is up to you to make it interact with your host application in any way you can imagine!

The Config set

On every Startup FreeCAD examines its surrounding and the command line parameters. It builds up a **configuration set** which holds the essence of the runtime information. This information is later used to determine the place where to save user data or log files. It is also very important for post mortem analyzes. Therefore it is saved in the log file.

User related information

User config entries

Config var name	Synopsis	Example M\$	Example Pos
UserAppData	Path where FreeCAD stores User Related application data.	C:\Documents and Settings\username\Application Data\FreeCAD	/home/user
UserParameter	File where FreeCAD stores User Related application data.	C:\Documents and Settings\username\Application Data\FreeCAD\user.cfg	/home/user
SystemParameter	File where FreeCAD stores Application Related data.	C:\Documents and Settings\username\Application Data\FreeCAD\system.cfg	/home/user
UserHomePath	Home path of the current user	C:\Documents and Settings\username\My Documents	/home/user

Command line arguments

User config entries

Config var name	Synopsis	Example
LoggingFile	1 if the logging is switched on	
LoggingFileName	File name where the log is placed	C:\Documents and Settings\username\Application Data\FreeCAD\FreeCAD.log
RunMode	This indicates how the main loop will work. "Script" means that the given script is called and then exit. "Cmd" runs the command line interpreter. "Internal" runs an internal script. "Gui" enters the Gui event loop. "Module" loads a given python module.	"Cmd"
FileName	Meaning depends on the RunMode	
ScriptFileName	Meaning depends on the RunMode	
Verbose	Verbosity level of FreeCAD	"" or "strict"

OpenFileCount	Holds the number of files opened through command line arguments	"12"
AdditionalModulePaths	Holds the additional Module paths given in the cmd line	"extraModules/"

System related

User config entries

Config var name	Synopsis	Example M\$	Example Posix (Linux)
AppHomePath	Path where FreeCAD is installed	c:/Progam Files/FreeCAD_0.7	/user/local/FreeCAD_0.7
PythonSearchPath	Holds a list of paths which python search modules. This is at startup can change during execution		

Some libraries need to call system environment variables. Sometimes when there is a problem with a FreeCAD installation, it is because some environment variable is absent or set wrongly. Therefore, some important variables get duplicated in the Config and saved in the log file.

Python related environment variables:< /translate>

- PYTHONPATH
- PYTHONHOME
- TCL_LIBRARY
- TCLLIBPATH

<translate> **OpenCascade related environment variables:**< /translate>

- CSF_MDTVFontDirectory
- CSF_MDTVTexturesDirectory
- CSF_UnitsDefinition
- CSF_UnitsLexicon
- CSF_StandardDefaults
- CSF_PluginDefaults
- CSF_LANGUAGE
- CSF_SHMessage
- CSF_XCAFDefaults
- CSF_GraphicShr
- CSF_IGESDefaults
- CSF_STEPDefaults

<translate> **System related environment variables:**< /translate>

- PATH

<translate>

Build related information

The table below shows the available informations about the Build version. Most of it comes from the Subversion repository. This stuff is needed to exactly rebuild a version!

User config entries

Config var name	Synopsis	Example
BuildVersionMajor	Major Version number of the Build. Defined in src/Build/Version.h.in	0
BuildVersionMinor	Minor Version number of the Build. Defined in src/Build/Version.h.in	7
BuildRevision	SVN Repository Revision number of the src in the Build. Generated by SVN	356
BuildRevisionRange	Range of differnt changes	123-356
BuildRepositoryURL	Repository URL	https://free-cad.svn.sourceforge.net/svnroc cad/trunk/src (https: cad.svn.sourceforge.net/svnroc cad/trunk/src)
BuildRevisionDate	Date of the above Revision	2007/02/03 22:21:18
BuildScrClean	Indicates if the source was changed after checkout	Src modified
BuildScrMixed		Src not mixed

Branding related

These Config entries are related to the branding mechanism of FreeCAD. See [Branding \(/wiki/index.php?title=Branding\)](/wiki/index.php?title=Branding) for more details.

User config entries

Config var name	Synopsis	Example
ExeName	Name of the build Executable file. Can diver from FreeCAD if a different main.cpp is used.	FreeCAD.exe
ExeVersion	Over all Version shows up at start time	V0.7
AppIcon	Icon which is used for the Executable, shows in Application MainWindow.	"FCIcon"
ConsoleBanner	Banner which is prompted in console mode	
SplashPicture	Name of the Icon used for the Splash Screen	"FreeCADSplasher"
SplashAlignment	Alignment of the Text in the Splash dialog	Left"
SplashTextColor	Color of the splasher Text	"#000000"
StartWorkbench	Name of the Workbench which get started automatically after Startup	"Part design"
HiddenDockWindow	List of dockwindows (separated by a semicolon) which will be disabled	"Property editor"

< previous: [Third Party Tools \(/wiki/index.php?title=Third_Party_Tools\)](/wiki/index.php?title=Third_Party_Tools)

next: [FreeCAD Build Tool > \(/wiki/index.php?title=FreeCAD_Build_Tool\)](/wiki/index.php?title=FreeCAD_Build_Tool)

[Index \(/wiki/index.php?title=Online_Help_Toc\)](/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate> The **FreeCAD build tool** or **fbct** is a python script located at< /translate>

```
trunc/src/Tools/fbct.py
```

<translate> It can be used to simplify some frequent tasks in building, distributing and extending FreeCAD.

Usage

With Python ([http://en.wikipedia.org/wiki/Python_\(programming_language\)](http://en.wikipedia.org/wiki/Python_(programming_language))) correctly installed, *fbct* can be invoked by the command < /translate>

```
python fbct.py
```

<translate> It displays a menu, where you can select the task you want to

use it for:< /translate>

```
FreeCAD Build Tool
Usage:
  fcbt <command name> [command parameter]
possible commands are:
- DistSrc      (DS)  Build a source Distr. of the current source tree
- DistBin      (DB)  Build a binary Distr. of the current source tree
- DistSetup    (DI)  Build a Setup Distr. of the current source tree
- DistSetup    (DUI) Build a User Setup Distr. of the current source tree
- DistAll      (DA)  Run all three above modules
- NextBuildNumber (NBN) Increase the Build Number of this Version
- CreateModule (CM)  Insert a new FreeCAD Module in the module directory

For help on the modules type:
fcbt <command name> ?
```

<translate> At the input prompt enter the abbreviated command you want to call. For example type "CM" for creating a module (/wiki/index.php?title=Module_Creation).

DistSrc

The command "DS" **creates a source distribution** of the current source tree.

DistBin

The command "DB" **creates a binary distribution** of the current source tree.

DistSetup

The command "DI" **creates a setup distribution** of the current source tree.

DistSetup

The command "DUI" **creates a user setup distribution** of the current source tree.

DistAll

The command "DA" executes "DS", "DB" and "DI" in sequence.

NextBuildNumber

The "NBN" command **increments the build number** to create a new release version of FreeCAD.

CreateModule

The "CM" command creates a new application module (/wiki/index.php?title=Module_Creation).

< previous: [Start up and Configuration](/wiki/index.php?title=Start_up_and_Configuration) (/wiki/index.php?title=Start_up_and_Configuration)

next: [Module Creation](/wiki/index.php?title=Module_Creation) > (/wiki/index.php?title=Module_Creation)

[Index](/wiki/index.php?title=Online_Help_Toc) (/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate> Adding new modules and workbenches in FreeCAD is very easy. We call module any extension of FreeCAD, while a workbench is a special GUI configuration that groups some toolbars and menus. Usually you create a new module which contains its own workbench.

Modules can be programmed in C++ or in python, or in a mixture of both, but the module init files must be in python. Setting up a new module with those init files is easy, and can be done either manually or with the FreeCAD build tool.

Using the FreeCAD Build tool

Creating a new application module in FreeCAD is rather simple. In the FreeCAD development tree exists the *FreeCAD Build Tool* ([/wiki/index.php?title=FreeCAD_Build_Tool](http://wiki/index.php?title=FreeCAD_Build_Tool)) (fcbt) that does the most important things for you. It is a Python ([http://en.wikipedia.org/wiki/Python_\(programming_language\)](http://en.wikipedia.org/wiki/Python_(programming_language))) script located under

```
trunk/src/Tools/fcbt.py
```

When your python interpreter is correctly installed you can execute the script from a command line with

```
python fcbt.py
```

It will display the following menu:

```
FreeCAD Build Tool
Usage:
  fcbt <command name> [command parameter]
possible commands are:
- DistSrc      (DS)  Build a source Distr. of the current source tree
- DistBin      (DB)  Build a binary Distr. of the current source tree
- DistSetup    (DI)  Build a Setup Distr. of the current source tree
- DistSetup    (DUI) Build a User Setup Distr. of the current source tree
- DistAll      (DA)  Run all three above modules
- BuildDoc     (BD)  Create the documentation (source docs)
- NextBuildNumber (NBN) Increase the Build Number of this Version
- CreateModule (CM)  Insert a new FreeCAD Module in the module directory

For help on the modules type:
fcbt <command name> ?
```

At the command prompt enter *CM* to start the creation of a module:

```
Insert command: 'CM'
```

You are now asked to specify a name for your new module. Lets call it *TestMod* for example:

```
Please enter a name for your application: 'TestMod'
```

After pressing *enter* fcbt starts copying all necessary files for your module in a new folder at

```
trunk/src/Mod/TestMod/
```

Then all files are modified with your new module name. The only thing you need to do now is to add the two new projects "appTestMod" and "appTestModGui" to your workspace (on Windows) or to your makefile targets (unix). Thats it!

Setting up a new module manually

You need two things to create a new module:

- A new **folder** in the FreeCAD Mod folder (either in InstalledPath/FreeCAD/Mod or in UserPath/.FreeCAD/Mod). You can name it as you like.
- Inside that folder, an **InitGui.py** file. That file will be executed automatically on FreeCAD start (for ex, put a print("hello world") inside)

Additionally, you can also put an **Init.py** file. The difference is, the InitGui.py file is loaded only when FreeCAD runs in GUI mode, the Init.py file is loaded always. But if we are going to make a workbench, we'll put it in InitGui.py, because workbenches are used only in GUI mode, of course.

Creating a new workbench

Inside the InitGui.py file, one of the first thing you will want to do is to define a workbench. Here is a minimal code that you can use:

< translate>

Test First

Before you go through the pain of debugging use the Test framework (</wiki/index.php?title=Testing>) to check if the standard tests work properly. If they do not run complete there is possibly a broken installation.

Command Line

The *debugging* of FreeCAD is supported by a few internal mechanisms. The command line version of FreeCAD provides some options for debugging support.

These are the currently recognized options in FreeCAD 0.15:

Generic options:

```
-v [ --version ]      Prints version string
-h [ --help ]        Prints help message
-c [ --console ]     Starts in console mode
--response-file arg  Can be specified with '@name', too
```

Configuration:

```
-l [ --write-log ]    Writes a log file to:
                    /home/graphos/.FreeCAD/FreeCAD.log
--log-file arg       Unlike to --write-log this allows to log to an
                    arbitrary file
-u [ --user-cfg ] arg User config file to load/save user settings
-s [ --system-cfg ] arg System config file to load/save system settings
-t [ --run-test ] arg Test level
-M [ --module-path ] arg Additional module paths
-P [ --python-path ] arg Additional python paths
```

Generating a Backtrace

If you are running a version of FreeCAD from the bleeding edge of the development curve, it may "crash". You can help solve such problems by providing the developers with a "backtrace". To do this, you need to be running a "debug build" of the software. "Debug build" is a parameter that is set at compile time, so you'll either need to compile FreeCAD yourself, or obtain a pre-compiled "debug" version.

For Linux

Prerequisites:

- software package gdb installed
- a debug build of FreeCAD
- a FreeCAD model that causes a crash

Steps: Enter the following in your terminal window: < /translate>

```
$ cd FreeCAD/bin
$ gdb FreeCAD
```

GNUdebugger will output some initializing information. The (gdb) shows GNUDebugger is running in the terminal, now input:

```
(gdb) handle SIG33 noprint nostop
(gdb) run
```

<translate> FreeCAD will now start up. Perform the steps that cause FreeCAD to crash or freeze, then enter in the terminal window:

```
(gdb) bt
```

This will generate a lengthy listing of exactly what the program was doing when it crashed or froze. Include this with your problem report.

Python Debugging

Here is an example of using winpdb inside FreeCAD:

1. Run winpdb and set the password (e.g. test)
2. Create a Python file with this content

</translate>

```
import rpdb2
rpdb2.start_embedded_debugger("test")
import FreeCAD
import Part
import Draft
print "hello"
print "hello"
import Draft
points=[FreeCAD.Vector(-3.0, -1.0, 0.0), FreeCAD.Vector(-2.0, 0.0, 0.0)]
Draft.makeWire(points, closed=False, face=False, support=None)
```

<translate>

1. Start FreeCAD and load the above file into FreeCAD
2. Press F6 to execute it
3. Now FreeCAD will become unresponsive because the Python debugger is waiting
4. Switch to the Windpdb GUI and click on "Attach". After a few seconds an item "<Input>" appears where you have to double-click
5. Now the currently executed script appears in Winpdb.
6. Set a break at the last line and press F5
7. Now press F7 to step into the Python code of Draft.makeWire

< previous: Module Creation (/wiki/index.php?title=Module_Creation)
 Index next: Testing > (/wiki/index.php?title=Testing)
 (/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate> FreeCAD comes with an extensive testing framework. The testing bases on a set of Python scripts which are located in the test module.

Introduction

This is the list of test apps as of 0.15 Git 4207:

TestAPP.All

Add test function

BaseTests

Add test function

UnitTests

Add test function

Document

Add test function

UnicodeTests

Add test function

MeshTestsApp

Add test function

TestSketcherApp

Add test function

TestPartApp

Add test function

TestPartDesignApp

Add test function

Workbench

Add test function

Menu

Add test function

Menu.MenuDeleteCases

Add test function

Menu.MenuCreateCases

Add test function

< previous: [Debugging \(/wiki/index.php?title=Debugging\)](/wiki/index.php?title=Debugging) [Index](#)
next: [Branding > \(/wiki/index.php?title=Branding\)](/wiki/index.php?title=Branding)
(/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate> This article describes the **Branding** of FreeCAD. Branding means to start your own application on base of FreeCAD. That can be only your own executable or splash screen (/wiki/index.php?title=Splash_screen) till a complete reworked program. On base of the flexible architecture of FreeCAD it's easy to use it as base for your own special purpose program.

General

Most of the branding is done in the **MainCmd.cpp** *or* **MainGui.cpp**. These Projects generate the executable files of FreeCAD. To make your own Brand just copy the Main or MainGui projects and give the executable an own name, e.g. FooApp.exe. The most important settings for a new look can be made in one place in the main() function. Here is the code section that controls the branding:

</translate>

```

int main( int argc, char ** argv )
{
    // Name and Version of the Application
    App::Application::Config()["ExeName"] = "FooApp";
    App::Application::Config()["ExeVersion"] = "0.7";

    // set the banner (for logging and console)
    App::Application::Config()["CopyrightInfo"] = sBanner;
    App::Application::Config()["AppIcon"] = "FooAppIcon";
    App::Application::Config()["SplashScreen"] = "FooAppSplasher";
    App::Application::Config()["StartWorkbench"] = "Part design";
    App::Application::Config()["HiddenDockWindow"] = "Property editor";
    App::Application::Config()["SplashAlignment"] = "Bottom|Left";
    App::Application::Config()["SplashTextColor"] = "#000000"; // black

    // Inits the Application
    App::Application::Config()["RunMode"] = "Gui";
    App::Application::init(argc,argv);

    Gui::BitmapFactory().addXPM("FooAppSplasher", ( const char** ) splash_screen);

    Gui::Application::initApplication();
    Gui::Application::runApplication();
    App::Application::destruct();

    return 0;
}

```

<translate> The first Config entry defines the program name. This is not the executable name, which can be changed by renaming or by compiler settings, but the name that is displayed in the task bar on windows or in the program list on Unix systems.

The next lines define the Config entries of your FooApp Application. A description of the Config and its entries you find in Start up and Configuration (/wiki/index.php?title=Start_up_and_Configuration).

Images

Image resources are compiled into FreeCAD using Qt's resource system (<http://qt-project.org/doc/qt-4.8/resources.html>). Therefore you have to write a .qrc file, an XML-based file format that lists image files on the disk but also any other kind of resource files. To load the compiled resources inside the application you have to add a line< /translate>

```
Q_INIT_RESOURCE(FooApp);
```

<translate> into the main() function. Alternatively, if you have an image in XPM format you can directly include it into your main.cpp and add the following line to register it:< /translate>

```
Gui::BitmapFactory().addXPM("FooAppSplasher", ( const char** ) splash_screen);
```

<translate>

Branding XML

In FreeCAD there is also a method supported without writing a customized main() function. For this method you must write a file name called branding.xml and put it into the installation directory of FreeCAD. Here is an example with all supported tags:< /translate>

Translating with Qt-Linguist (old way)

The following information doesn't need to be used anymore and will likely become obsolete.
It is being kept here so that programmers may familiarize themselves with how it works.

- Open all of the language folders of FreeCAD shown below
- Verify that a .ts file with your language code doesn't exist ("fr" for french, "de" for german, etc...)
- If it exists, you can download that file, if you want to modify/review/better the translation (click the file, then download)
- If it doesn't exist, download the .ts file without language code (or any other .ts available, it will work too)
- Rename that file with your language code
- Open it with the Qt-Linguist program
- Start translating (Qt Linguist is very easy to use)
- Once it's completely done, save your file
- send the files to us (http://www.freecadweb.org/tracker/main_page.php) so we can include them in the freecad source code so they benefit other users too.

Available translation files

- FreeCAD main GUI (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Gui/Language/>)
- Complete Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Complete/Gui/Resources/translations/>)
- Drawing Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Drawing/Gui/Resources/translations/>)
- Draft Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Draft/Resources/translations/>)
- Reverse Engineering Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/ReverseEngineering/Gui/Resources/translations/>)
- FEM Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Fem/Gui/Resources/translations/>)
- Robot Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Robot/Gui/Resources/translations/>)
- Image Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Image/Gui/Resources/translations/>)
- Sketcher Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Sketcher/Gui/Resources/translations/>)
- Mesh Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Mesh/Gui/Resources/translations/>)
- Test Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Test/Gui/Resources/translations/>)
- Points Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Points/Gui/Resources/translations/>)
- Raytracing Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Raytracing/Gui/Resources/translations/>)
- Part Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Part/Gui/Resources/translations/>)
- PartDesign Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/PartDesign/Gui/Resources/translations/>)

- Assembly Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Assembly/Gui/Resources/translations/>)
- MeshPart Workbench (<http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/MeshPart/Gui/Resources/translations/>)

Preparing your own modules/applications for translation

Prerequisites

To localise your application module you need to helpers that come with *Qt*. You can download them from the Trolltech-Website (<http://www.trolltech.com/products/qt/downloads>), but they are also contained in the LibPack (/wiki/index.php?title=Third_Party_Libraries):

qmake

Generates project files

lupdate

Extracts or updates the original texts in your project by scanning the source code

Qt-Linguist

The *Qt-Linguist* is very easy to use and helps you translating with nice features like a phrase book for common sentences.

Project Setup

To start the localisation of your project go to the GUI-Part of you module and type on the command line:< /translate>

```
qmake -project
```

<translate> This scans your project directory for files containing text strings and creates a project file like the following example:< /translate>

```
#####
# Automatically generated by qmake (1.06c) Do 2. Nov 14:44:21 2006
#####

TEMPLATE = app
DEPENDPATH += .\Icons
INCLUDEPATH += .

# Input
HEADERS += ViewProvider.h Workbench.h
SOURCES += AppMyModGui.cpp \
          Command.cpp \
          ViewProvider.cpp \
          Workbench.cpp
TRANSLATIONS += MyMod_de.ts
```

<translate>

You can manually add files here. The section `TRANSLATIONS` contains a list of files with the translation for each language. In the above example *MyMod_de.ts* is the german translation.

Now you need to run `lupdate` to extract all string literals in your GUI. Running `lupdate` after changes in the source code is always safe since it never deletes strings from your translations files. It only adds new strings.

Now you need to add the `.ts`-files to your VisualStudio project. Specify the following custom build method for them:< /translate>

```
python ..\..\..\Tools\qembed.py "$(InputDir)\$(InputName).ts"
    "$(InputDir)\$(InputName).h" "$(InputName)"
```

<translate> Note: Enter this in one command line, the line break is only for

layout purpose.

By compiling the `.ts`-file of the above example, a header file `MyMod_de.h` is created. The best place to include this is in `App<Modul>Gui.cpp`. In our example this would be `AppMyModGui.cpp`. There you add the line `</translate>`

```
new Gui::LanguageProducer("Deutsch", <Modul>_de_h_data, <Modul>_de_h_len);
```

`</translate>` to publish your translation in the application.

Setting up python files for translation

To ease localization for the py files you can use the tool "pylupdate4" which accepts one or more py files. With the `-ts` option you can prepare/update one or more `.ts` files. For instance to prepare a `.ts` file for French simply enter into the command line: `</translate>`

```
pylupdate4 *.py -ts YourModule_fr.ts
```

`</translate>` the pylupdate tool will scan your `.py` files for `translate()` or `tr()` functions and create a `YourModule_fr.ts` file. That file can be translated with QLinguist and a `YourModule_fr.qm` file produced from QLinguist or with the command `</translate>`

```
lrelease YourModule_fr.ts
```

`</translate>` Beware that the pylupdate4 tool is not very good at recognizing `translate()` functions, they need to be formatted very specifically (see the Draft module files for examples). Inside your file, you can then setup a translator like this (after loading your `QApplication` but BEFORE creating any qt widget): `</translate>`

```
translator = QtCore.QTranslator()
translator.load("YourModule_"+languages[ln])
QtGui.QApplication.installTranslator(translator)
```

`</translate>` Optionally, you can also create the file XML Draft.qrc with this content: `</translate>`

```
<RCC>
<qresource prefix="/translations" >
<file>Draft_fr.qm</file>
</qresource>
</RCC>
```

`</translate>` and running `pyrcc4 Draft.qrc -o qrc_Draft.py` creates a big Python containing all resources. BTW this approach also works to put icon files in one resource file

Translating the wiki

This wiki is hosting a lot of contents, the majority of which build up the manual. You can browse the documentation starting from the Main Page (`/wiki/index.php?title=Main_Page`), or have a look at the User's manual Online Help Toc (`/wiki/index.php?title=Online_Help_Toc`).

Translation plugin

When the Wiki moved away from SourceForge, Yorik (`/wiki/index.php?title=User:Yorik`) installed a Translation plugin (`http://www.mediawiki.org/wiki/Help:Extension:Translate`) which allows to ease translations between pages. For example, the page title can now be translated. Other advantages of the Translation plugin are that it keeps track of translations, notifies if the original page has been updated, and maintains translations in sync with the original English page.

The tool is documented in Extension:Translate (`http://www.mediawiki.org/wiki/Help:Extension:Translate`), and is part of a Language Extension Bundle (`http://www.mediawiki.org/wiki/MediaWiki_Language_Extension_Bundle`).

To quickly get started on preparing a page for translation and activating the plugin, please read the Page translation example (http://www.mediawiki.org/wiki/Help:Extension:Translate/Page_translation_exampl

To see an example of how the Translation tool works once the translation plugin is activated on a page, you can visit the Main Page (/wiki/index.php?title=Main_Page). You will see a new language menu bar at the bottom. It is automatically generated. Click for instance on the German link, it will get you to Main Page/de (/wiki/index.php?title=Main_Page/de). Right under the title, you can read "This page is a **translated version** of a page Main Page and the translation is xx% complete." (xx being the actual percentage of translation). Click on the "translated version" link to start translation, or to update or correct the existing translation.

You will notice that you cannot directly edit a page anymore once it's been marked as a translation. You have to go through the translation utility.

When adding new content, the English page should be created first, then translated into another language. If someone wants to change/add content in a page, he should do the English one first.

It is recommended to have basic knowledge of wiki style formatting and general guidelines of the FreeCAD wiki, because you will have to deal with some tags while translating. You can find this information on WikiPages (</wiki/index.php?title=WikiPages>).

The sidebar (navigation menu on the left) is also translatable. Please follow dedicated instructions on Localisation Sidebar (/wiki/index.php?title=Localisation_Sidebar) page.

REMARK: The first time you switch a page to the new translation system, it loses all its old 'manual' translations. To recover the translation, you need to open an earlier version from the history, and copy/paste manually the paragraphs to the new translation system.

Remark: to be able to translate in the wiki, you must of course gain wiki edit permission (/wiki/index.php?title=FAQ#How_can_I_get_edit_permission_on_the_wiki.3F).

If you are unsure how to proceed, don't hesitate to ask for help in the forum (<http://forum.freecadweb.org>).

Old translation instructions

These instructions are for historical background only, while the pages are being passed to the new translation plugin.

So the first step is to **check if the manual translation has already been started for your language** (look in the left sidebar, under "manual").

If not, head to the forum (<http://forum.freecadweb.org/>) and say that you want to start a new translation, we'll create the basic setup for the language you want to work on.

You must then gain wiki edit permission (/wiki/index.php?title=FAQ#How_can_I_get_edit_permission_on_the_wiki.3F).

If your language is already listed, see what pages are still missing a translation (they will be listed in red). The technique is simple: **go into a red page, and copy/paste the contents of the corresponding English page, and start translating.**

Do not forget to include all the tags and templates from the original English page. Some of those templates will have an equivalent in your language (for example, there is a French Docnav template called Docnav/fr). You should use **a slash and your language code** in almost all the links. Look at other already translated pages to see how they did it.

Add a slash and your language code in the categories, like `[[Category:Developer Documentation/fr]]`

And if you are unsure, head to the forums and ask people to check what you did and tell you if it's right or not.

Four templates are commonly used in manual pages. These 4 templates have localized versions (Template:Docnav/fr, Template:fr, etc...)

- `Template:GuiCommand` (</wiki/index.php?title=Template:GuiCommand>): is the Gui Command information block in upper-right of command documentation.
- `Template:Docnav` (</wiki/index.php?title=Template:Docnav>): it is the navigation bar at the bottom of the pages, showing previous and next pages.

Page Naming Convention

Please take note that, due to limitations in the Sourceforge implementation of the MediaWiki engine, we require that your pages all keep their original English counterpart's name, appending a slash and your language code. For example, the translated page for About FreeCAD should be About FreeCAD/es for Spanish, About FreeCAD/pl for polish, etc. The reason is simple: so that if translators go away, the wiki's administrators, who do not speak all languages, will know what these pages are for. This will facilitate maintenance and avoid lost pages.

If you want the Docnav template to show linked pages in your language, you can use **redirect pages**. They are basically shortcut links to the actual page. Here is an example with the French page for About FreeCAD.

- The page About FreeCAD/fr is the page with content
- The page À propos de FreeCAD contains this code:

```
#REDIRECT [[About FreeCAD/fr]]
```

- In the About FreeCAD/fr page, the Docnav code will look like this:

```
{{docnav/fr|Bienvenue sur l'aide en ligne|Fonctionnalités}}
```

The page "Bienvenue sur l'aide en ligne" redirects to Online Help Startpage/fr, and the page "Fonctionnalités" redirects to Feature list/fr.

< previous: Branding (</wiki/index.php?title=Branding>) Index
 next: Extra python modules > (/wiki/index.php?title=Extra_python_modules)
 (/wiki/index.php?title=Online_Help_Toc)

</translate>

< translate> This page lists several additional python modules or other pieces of software that can be downloaded freely from the internet, and add functionality to your FreeCAD installation.

PySide (previously PyQt4)

- homepage (PySide): <http://qt-project.org/wiki/PySide> (<http://qt-project.org/wiki/PySide>)
- license: LGPL
- optional, but needed by several modules: Draft, Arch, Ship, Plot, OpenSCAD, Spreadsheet

PySide (previously PyQt) is required by several modules of FreeCAD to access FreeCAD's Qt interface. It is already bundled in the windows version of FreeCAD, and is usually installed automatically by FreeCAD on Linux, when installing from official repositories. If those modules (Draft, Arch, etc) are enabled after FreeCAD is installed, it means PySide (previously PyQt) is already there, and you don't need to do anything more.

Note: FreeCAD progressively moved away from PyQt after version 0.13, in favour of PySide (<http://qt-project.org/wiki/PySide>), which does exactly the same job but has a license (LGPL) more compatible with FreeCAD.

Installation

Linux

The simplest way to install PySide is through your distribution's package manager. On Debian/Ubuntu systems, the package name is generally *python-PySide*, while on RPM-based systems it is named *pyside*. The necessary dependencies (Qt and SIP) will be taken care of automatically.

Windows

The program can be downloaded from <http://qt-project.org/wiki/Category:LanguageBindings::PySide::Downloads> (<http://qt-project.org/wiki/Category:LanguageBindings::PySide::Downloads>) . You'll need to install the Qt and SIP libraries before installing PySide (to be documented).

MacOSX

PyQt on Mac can be installed via homebrew or port. See [CompileOnMac#Install_Dependencies](#) ([/wiki/index.php?title=CompileOnMac#Install_Dependencies](#)) for more information.

Usage

Once it is installed, you can check that everything is working by typing in FreeCAD python console:< /translate>

```
import PySide
```

<translate> To access the FreeCAD interface, type :< /translate>

```
from PySide import QtCore,QtGui
FreeCADWindow = FreeCADGui.getMainWindow()
```

<translate> Now you can start to explore the interface with the `dir()` command. You can add new elements, like a custom widget, with commands like :< /translate>

```
FreeCADWindow.addDockWidget(QtCore.Qt.RightDockWidgetArea,my_custom_widget)
```

<translate> Working with Unicode :< /translate>

```
text = text.encode('utf-8')
```

<translate> Working with QFileDialog and OpenFileName :< /translate>

```

path = FreeCAD.ConfigGet("AppHomePath")
#path = FreeCAD.ConfigGet("UserAppData")
OpenName, Filter = PySide.QtGui.QFileDialog.getOpenFileName(None, "Read a txt file", path, "*.txt")

```

<translate> Working with QFileDialog and SaveFileName :< /translate>

```

path = FreeCAD.ConfigGet("AppHomePath")
#path = FreeCAD.ConfigGet("UserAppData")
SaveName, Filter = PySide.QtGui.QFileDialog.getSaveFileName(None, "Save a file txt", path, "*.txt")

```

<translate>

Example of transition from PyQt4 and PySide

PS: these examples of errors were found in the transition PyQt4 to PySide and these corrections were made, other solutions are certainly available with the examples above < /translate>

```

try:
    import PyQt4                                # PyQt4
    from PyQt4 import QtGui ,QtCore            # PyQt4
    from PyQt4.QtGui import QComboBox          # PyQt4
    from PyQt4.QtGui import QMessageBox        # PyQt4
    from PyQt4.QtGui import QTableWidgetItem, QApplication # PyQt4
    from PyQt4.QtGui import *                  # PyQt4
    from PyQt4.QtCore import *                 # PyQt4
except Exception:
    import PySide                                # PySide
    from PySide import QtGui ,QtCore            # PySide
    from PySide.QtGui import QComboBox          # PySide
    from PySide.QtGui import QMessageBox        # PySide
    from PySide.QtGui import QTableWidgetItem, QApplication # PySide
    from PySide.QtGui import *                  # PySide
    from PySide.QtCore import *                 # PySide

```

<translate> To access the FreeCAD interface, type: You can add new elements, like a custom widget, with commands like :< /translate>

```

myNewFreeCADWidget = QtGui.QDockWidget()      # create a new dockwidget
myNewFreeCADWidget.ui = Ui_MainWindow()      # myWidget_Ui()          # load the Ui script
myNewFreeCADWidget.ui.setupUi(myNewFreeCADWidget) # setup the ui
try:
    app = QtGui.QApp                            # PyQt4 # the active qt window, = the freecad window since we are inside it
    FCmw = app.activeWindow()                    # PyQt4 # the active qt window, = the freecad window since we are inside it
    FCmw.addDockWidget(QtGui.Qt.RightDockWidgetArea,myNewFreeCADWidget) # add the widget to the main window
except Exception:
    FCmw = FreeCADGui.getMainWindow()            # PySide # the active qt window, = the freecad window since we are inside it
    FCmw.addDockWidget(QtGui.Qt.RightDockWidgetArea,myNewFreeCADWidget) # add the widget to the main window

```

<translate> Working with Unicode : < /translate>

```

try:
    text = unicode(text, 'ISO-8859-1').encode('UTF-8') # PyQt4
except Exception:
    text = text.encode('utf-8')                       # PySide

```

<translate> Working with QFileDialog and OpenFileName :< /translate>

```

OpenName = ""
try:
    OpenName = QFileDialog.getOpenFileName(None,QString.fromLocal8Bit("Lire un fichier FCInfo ou txt"),path,"*.FCInfo *.txt") # PyQt4
except Exception:
    OpenName, Filter = PySide.QtGui.QFileDialog.getOpenFileName(None, "Lire un fichier FCInfo ou txt", path, "*.FCInfo *.txt")#PySide

```

<translate> Working with QFileDialog and SaveFileName :< /translate>

```

SaveName = ""
try:
    SaveName = QFileDialog.getSaveFileName(None,QString.fromLocal8Bit("Sauver un fichier FCInf
o"),path,"*.FCInfo") # PyQt4
except Exception:
    SaveName, Filter = PySide.QtGui.QFileDialog.getSaveFileName(None, "Sauver un fichier FCInf
o", path, "*.FCInfo")# PySide

```

<translate> The MessageBox:< /translate>

```

def errorDialog(msg):
    diag = QtGui.QMessageBox(QtGui.QMessageBox.Critical,u"Error Message",msg )
    try:
        diag.setWindowFlags(PyQt4.QtCore.Qt.WindowStaysOnTopHint) # PyQt4 # this function sets
the window before
    except Exception:
        diag.setWindowFlags(PySide.QtCore.Qt.WindowStaysOnTopHint)# PySide # this function set
s the window before
#    diag.setWindowModality(QtCore.Qt.ApplicationModal)      # function has been disabled to
promote "WindowStaysOnTopHint"
    diag.exec_()

```

<translate> Working with setProperty (PyQt4) and setValue (PySide)
< /translate>

```
self.doubleSpinBox.setProperty("value", 10.0) # PyQt4
```

<translate> replace to :< /translate>

```
self.doubleSpinBox.setValue(10.0) # PySide
```

<translate> Working with setToolTip< /translate>

```
self.doubleSpinBox.setToolTip(_translate("MainWindow", "Coordinate placement Axis Y", None))
# PyQt4
```

<translate> replace to :< /translate>

```
self.doubleSpinBox.setToolTip(_fromUtf8("Coordinate placement Axis Y")) # PySide
```

<translate> or :< /translate>

```
self.doubleSpinBox.setToolTip(u"Coordinate placement Axis Y.")# PySide
```

<translate>

Additional documentation

Some pyQt4 tutorials (including how to build interfaces with Qt Designer to use with python):

- <http://pyqt.sourceforge.net/Docs/PyQt4/classes.html>
(<http://pyqt.sourceforge.net/Docs/PyQt4/classes.html>) - the PyQt4 API Reference on sourceforge
- <http://www.rkblog.rk.edu.pl/w/p/introduction-pyqt4/>
(<http://www.rkblog.rk.edu.pl/w/p/introduction-pyqt4/>) - a simple introduction
- <http://www.zetcode.com/tutorials/pyqt4/>
(<http://www.zetcode.com/tutorials/pyqt4/>) - very complete in-depth tutorial

Pivy

- homepage: <https://bitbucket.org/Coin3D/coin/wiki/Home>
(<https://bitbucket.org/Coin3D/coin/wiki/Home>)
- license: BSD
- optional, but needed by several modules of FreeCAD: Draft, Arch

Pivy is needed by several modules to access the 3D view of FreeCAD. On windows, Pivy is already bundled inside the FreeCAD installer, and on Linux it is usually automatically installed when you install FreeCAD from an official repository. On MacOSX, unfortunately, you will need to compile pivy yourself.

Installation

Prerequisites

I believe before compiling Pivy you will want to have Coin and SoQt installed.

I found for building on Mac it was sufficient to install the Coin3 binary package (http://www.coin3d.org/lib/plonesoftwarecenter_view). Attempting to install coin from MacPorts was problematic: tried to add a lot of X Windows packages and ultimately crashed with a script error.

For Fedora I found an RPM with Coin3.

SoQt compiled from source (<http://www.coin3d.org/lib/soqt/releases/1.5.0>) fine on Mac and Linux.

Debian & Ubuntu

Starting with Debian Squeeze and Ubuntu Lucid, pivy will be available directly from the official repositories, saving us a lot of hassle. In the meantime, you can either download one of the packages we made (for debian and ubuntu karmic) availables on the Download (</wiki/index.php?title=Download>) pages, or compile it yourself.

The best way to compile pivy easily is to grab the debian source package for pivy and make a package with debuild. It is the same source code from the official pivy site, but the debian people made several bug-fixing additions. It also compiles fine on ubuntu karmic: <http://packages.debian.org/squeeze/python-pivy> (<http://packages.debian.org/squeeze/python-pivy>) download the .orig.gz and the .diff.gz file, then unzip both, then apply the .diff to the source: go to the unzipped pivy source folder, and apply the .diff patch: < /translate>

```
patch -p1 < ../pivy_0.5.0~svn765-2.diff
```

<translate> then< /translate>

```
debuild
```

<translate> to have pivy properly built into an official installable package. Then, just install the package with gdebi.

Other linux distributions

First get the latest sources from the project's repository (<http://pivy.coin3d.org/mercurial/>):< /translate>

```
hg clone http://hg.sim.no/Pivy/default Pivy
```

<translate> As of March 2012, the latest version is Pivy-0.5.

Then you need a tool called SWIG to generate the C++ code for the Python bindings. Pivy-0.5 reports that it has only been tested with SWIG 1.3.31, 1.3.33, 1.3.35, and 1.3.40. So you can download a source tarball for one of these old versions from <http://www.swig.org> (<http://www.swig.org>). Then unpack it and from a command line do (as root):< /translate>

```
./configure
make
make install (or checkinstall if you use it)
```

<translate> It takes just a few seconds to build.

Alternatively, you can try building with a more recent SWIG. As of March 2012, a typical repository version is 2.0.4. Pivy has a minor compile problem with SWIG 2.0.4 on Mac OS (see below) but seems to build fine on Fedora Core 15.

After that go to the pivy sources and call < /translate>

```
python setup.py build
```

<translate> which creates the source files. Note that build can produce thousands of warnings, but hopefully there will be no errors.

This is probably obsolete, but you may run into a compiler error where a 'const char*' cannot be converted in a 'char*'. To fix that you just need to write a 'const' before in the appropriate lines. There are six lines to fix.

After that, install by issuing (as root):< /translate>

```
python setup.py install (or checkinstall python setup.py install)
```

<translate> That's it, pivy is installed.

Mac OS

These instructions may not be complete. Something close to this worked for OS 10.7 as of March 2012. I use MacPorts for repositories, but other options should also work.

As for linux, get the latest source: < /translate>

```
hg clone http://hg.sim.no/Pivy/default Pivy
```

<translate> If you don't have hg, you can get it from MacPorts:< /translate>

```
port install mercurial
```

<translate> Then, as above you need SWIG. It should be a matter of:< /translate>

```
port install swig
```

<translate> I found I needed also:< /translate>

```
port install swig-python
```

<translate> As of March 2012, MacPorts SWIG is version 2.0.4. As noted above for linux, you might be better off downloading an older version. SWIG 2.0.4 seems to have a bug that stops Pivy building. See first message in this digest: https://sourceforge.net/mailarchive/message.php?msg_id=28114815 (https://sourceforge.net/mailarchive/message.php?msg_id=28114815)

This can be corrected by editing the 2 source locations to add dereferences: *arg4, *arg5 in place of arg4, arg5. Now Pivy should build: < /translate>

```
python setup.py build
sudo python setup.py install
```

<translate>

Windows

Assuming you are using Visual Studio 2005 or later you should open a command prompt with 'Visual Studio 2005 Command prompt' from the Tools menu. If the Python interpreter is not yet in the system path do< /translate>

```
set PATH=path_to_python_2.5;%PATH%
```

<translate> To get pivy working you should get the latest sources from the project's repository:< /translate>

```
svn co https://svn.coin3d.org/repos/Pivy/trunk Pivy
```

<translate> Then you need a tool called SWIG to generate the C++ code for the Python bindings. It is recommended to use version 1.3.25 of SWIG, not the latest version, because at the moment pivy will only function correctly with 1.3.25. Download the binaries for 1.3.25 from <http://www.swig.org> (<http://www.swig.org>). Then unpack it and from the command line add it to the system path< /translate>

```
set PATH=path_to_swig_1.3.25;%PATH%
```

<translate> and set COINDIR to the appropriate path< /translate>

```
set COINDIR=path_to_coin
```

<translate> On Windows the pivy config file expects SoWin instead of SoQt as default. I didn't find an obvious way to build with SoQt, so I modified the file setup.py directly. In line 200 just remove the part 'sowin': ('gui._sowin', 'sowin-config', 'pivy.gui.')

After that go to the pivy sources and call < /translate>

```
python setup.py build
```

<translate> which creates the source files. You may run into a compiler error several header files couldn't be found. In this case adjust the INCLUDE variable< /translate>

```
set INCLUDE=%INCLUDE%;path_to_coin_include_dir
```

<translate> and if the SoQt headers are not in the same place as the Coin headers also < /translate>

```
set INCLUDE=%INCLUDE%;path_to_soqt_include_dir
```

<translate> and finally the Qt headers< /translate>

```
set INCLUDE=%INCLUDE%;path_to_qt4\include\Qt
```

<translate> If you are using the Express Edition of Visual Studio you may get a python keyerror exception. In this case you have to modify a few things in msvccompiler.py located in your python installation.

Go to line 122 and replace the line< /translate>

```
vsbase = r"Software\Microsoft\VisualStudio\%0.1f" % version
```

<translate> with< /translate>

```
vsbase = r"Software\Microsoft\VCExpress\%0.1f" % version
```

<translate> Then retry again. If you get a second error like < /translate>

```
error: Python was built with Visual Studio 2003;...
```

<translate> you must also replace line 128< /translate>

```
self.set_macro("FrameworkSDKDir", net, "sdkinstallrootv1.1")
```

<translate> with< /translate>

```
self.set_macro("FrameworkSDKDir", net, "sdkinstallrootv2.0")
```

<translate> Retry once again. If you get again an error like< /translate>

```
error: Python was built with Visual Studio version 8.0, and extensions need to be built with the same version of the compiler, but it isn't installed.
```

<translate> then you should check the environment variables DISTUTILS_USE_SDK and MSSDK with< /translate>

```
echo %DISTUTILS_USE_SDK%
echo %MSSDK%
```

<translate> If not yet set then just set it e.g. to 1< /translate>

```
set DISTUTILS_USE_SDK=1
set MSSDK=1
```

<translate> Now, you may run into a compiler error where a 'const char*' cannot be converted in a 'char*'. To fix that you just need to write a 'const' before in the appropriate lines. There are six lines to fix. After that copy the generated pivy directory to a place where the python interpreter in FreeCAD can find it.

Usage

To check if Pivy is correctly installed:< /translate>

```
import pivy
```

<translate> To have Pivy access the FreeCAD scenegraph do the following:< /translate>

```
from pivy import coin
App.newDocument() # Open a document and a view
view = Gui.ActiveDocument.ActiveView
FCSceneGraph = view.getSceneGraph() # returns a pivy Python object that holds a SoSeparator, the main "container" of the Coin scenegraph
FCSceneGraph.addChild(coin.SoCube()) # add a box to scene
```

<translate> You can now explore the FCSceneGraph with the dir() command.

Additional Documentation

Unfortunately documentation about pivy is still almost inexistant on the net. But you might find Coin documentation useful, since pivy simply translate Coin functions, nodes and methods in python, everything keeps the same name and properties, keeping in mind the difference of syntax between C and python:

- <https://bitbucket.org/Coin3D/coin/wiki/Documentation>
(<https://bitbucket.org/Coin3D/coin/wiki/Documentation>) - Coin3D API Reference
- http://www-evasion.imag.fr/~Francois.Faure/doc/inventorMentor/sgi_html/index.html
(http://www-evasion.imag.fr/~Francois.Faure/doc/inventorMentor/sgi_html/index.html)
- The Inventor Mentor - The "bible" of Inventor scene description language.

You can also look at the Draft.py file in the FreeCAD Mod/Draft folder, since it makes big use of pivy.

pyCollada

- homepage: <http://pycollada.github.com> (<http://pycollada.github.com>)
- license: BSD
- optional, needed to enable import and export of Collada (.DAE) files

pyCollada is a python library that allow programs to read and write Collada (*.DAE) (<http://en.wikipedia.org/wiki/COLLADA>) files. When pyCollada is installed on your system, FreeCAD will be able to handle importing and exporting in the Collada file format.

Installation

Pycollada is usually not yet available in linux distributions repositories, but since it is made only of python files, it doesn't require compilation, and is easy to install. You have 2 ways, or directly from the official pycollada git repository, or with the easy_install tool.

Linux

In either case, you'll need the following packages already installed on your system:< /translate>

```
python-lxml
python-numpy
python-dateutil
```

<translate>

From the git repository

</translate>

```
git clone git://github.com/pycollada/pycollada.git pycollada
cd pycollada
sudo python setup.py install
```

<translate>

With easy_install

Assuming you have a complete python installation already, the easy_install utility should be present already:< /translate>

```
easy_install pycollada
```

<translate> You can check if pycollada was correctly installed by issuing in a python console: < /translate>

```
import collada
```

<translate> If it returns nothing (no error message), then all is OK

Windows

1. Install Python. While FreeCAD and some other programs come with a bundled version of Python, having a fixed install will help with the next steps. You can get Python here: <https://www.python.org/downloads/> (<https://www.python.org/downloads/>) . Of course you should pick the right version, in this case that would be 2.6.X, as FreeCAD currently uses 2.6.2 (Personally I went with 2.6.2, and by the way, you can check the version yourself by starting the Python.exe in the bin folder of FreeCAD). You'll also have to add the path of the installation directory into the path variable so you can access Python from the cmd. Now we can install the missing things, in total there are 3 things we need to install: numpy, setuptools, pycollada
2. Fetch [numpy](http://sourceforge.net/projects/numpy/files/NumPy/) here: <http://sourceforge.net/projects/numpy/files/NumPy/> (<http://sourceforge.net/projects/numpy/files/NumPy/>) . Pick a version which fits to the version used by FreeCAD, there are multiple installers for different Python versions in every numpy version folder, the installer will put numpy into the folder of your Python installation, where FreeCAD can access it as well
3. Fetch [setuptools](https://pypi.python.org/pypi/setuptools) here: <https://pypi.python.org/pypi/setuptools> (<https://pypi.python.org/pypi/setuptools>) (We need to install the setuptools in order to install pycollada in the next step)
4. Unzip the downloaded setuptools file somewhere
5. Start a cmd with admin permission
6. Navigate to the unpacked setuptools folder

7. Install the setuptools by tipping "Python setup.py install" into the cmd, this will not work when Python is not installed or when the path variable hasn't been configured
8. Fetch `pycollada` here:
<https://pypi.python.org/pypi/pycollada/>
 (<https://pypi.python.org/pypi/pycollada/>) (has already been posted above) and once again:
9. Unzip the downloaded pycollada file somewhere
10. Start a cmd with admin permission, or use the one you opened not long ago
11. Navigate to the unpacked pycollada folder
12. Install the setuptools by tipping "Python setup.py install" into the cmd
 - Another reference on how to use easy_install: <http://jishus.org/?p=452> (<http://jishus.org/?p=452>)

Mac OS

If you are using the Homebrew build of FreeCAD you can install pycollada into your system Python using pip.

If you need to install pip:< /translate>

```
$ sudo easy_install pip
```

<translate> Install pycollada:< /translate>

```
$ sudo pip install pycollada
```

<translate> If you are using a binary version of FreeCAD, you can tell pip to install pycollada into the site-packages inside FreeCAD.app:< /translate>

```
$ pip install --target="/Applications/FreeCAD.app/Contents/lib/python2.7/site-packages" pycollada
```

<translate>

IfcOpenShell

- homepage: <http://www.ifcopenshell.org> (<http://www.ifcopenshell.org>)
- license: LGPL
- optional, needed to extend import abilities of IFC files

IfcOpenShell is a library currently in development, that allows to import (and soon export) Industry foundation Classes (*.IFC) (http://en.wikipedia.org/wiki/Industry_Foundation_Classes) files. IFC is an extension to the STEP format, and is becoming the standard in BIM (http://en.wikipedia.org/wiki/Building_information_modeling) workflows. When ifcopenshell is correctly installed on your system, the FreeCAD Arch Module (/wiki/index.php?title=Arch_Module) will detect it and use it to import IFC files, instead of its built-in rudimentary importer. Since ifcopenshell is based on OpenCasCade, like FreeCAD, the quality of the import is very high, producing high-quality solid geometry.

Installation

Since ifcopenshell is pretty new, you'll likely need to compile it yourself.

Linux

You will need a couple of development packages installed on your system in order to compile ifcopenshell:< /translate>

```
liboce-*-dev
python-dev
swig
```

<translate> but since FreeCAD requires all of them too, if you can compile FreeCAD, you won't need any extra dependency to compile IfcOpenShell.

Grab the latest source code from here:< /translate>

```
svn co https://svn.code.sf.net/p/ifcopenshell/svn/trunk ifcopenshell ifcopenshell
```

<translate> or< /translate>

```
svn co https://ifcopenshell.svn.sourceforge.net/svnroot/ifcopenshell ifcopenshell
```

<translate> The build process is very easy:< /translate>

```
mkdir ifcopenshell-build
cd ifcopenshell-build
cmake ../ifcopenshell/cmake
```

<translate> or, if you are using oce instead of opencascade:< /translate>

```
cmake -DOCC_INCLUDE_DIR=/usr/include/oce ../ifcopenshell/cmake
```

<translate> Since ifcopenshell is made primarily for Blender, it uses python3 by default. To use it inside FreeCAD, you need to compile it against the same version of python that is used by FreeCAD. So you might need to force the python version with additional cmake parameters (adjust the python version to yours): < /translate>

```
cmake -DOCC_INCLUDE_DIR=/usr/include/oce -DPYTHON_INCLUDE_DIR=/usr/include/python2.7 -DPYTHON_LIBRARY=/usr/lib/python2.7.so ../ifcopenshell/cmake
```

<translate> Then:< /translate>

```
make
sudo make install
```

<translate> You can check that ifcopenshell was correctly installed by issuing in a python console: < /translate>

```
import IfcImport
```

<translate> If it returns nothing (no error message), then all is OK

Windows

Copied from the IfcOpenShell README file

Users are advised to use the Visual Studio .sln file in the win/ folder. For Windows users a prebuilt Open CASCADE version is available from the <http://opencascade.org> (<http://opencascade.org>) website. Download and install this version and provide the paths to the Open CASCADE header and library files to MS Visual Studio C++.

For building the IfcPython wrapper, SWIG needs to be installed. Please download the latest swigwin version from <http://www.swig.org/download.html> (<http://www.swig.org/download.html>) . After extracting the .zip file, please add the extracted folder to the PATH environment variable. Python needs to be installed, please provide the include and library paths to Visual Studio.

Teigha Converter

- homepage: <http://www.opendesign.com/guestfiles/TeighaFileConverter> (<http://www.opendesign.com/guestfiles/TeighaFileConverter>)
- license: freeware
- optional, used to enable import and export of DWG files

The Teigha Converter is a small freely available utility that allows to convert between several versions of DWG and DXF files. FreeCAD can use it to offer DWG import and export, by converting DWG files to the DXF format under the hood, then using its standard DXF importer to import the file contents. The restrictions of the DXF importer (/wiki/index.php?title=Draft_DXF) apply.

Installation

On all platforms, only by installing the appropriate package from <http://www.opendesign.com/guestfiles/TeighaFileConverter> (<http://www.opendesign.com/guestfiles/TeighaFileConverter>) . After installation, if the utility is not found automatically by FreeCAD, you might need to set the path to the converter executable manually, in the menu Edit -> Preferences -> Draft -> Import/Export options.

< previous: Localisation (</wiki/index.php?title=Localisation>) Index
next: Source documentation > (/wiki/index.php?title=Source_documentation)

(/wiki/index.php?title=Online_Help_Toc)

</translate>

Credits

FreeCAD would not be what it is without the generous contributions of many people. Here's an overview of the people and companies who contributed to FreeCAD over time. For credits for the third party libraries see the Third Party Libraries (/wiki/index.php?title=Third_Party_Libraries) page.

Development

Project managers

Lead developers of the FreeCAD project:

- Jürgen Riegel (</wiki/index.php?title=User:Jriegel>)
- Werner Mayer (</wiki/index.php?title=User:Wmayer>)
- Yorik van Havre (</wiki/index.php?title=User:Yorikvanhavre>)

Main developers

People who work regularly on the FreeCAD code (retrieved from <https://github.com/FreeCAD/FreeCAD/graphs/contributors> (<https://github.com/FreeCAD/FreeCAD/graphs/contributors>)):

- Abdullah Tahiriyo (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=3232>)
- Alexander Golubev (Fat-Zer) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=4325>)
- Bernd Hahnbach (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=2069>)
- Brad Collette (sliptonic) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=708>)
- Daniel Falck (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=689>)
- Eivind Kvedalen (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1546>)
- f3nix (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=6125>)
- Ian Rees (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=3449>)

- Jan Rheinlaender (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=997>)
- Jonathan Wiedemann (rockn) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=681>)
- Jose Luis Cercos Pita (sanguinariojoe) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=574>)
- Logari81 (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=270>)
- Luke A. Parry (<http://freecadamusements.blogspot.co.uk/>)
- mdinger (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=2928>)
- mghansen
- Przemo Firszt(PrzemoF) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=3666>)
- sgrogan (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=4252>)
- shoogen (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=765>)
- Stefan Tröger (ickby) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=686>)
- tanderson69 (blobfish) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=208>)
- vejmarie (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=7506>)
- Victor Titov (DeepSOIC) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=3888>)
- wandererfan (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1375>)

Other coders

Other people who contributed code to the FreeCAD project:

- jmaustpc (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=611>)
- j-dowsett (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=652>)
- keithsloan52 (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=930>)
- Joachim Zettler
- Graeme van der Vlugt
- Berthold Grupp
- Georg Wiora (</wiki/index.php?title=User:Xorx>)
- Martin Burbaum
- Jacques-Antoine Gaudin
- Ken Cline
- Dmitry Chigrin
- Remigiusz Fiedler (DXF-parser)
- peterl94

- jobermayr
- ovginkel
- triplus
- tomate44
- maurerpe
- Johan3DV
- Mandeep Singh
- fandaL
- jonnor
- usakhelo
- plaes
- SebKuzminsky
- jcc242
- ezziygyuwuf
- marktaff
- poutine70
- qingfengxia
- dbtayl
- itain
- Barleyman

Companies

Companies which donated code or developer time:

- Imetric 3D

Forum moderators

People in charge of the FreeCAD forum (<http://forum.freecadweb.org>) (retrieved from <http://forum.freecadweb.org/memberlist.php?mode=team> (<http://forum.freecadweb.org/memberlist.php?mode=team>)):

- Daniel Falck (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=689>)
- DeepSOIC (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=3888>)
- ediloren (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1783>)
- jmaustpc (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=611>)
- jriegel (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=67>)
- Logari81 (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=270>)
- mrlukeparry (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=607>)
- onesz (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=729>)
- Przemof (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=3666>)

- r-frank (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1529>)
- Renato Rebelo (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=3315>)
- rockn (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=681>)
- shoogen (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=765>)
- wmayer (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=69>)
- yorik (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=68>)

Community

People from the community who put a lot of efforts in helping the FreeCAD project either by being active on the forum, keeping a blog about FreeCAD, making video tutorials, packaging FreeCAD for Windows/Linux/MacOS X, writing a FreeCAD book... (listed by alphabetical order) (retrieved from <http://forum.freecadweb.org/memberlist.php?mode=&sk=d&sd=d#memberlist> (<http://forum.freecadweb.org/memberlist.php?mode=&sk=d&sd=d#memberlist>))

- bejant (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1940>)
- Brad Collette (<http://www.packtpub.com/freecad-solid-modeling-with-python/book>)
- cblt2l (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=251>)
- cox (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=4523>)
- Daniel Falck (<http://opensource.designtools.blogspot.com/>)
- Eduardo Magdalena (</wiki/index.php?title=User:Emagdalena>)
- hobbes1069 (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=725>)
- jdurston (5needinginput) (<http://www.youtube.com/user/5needinginput>)
- John Morris (butchwax) (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=861>)
- Kwahooo (<http://freecad-tutorial.blogspot.com/>)
- lhagan (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=108>)
- marcxS (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1047>)
- Mario52 (</wiki/index.php?title=User:Mario52>)
- Normandc (</wiki/index.php?title=User:Normandc>)
- peterl94 (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1819>)
- pperisin (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=356>)
- Quick61 (</wiki/index.php?title=User:Quick61>)
- Renatorivo (</wiki/index.php?title=User:Renatorivo>)
- Rockn (</wiki/index.php?title=User:Rockn>)

- triplus (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=782>)
- ulrich1a (<http://forum.freecadweb.org/memberlist.php?mode=viewprofile&u=1928>)

Documentation writers

People who wrote the documentation on this wiki (/wiki/index.php?title=Main_Page):

- Renato Rivoira (renatorivo)
- Honza32
- Hervé Blorec
- Eduardo Magdalena
- piffpoof
- Wurstwasser
- Roland Frank (r-frank)
- bejant
- Ediloren
- Isaac Ayala

Translators

People who helped to translate the FreeCAD application (retrieved from <https://crowdin.com/project/freecad> (<https://crowdin.com/project/freecad>)):

- Gerhard Scheepers
- wbrwbr2011
- hanhsuan
- hicarl
- fandaL
- Peta T
- Zdeněk Havlík
- Jodbe
- Peter Hageman
- Vilfredo
- Bruno Gonçalves Pirajá
- Timo Seppola
- rako
- Pasi Kukkola
- Ettore Atalan
- nikoss
- yang12
- totyg
- htsubota
- asakura
- Masaya Ootsuki
- Jiyong Choi
- Bartłomiej Niemiec
- trzyha

- bluecd
- Miguel Morais
- Nicu Tofan
- Victor Radulescu
- Angelescu Constantin
- sema
- Николай Матвеев
- pinkpony
- Alexandre Prokoudine
- Марко Пејовић
- Marosh
- Peter Klofutar
- Raulshc
- javierMG
- Lars
- kunguz
- Igor
- Федір

Addons developers

Developers of FreeCAD addons (retrieved from <https://github.com/FreeCAD/FreeCAD-addons> (<https://github.com/FreeCAD/FreeCAD-addons>)):

- microelly2
- hamish2014
- jreinhardt
- jmwright
- cbt2l
- javierMG
- looooo
- shaise
- marmni
- Maaphoo
- Rentlau

Kategorien (/wiki/index.php?title=Special:Categories):

Pages with syntax highlighting errors (/wiki/index.php?title=Category:Pages_with_syntax_highlighting_errors&action=edit&redlink=1)

User Documentation/en (/wiki/index.php?title=Category:User_Documentation/en)

User Documentation (/wiki/index.php?title=Category:User_Documentation)

Pages with broken file links (/wiki/index.php?title=Category:Pages_with_broken_file_links)

Poweruser Documentation/en (/wiki/index.php?title=Category:Poweruser_Documentation/en)

Poweruser Documentation (/wiki/index.php?title=Category:Poweruser_Documentation)

[Python Code \(/wiki/index.php?title=Category:Python_Code\)](/wiki/index.php?title=Category:Python_Code)
[Tutorials \(/wiki/index.php?title=Category:Tutorials\)](/wiki/index.php?title=Category:Tutorials)
[Tutorials/en \(/wiki/index.php?title=Category:Tutorials/en\)](/wiki/index.php?title=Category:Tutorials/en)
[Python Code/en \(/wiki/index.php?title=Category:Python_Code/en\)](/wiki/index.php?title=Category:Python_Code/en)
[Developer Documentation \(/wiki/index.php?title=Category:Developer_Documentation\)](/wiki/index.php?title=Category:Developer_Documentation)
[Developer Documentation/en \(/wiki/index.php?title=Category:Developer_Documentation/en\)](/wiki/index.php?title=Category:Developer_Documentation/en)
[Developer \(/wiki/index.php?title=Category:Developer\)](/wiki/index.php?title=Category:Developer)

Community

Learn

Help the project

Code

<p>Github https://github.com/FreeCAD/FreeCAD</p> <p>Facebook https://www.facebook.com/FreeCAD/</p> <p>Google+ https://plus.google.com/+StackExchange</p>	<p>Tutorials (/wiki/?title=FreeCAD)</p> <p>Youtube videos https://www.youtube.com/watch?v=31474646</p> <p>Stack Exchange http://area51.stackexchange.com/proposals/88424/freecad</p>	<p>How can I help? (/wiki/?title=Help_FreeCAD)</p> <p>Donate! (/wiki/?title=Donate)</p> <p>Provide a license! (/wiki/?title=Licence)</p>	<p>Building from source (/wiki/?title=Compiling)</p> <p>C++ & Python API (/api/)</p> <p>License information (/wiki/?title=Licence)</p>
---	---	--	---