

Online Help Startpage



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://apps.sourceforge.net/mediawiki/free-cad/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](#), 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://apps.sourceforge.net/mediawiki/free-cad/index.php> in the User section. You will need a [sourceforge account](#) to log in, and then ask on the forum or on the irc channel for write permission (the wiki is write-protected to avoid spamming). Then you can start editing! Also, check out the page at https://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Help_FreeCAD for more ways you can help FreeCAD.

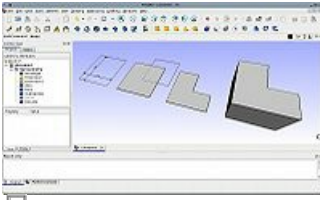
Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Online_Help_Startpage"

Online Help Toc

Here is the table of Content for the On-line Help system in FreeCAD. The articles referenced here are automatically included in the FreeCAD.chm file by the wiki2chm.py tool. You find that tool under src/Tools/wiki2chm.py. A [printable version](#) of this manual is also available.

- [Welcome](#)
- Introduction
 - [About FreeCAD](#)
 - [Features](#)
 - [Installing on Windows](#)
 - [Installing on Linux/Unix](#)
 - [Installing on Mac](#)
- Working with FreeCAD
 - [Getting started](#)
 - [Navigating in the 3D space](#)
 - [The FreeCAD Document](#)
 - [Setting user preferences](#)
 - [Customizing the interface](#)
 - [Object properties](#)
 - [Working with workbenches](#)
 - [The PartDesign workbench](#)
 - [The Mesh workbench](#)
 - [The Part workbench](#)
 - [The Drawing workbench](#)
 - [The Raytracing workbench](#)
 - [The Image workbench](#)
 - [The 2D drafting workbench](#)
 - [The Robot workbench](#)
 - [List of all FreeCAD commands](#)
 - [Tutorials](#)
- Scripting and Macros
 - [Working with macros](#)
 - [Introduction to python](#)
 - [FreeCAD Scripting Basics](#)
 - [Mesh Scripting](#)
 - [Part Scripting](#)
 - [Converting between Meshes and Parts](#)
 - [The Coin Scenagraph](#)
 - [Working with Pivy](#)
 - [Working with PyQt](#)
 - [Creating parametric objects](#)
 - [Embedding FreeCAD](#)
 - [API documentation](#)
 - Scripting Examples
 - [Code snippets](#)
 - [Line drawing function](#)
 - [Dialog creation](#)
- Developing applications for FreeCAD
 - [Licence](#)
 - Compiling FreeCAD
 - [Finding assistance](#)
 - [Compiling on Windows](#)
 - [Compiling on Unix](#)
 - [Compiling on Mac](#)
 - [Third Party Libraries](#)
 - [Third Party Tools](#)
 - [Start up and Configuration](#)
 - Build Support Tools
 - [The FreeCAD build tool](#)
 - [Adding an application module](#)
 - [Debugging FreeCAD](#)
 - [Testing FreeCAD](#)
 - Modifying FreeCAD
 - [Branding](#)
 - [Translating FreeCAD](#)
 - [Installing extra python modules](#)
 - [Source documentation](#)
- Credits
 - [Contributors](#)

About FreeCAD



the FreeCAD interface

FreeCAD is a general purpose 3D **CAD** modeler. The development is completely **Open Source** (GPL & LGPL License). FreeCAD is aimed directly at **mechanical engineering** and **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**, **SolidWorks** or **Solid Edge**, and therefore also falls into the category of **MCAD**, **PLM**, **CAX** and **CAE**. It will be a **feature based parametric 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** modelers it has many 2D components in order to sketch 2D shapes or extract design detail from the 3D model to create 2D production drawings, but direct 2D drawing (like **AutoCAD LT**) is not the focus, neither are animation or organic shapes (like **Maya**, **3ds Max**, **Blender** or **Cinema 4D**), although, thanks to its wide adaptability, FreeCAD might become useful in a much broader area than its current focus.

Another major concern of FreeCAD is to make heavy use of all the great open-source libraries that exist out there in the field of **Scientific Computing**. Among them are **OpenCascade**, a powerful CAD kernel, **Coin3D**, an incarnation of **OpenInventor**, **Qt**, the world-famous UI framework, and **Python**, 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**, 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 about FreeCAD's capabilities, take a look at the **Feature list**, the **latest release notes** or the **Getting started** articles, or head directly to the **User hub!**

To get started with FreeCAD, continue to the next article walking through the installation process.

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=About_FreeCAD"

Feature list

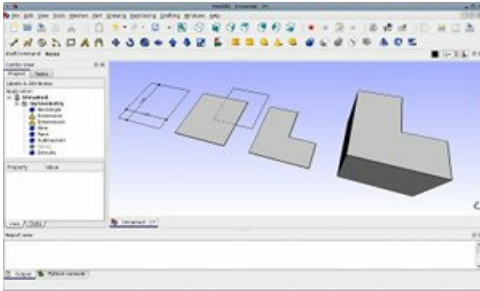
This is an extensive, hence not complete, list of features FreeCAD implements. If you want to look into the future see the [Development roadmap](#) for a quick overview the [Screenshots](#) are a nice place to go.

Release notes

- [Release 0.11](#)

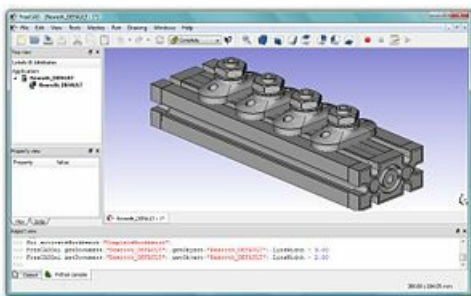
General features

Base application



- **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](#) framework, with a 3D viewer based on [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](#),** 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.
- **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.
- **Built-in [scripting](#) framework:** FreeCAD features a built-in [Python](#) 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.
- **a modular MSI installer** allows flexible installations on Windows systems. Packages for Ubuntu systems are also maintained.

Document structure



- **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.
- **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.
- **Compound (ZIP based) document save format:** FreeCAD documents saved with [.fcstd](#) extension can contain many different types of information, such as geometry, scripts or thumbnail icons.

User Interface

- **Fully customizable/scriptable Graphical User Interface.** The [Qt](#)-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.
- **Workbench concept:** In the FreeCAD interface, tools are grouped by [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.
- **Built-in Python console** with syntax highlighting, autocomplete and class browser: Python commands can be issued directly in FreeCAD and

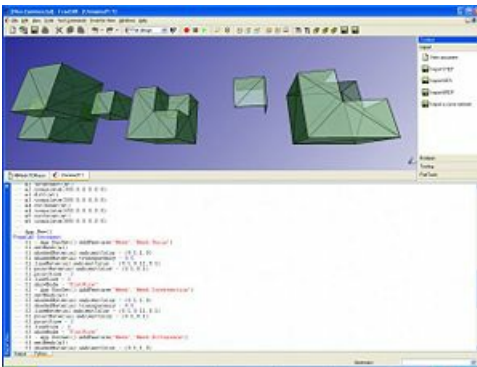
immediately return results, permitting scriptwriters to test functionality on the fly, explore the contents of the modules and easily learn about FreeCAD internals.

- **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.
- **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.

Application specific features

The functionality of FreeCAD is separated in modules, each one dealing with special data types and applications:

Meshes

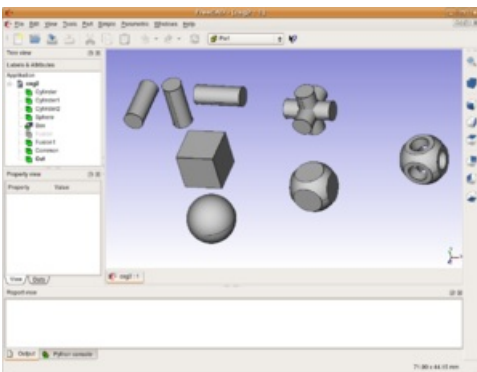


- The **Mesh Module** deals with 3D meshes. It is intended primarily for import, healing and conversion of third-party generated mesh geometry into FreeCAD, and export of FreeCAD geometry into mesh formats. But FreeCAD itself also features much more advanced geometry types than meshes.
- **Primitive creation** (box, sphere, cylinder, etc), **offset** (trivial or after Jung/Shin/Choi) or **boolean operations** (add, cut, intersect)
- **Import** of the following formats: ASCII or binary **STL (Stereo lithography format)** (*.stl, *.ast), the **OBJ format** (*.obj), limited **NASTRAN** support (*.nas), **Open Inventor** meshes (*.iv), and FreeCAD native mesh kernel (*.bms)
- **Export** of the following formats: ASCII or binary **STL (Stereo lithography format)** (*.stl, *.ast), the **OBJ format** (*.obj), limited **NASTRAN** support (*.nas, *.bri), **VRML** meshes (*.wrl), FreeCAD native mesh kernel (*.bms), mesh as Python module (*.py)
- **Testing and repairing** tools for meshes: solid test, non-two-manifolds test, self-intersection test, hole filling and uniform orientation.
- **Extensive Python scripting API.**

2D Drafting

- Graphical creation of **simple planar geometry** like lines, wires, rectangles, arcs or circles in any plane of the 3D space
- **Annotations** like texts or dimensions
- Graphical **modification operations** like translation, rotation, scaling, mirroring, offset or shape conversion, in any plane of the 3D space
- **Import** and **Export** of the following formats: Autodesk's Drawing Exchange Format (*.dxf), Open Cad Format (*.oca, *.gcad) e SVG (*.svg)

CAD



- The **Part Module** deals with everything around CAD modeling and the CAD data structures. The CAD functionality is under heavy development (see the [PartDesign_project](#) and [Assembly_project](#) in the [Development_roadmap](#)). The **Part Module** works with high-level **Open CASCADE** geometry.
- **Parametric primitive shapes** like box, sphere, cylinder, cone or torus.

- Topological components like **vertices**, **edges**, **wires** and **planes** (via python scripting).
- Modeling with straight or revolution **extrusions**, **sections** and **fillets**.
- **Boolean operations** like **union**, **difference** and **intersection**.
- **Extensive Python scripting API**.
- **Import** and **Export** of the following formats: **STEP** parts and assemblies (*.stp, *.step), **IGES** models (*.igs, *.iges) and BRep (*.brp), the native format of our **Open CASCADE** CAD kernel.

Raytracing

- The **Raytracing Module** permits the export of FreeCAD geometry to **external renderers** for generation of high-quality images. Currently, the only supported render engine is **POV-Ray**. The module currently permits the creation of a render sheet, and adding geometry to that render sheet for export to a POV-Ray file.

Drawing

- The **Drawing Module** allows to export projected views of your 3D geometry to a **2D SVG document**. It allows the creation of a 2D sheet with an existing svg template, and the insertion of projected views of your geometry in that sheet. Then the sheet can be saved as a SVG file.

CAM

- The **Cam Module** is dedicated to mechanical machining like milling. This module is at the very beginning and at the moment mostly dedicated to **Incremental Sheet Forming**. Although there are some algorithms for toolpath planing they are not usable for the end-user at the moment.

Install on Windows

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



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!](#)

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](#) 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\Program 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 on 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](#) i.e. Google [SketchUp](#).

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Install_on_Windows"

Install on Unix

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!](#)

Ubuntu

FreeCAD is available from Ubuntu repositories since version 9.04, and can be installed via the software center or with:

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

Alternatively, the freecad community provides a PPA with daily builds. This PPA is updated and built automatically every day from latest source code, and therefore contains maximum one-day old bleeding edge new features, but can also sometimes contain regressions (functionality that stops working). To add this PPA to your software sources list, do:

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

The freecad community also provides another PPA, which is updated manually every once in a while, which is therefore safer:

```
sudo add-apt-repository ppa:freecad-maintainers/freecad-dev
sudo apt-get update
sudo apt-get install freecad
```

More information on the [Download#Ubuntu_PPA_packages](#) page

Debian, Mint 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!](#)

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](#) page.



Ubuntu Lucid 32-bit

Once you downloaded the .deb corresponding to your system version, if you have the [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](#).

Installing Windows Version on Linux

See the [Install on Windows](#) page.

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Install_on_Unix"

Install on Mac

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



Mac OS X 32-bit

This page describes the usage and features of the FreeCAD installer. It also includes uninstallation instructions. Once installed, you can [get started!](#)

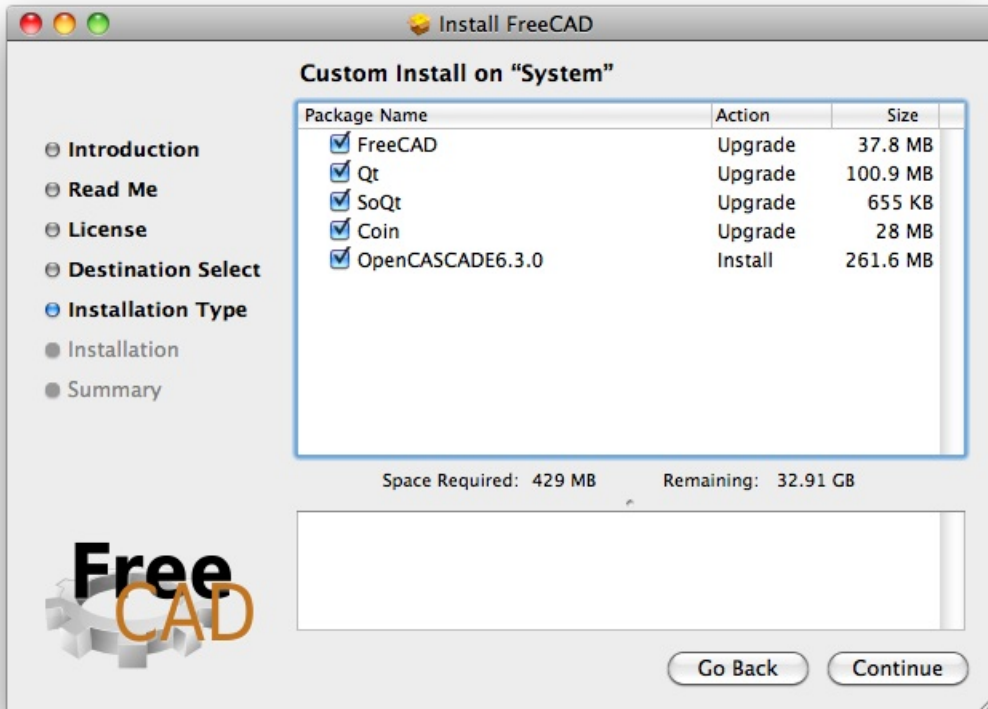
Simple Installation

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](#) page. After downloading the file, just mount the disk image, then run the **Install FreeCAD** package.



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.



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.

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Install_on_Mac"

Getting started

What's new

- [Version 0.12 Release notes](#) : Check what's new in the 0.12 release of FreeCAD

Foreword

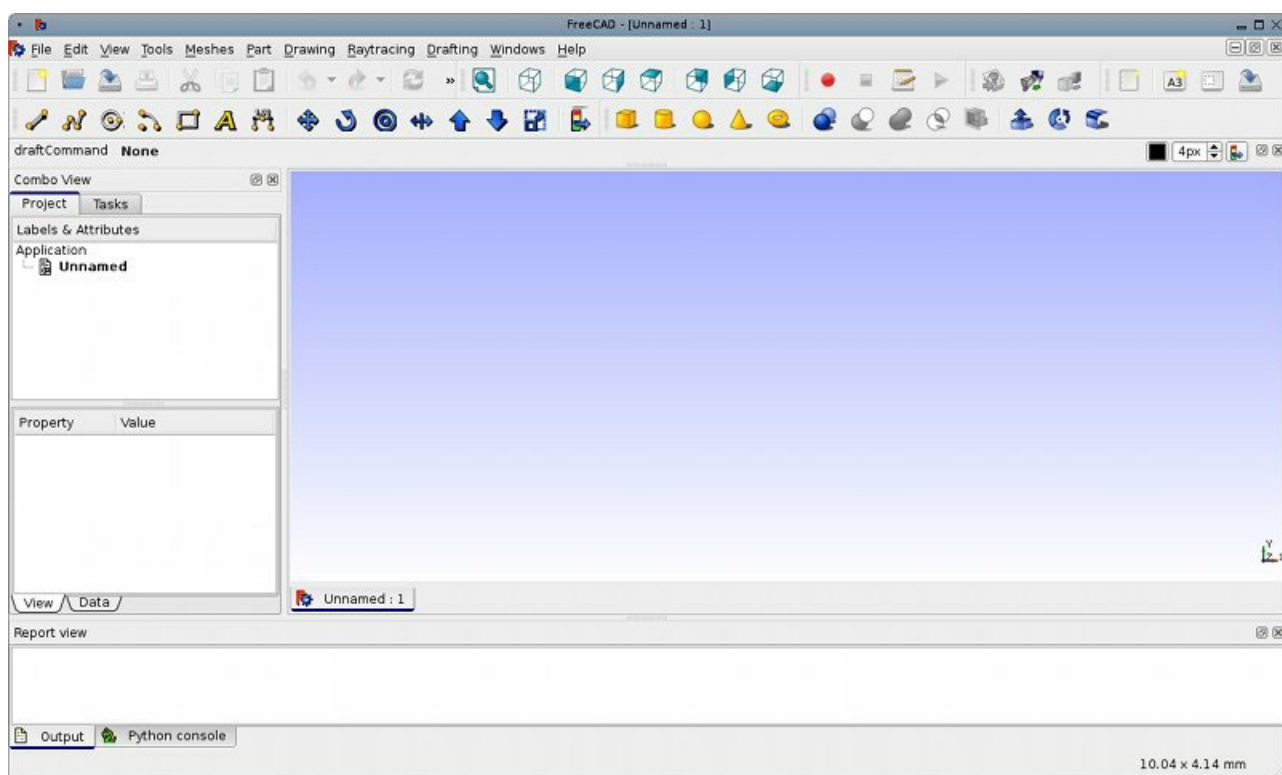
FreeCAD is a CAD/CAE parametric modeling application. It is still in early stage of development, so don't expect to be able to use it to produce work already. But, if you are curious about what FreeCAD looks like and what features are being developed, you are welcome to download it and give it a try. At the moment, much functionality is already present, but not much user interface has been created for it. This means that if you know a bit of python, you will already be able to produce and modify complex geometry relatively easily. If not, you will probably find that FreeCAD still has little to offer to you. But, be patient, this is expected to change soon.

And if after testing you have feedback, ideas or opinions, please share it with us on the [FreeCAD discussion forum](#)!

Installing

First of all (if not done already) download and install FreeCAD. See the [Download](#) page for information about current versions and updates. There are install packages ready for Windows (.msi), Ubuntu & Debian (.deb) openSUSE (.rpm) and Mac OSX.

Exploring FreeCAD



The FreeCAD interface when you start it for the first time. See more [screenshots](#) here.

FreeCAD is a general all-purpose 3D modeling application, focused on mechanical engineering and related areas, such as other engineering specialties or architecture. It is conceived as a platform for developing any kind of 3D application, but also for doing very specific tasks. For that purpose, its interface is divided into a series of [Workbenches](#). Workbenches allow to change the interface contents to display all and only the tools necessary for a specific task, or group of tasks.

The FreeCAD interface can therefore be described as a very simple container, with a menu bar, a 3D view area, and a couple of side panels for displaying the scene contents or object properties. All the contents of these panels can be changes depending on the workbench.

When you start FreeCAD for the first time, you will be presented a "general" workbench, that we call "complete workbench". This workbench simply gathers the most mature tools from other workbenches. Since FreeCAD is pretty young and not yet used for very specialized work, this workbench is very handy for discovering FreeCAD more easily. Basically, all the tools that are good enough for producing geometry are here.

Navigating in the 3D space










FreeCAD has two different [navigation modes](#) availables, that can be set in the preferences settings dialog. In the default mode, **zooming** is made with the [Mouse wheel](#), **panning** with the [Middle mouse button](#), and **rotating** with the [Left mouse button](#) and [Middle mouse button](#) simultaneously. Selecting an object is made simply by clicking on it with the [Left mouse button](#), with [CTRL](#) pressed if you want to select several objects.

You also have several view presets (top view, front view, etc) available in the View menu and on the View toolbar, and by numeric shortcuts ([1](#), [2](#), etc...)













Drawing objects

These are tools for creating objects.

-  **2-point Line**: Draws a line segment from 2 points
-  **Wire (multiple-point line)**: Draws a line made of multiple line segments
-  **Circle**: Draws a circle from center and radius
-  **Arc**: Draws an arc segment from center, radius, start angle and end angle
-  **Rectangle**: Draws a rectangle from 2 opposite points
-  **Polygon**: Draws a regular polygon from a center and a radius
-  **BSpline**: Draws a B-Spline from a serie of points
-  **Text**: Draws a multi-line text annotation
-  **Dimension**: Draws a dimension annotation

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.

-  **Move**: Moves object(s) from one location to another
-  **Rotate**: Rotates object(s) from a start angle to an end angle
-  **Offset**: Moves segments of an object about a certain distance
-  **Upgrade**: Joins objects into a higher-level object
-  **Downgrade**: Explodes objects into lower-level objects
-  **Trim/Extend (Trimex)**: Trims or extends an object
-  **Scale**: Scales selected object(s) around a base point
-  **Edit**: Edits a selected object
-  **Drawing**: Writes selected objects to a **Drawing sheet**
-  **Shape 2D View**: Creates a 2D object which is a flattened 2D view of another 3D object

Creating 3D Parts




Primitives








These are tools for creating primitive objects.

-  **Box**: Draws a box by specifying its dimensions
-  **Cone**: Draws a cone by specifying its dimensions
-  **Cylinder**: Draws a cylinder by specifying its dimensions
-  **Sphere**: Draws a sphere by specifying its dimensions
-  **Torus**: Draws a torus (ring) by specifying its dimensions

Modifying objects





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

-  **Booleans**: Performs boolean operations on objects
-  **Fuse**: Fuses (unions) two objects
-  **Common**: Extracts the common (intersection) part of two objects

-  **Cut**: Cuts (subtracts) one object from another
-  **Extrude**: Extrudes planar faces of an object
-  **Fillet**: Fillets (rounds) edges of an object
-  **Revolve**: Creates an object by revolving another object around an axis
-  **Section**: Creates a section by intersecting an object with a section plane
-  **Chamfer**: Chamfers edges of an object
-  **Mirror**: Mirrors the selected object around a given axis





Exporting to 2D Drawings

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

-  **New drawing sheet**: Creates a new drawing sheet from an existing SVG file
-  **New A3 landscape drawing**: Creates a new drawing sheet from FreeCAD's default A3 template
-  **Insert a view**: Inserts a view of the selected object in the active drawing sheet
-  **Save sheet**: Saves the current sheet as a SVG file

Exporting to external renderers

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

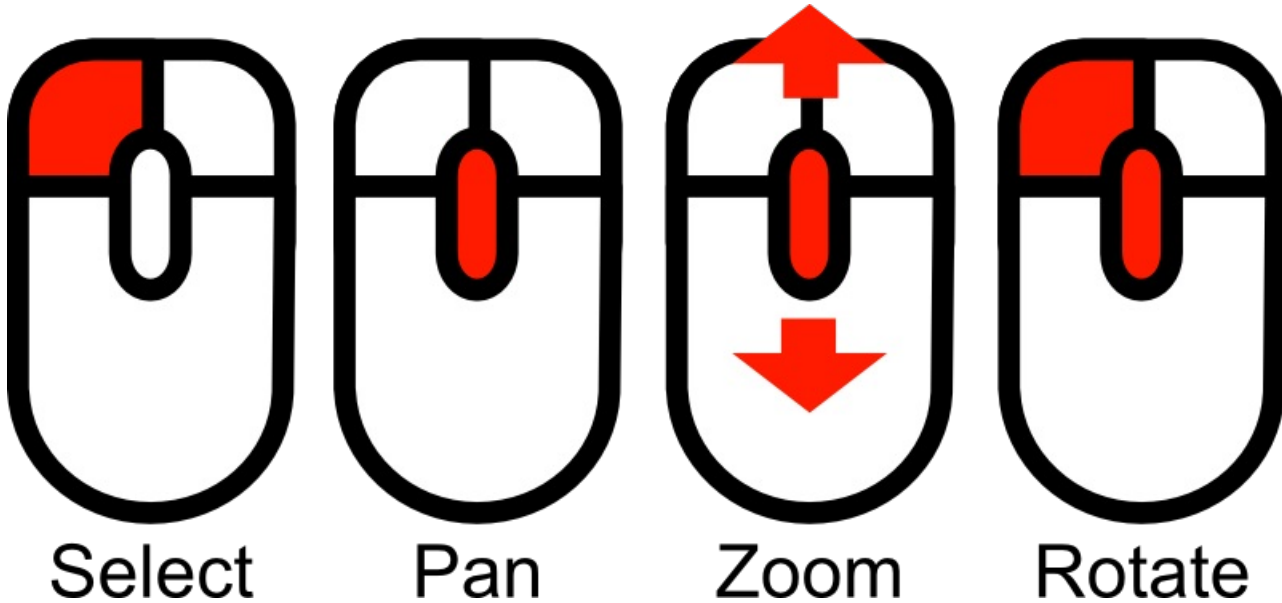
-  **New PovRay project**: Insert new PovRay project in the document
-  **Export view to povray**: Write the active 3D view with camera and all its content to a povray file
-  **Export camera to povray**: Export the camera position of the active 3D view in POV-Ray format to a file
-  **Export part to povray**: Write the selected Part (object) as a povray file

Scripting

And finally, one of the most powerful features of FreeCAD is the **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**, which provide an easy method to create custom commands.

Mouse Model

The **mouse model** of FreeCAD is very flexible and intuitive and with a few hints you can use it after only a minute of practice.



Selecting objects

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

Handling Objects

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

- Select**
Press the left mouse button over an object you want to select.
- Zoom**
Use the + or - keys or the mouse wheel to zoom in and out.
- Pan**
Click the middle mouse button and move the object around.
- Rotate**
Click first with the middle mouse button, hold it and then click the left mouse button on any visible part of an object and drag it in the desired direction. This works like spinning a ball that rotates around its center. If you release the buttons before you stop your motion, the object continues **spinning**, if this is enabled.
- Setting Center of Rotation**
A double click with the middle mouse button on any part of an object sets the new center of rotation and zooms in on this point.

Manipulating Objects

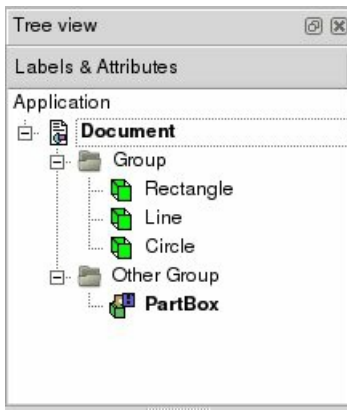
FreeCAD offers **manipulators** that can be used to modify an object or its visual appearance. A simple example is the **clipping plane** which can be activated with the *View > Clipping Plane* menu. After activation the clipping plane object appears and shows seven obvious manipulators as little boxes: 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

The SVN version of FreeCAD also supports a couple of **3D input devices**.

Document structure



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 and 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** creates mesh objects, the **Part Workbench** create Part objects, the **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. Create new views or close 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

Preferences Editor

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

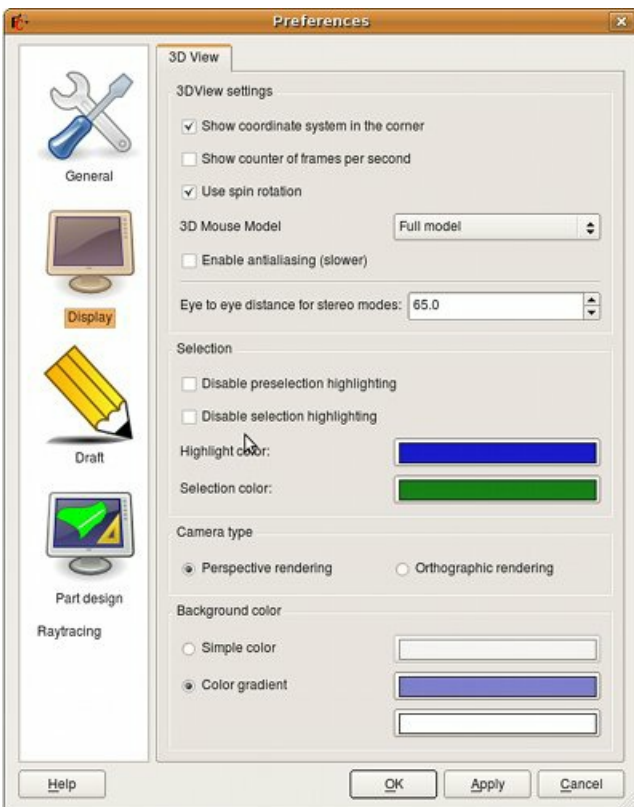
FreeCAD functionality is divided into different modules, each module being responsible for the working of a specific **workbench**. FreeCAD also uses a concept called late loading, which means that components are loaded only when they are needed. You may have noticed that when you select a workbench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



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.

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



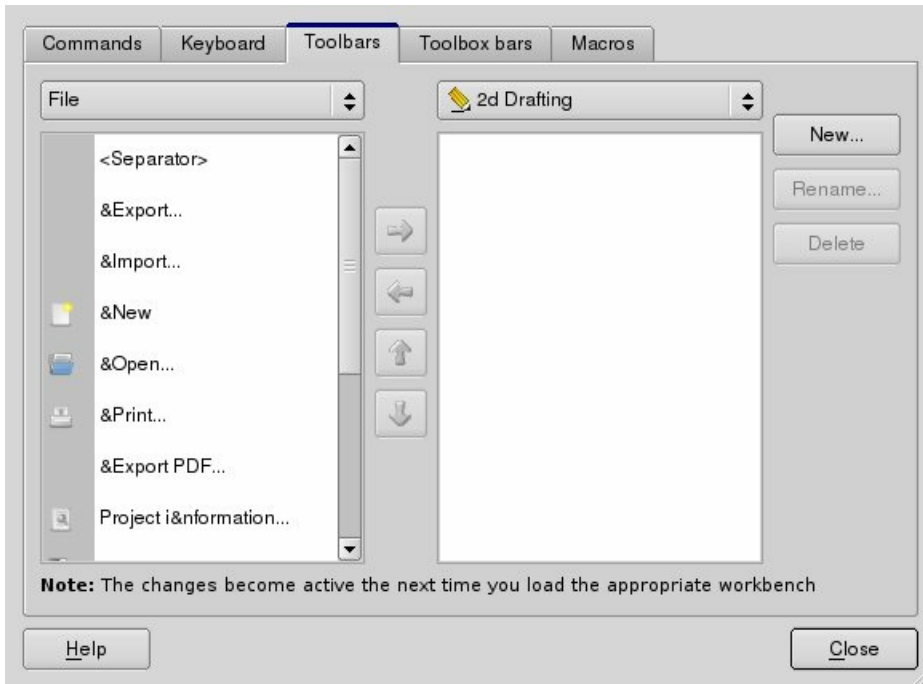
The display settings

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 meet any difficulty configuring FreeCAD to your needs.

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Preferences_Editor"

Interface Customization

Since FreeCAD interface is based on the modern **Qt** 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:



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**.

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Interface_Customization"

Property

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](#).

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](#) 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
```

Workbenches

FreeCAD, like many modern design applications such as **Revit** or **Catia**, is based on the concept of **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.

Currently we have the following workbenches available:

-  The **Part Design Workbench** for building Part shapes from sketches
-  The **Draft Workbench** for doing basic 2D CAD drafting
-  The **Mesh Workbench** for working with triangulated meshes
-  The **Part Workbench** for working with CAD parts
-  The **Image Workbench** for working with bitmap images
-  The **Raytracing Workbench** for working with ray-tracing (rendering)
-  The **Drawing workbench** for displaying your 3D work on a 2D sheet
-  The **Robot Workbench** for studying robot movements
-  The **Sketcher Workbench** for working with geometry-constrained sketches
-  The **Arch Workbench** for working with architectural elements

New workbenches are in development, stay tuned!

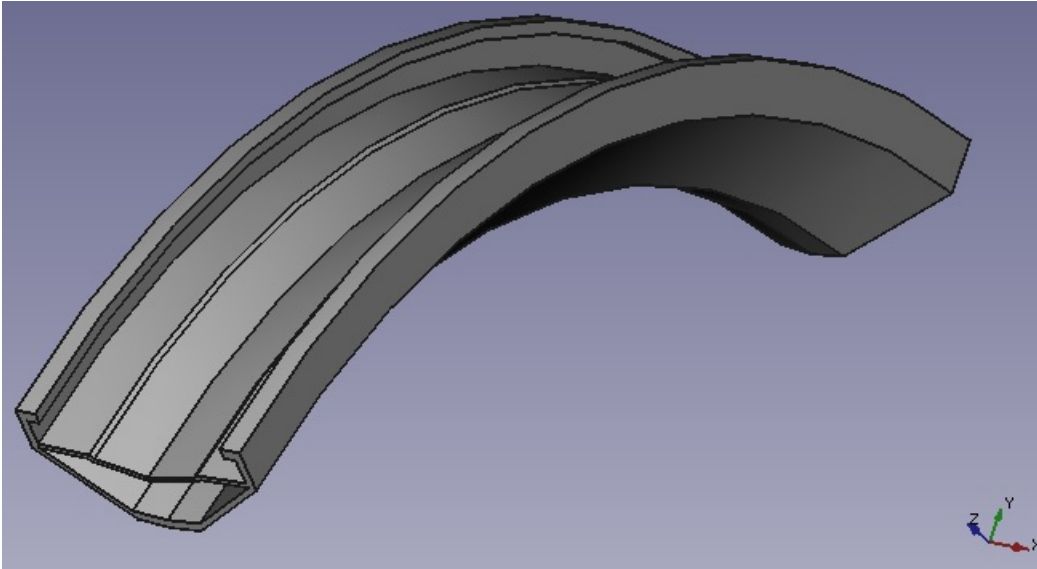
When you switch from one workbench to another, the tools available on the interface change. Toolbars, command bars and eventually other parts of the interface switch to the new workbench, but the contents of your scene doesn't change. You could, for example, start drawing 2D shapes with the Draft Workbench,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

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Workbenches>"

PartDesign Workbench

The **Part Design Workbench** aims to provide tools for modelling complex solid parts and is based on a **Feature editing methodology**. It is intricately linked with the [Sketcher Workbench](#).

This wiki page was written based on the **v0.12 version** of FreeCAD. Prior versions are missing some of the Part Design tools, or may even not include them. To get access to all the current Part Design features, go to the [Download](#) page to update your version of FreeCAD.



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 Pad (which extrudes a sketch), Pocket (which creates a pocket on an existing solid) and Revolve (which creates a solid by revolving a sketch along an axis). More tools are planned in future releases.

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](#) tools, since the Part Design module is so dependent on them.

The Sketcher Tools


Sketcher Geometries















These are tools for creating objects.

-  **Arc**: Draws an arc segment from center, radius, start angle and end angle
-  **Circle**: Draws a circle from center and radius
-  **2-point Line**: Draws a line segment from 2 points
-  **Polyline (multiple-point line)**: Draws a line made of multiple line segments
-  **Rectangle**: Draws a rectangle from 2 opposite points
-  **Fillet**: Makes a fillet between two lines joined at one point. Select both lines or click on the corner point, then activate the tool.
-  **Trimming**: Trims a line, circle or arc with respect to the clicked point.
-  **Construction Mode**: Toggles an element to/from construction mode. A construction object will not be used in a 3D geometry operation.



Sketcher Constraints

Constraints are used to set rules between sketch elements, and to lock the sketch along the vertical and horizontal axes.

-  **Lock**: Creates a lock constraint on the selected item by setting vertical and horizontal dimensions relative to the origin (dimensions can be edited afterwards).

-  **Coincident:** Creates a coincident (point-on-point) constraint between two selected points.
-  **Point On Object:** Creates a point-on-object constraint on selected items.
-  **Horizontal Distance:** Fixes the horizontal distance between 2 points or line ends. If only one item is selected, the distance is set to the origin.
-  **Vertical Distance:** Fixes the vertical distance between 2 points or line ends. If only one item is selected, the distance is set to the origin.
-  **Vertical:** Creates a vertical constraint to the selected lines or polylines elements. More than one object can be selected.
-  **Horizontal:** Creates a horizontal constraint to the selected lines or polylines elements. More than one object can be selected.
-  **Length:** Creates a length constraint on a selected line.
-  **Radius:** Creates a radius constraint on a selected arc or circle.
-  **Parallel:** Creates a parallel constraint between two selected lines.
-  **Perpendicular:** Creates a perpendicular constraint between two selected lines.
-  **Internal Angle:** Creates an internal angle constraint between two selected lines.
-  **Tangent:** Creates a tangent constraint between two selected entities, or a colinear constraint between two line segments.
-  **Equal Length:** Creates an equality constraint between two selected entities. If used on circle or arcs, the radius will be set equal.
-  **Symmetric:** Creates a symmetric constraint between 2 points with respect to a line.




Other

-  **New Sketch:** Creates a new sketch on a selected face or plane. If none were selected, the default work plane XY will be used.
-  **Leave Sketch:** Leave the Sketch editing mode.

The Part Design Tools



Construction tools

These are tools for creating solid objects.

-  **Pad:** Extrudes a solid object from a selected sketch.
-  **Pocket:** Creates a pocket from a selected sketch. The sketch must be mapped to an existing solid object's face.
-  **Revolve:** Creates a solid by revolving a sketch around an axis. The sketch must be a closed profile to get a solid object.

Modification tools

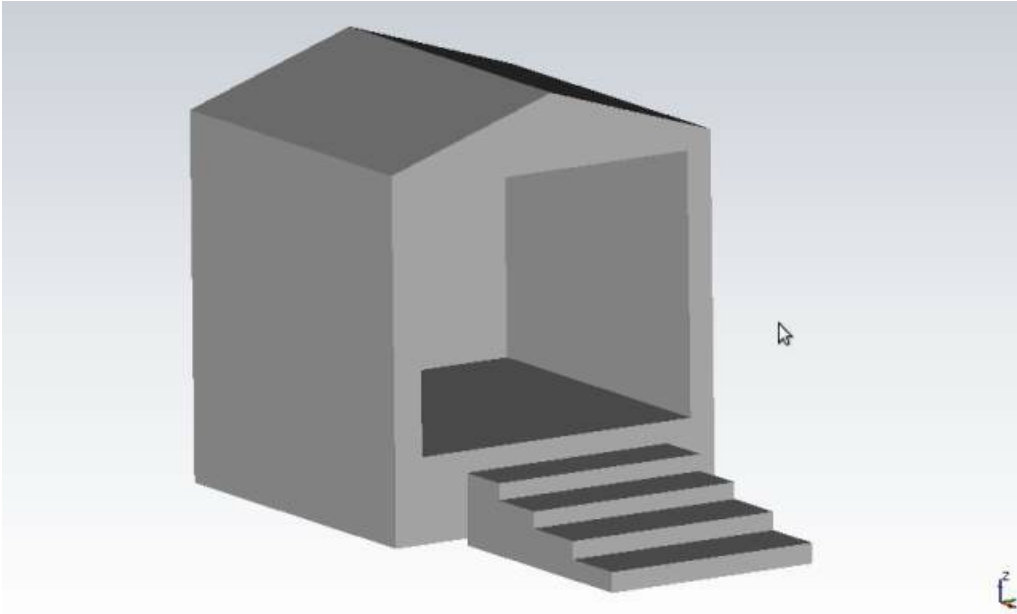
These are tools for modifying existing objects. They will allow you to choose which object to modify.

-  **Fillet:** Fillets (rounds) edges of an object.
-  **Chamfer:** Chamfers edges of an object.

Mesh Workbench

(Redirected from [Mesh Module](#))

The **Mesh Workbench** handles [triangle meshes](#). Meshes are a special type of 3D objects, composed of triangles connected by their edges and their corners (also called vertices).



An example of a mesh object

Many 3D applications use meshes as their primary type of 3D object, like [sketchup](#), [blender](#), [maya](#) or [3d studio 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 of 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 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](#).

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:

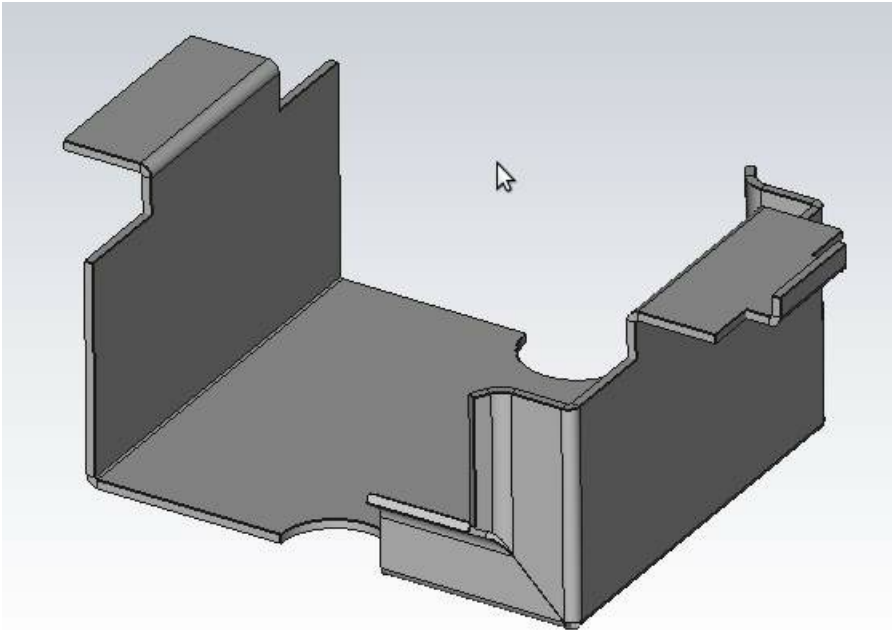
- Import meshes in several file formats
- Export meshes in several file formats
- Convert [Part](#) objects into meshes
- Analyse curvature, faces, and check if a mesh can be safely converted into a solid
- Flip mesh [normals](#)
- Close holes in meshes
- Remove faces of meshes
- Union, subtract and intersect meshes
- Create mesh primitives, like cubes, spheres, cones or cylinders
- Cut meshes along a line

These are only some of the basic operations currently present in the Mesh module interface. But the FreeCAD meshes can also be handled in many more ways by [scripting](#).

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Mesh_Workbench"

Part Module

the CAD capabilities of FreeCAD are based on the [OpenCasCade](#) 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](#) objects, are much more complex, and therefore permit much more advanced operations, like coherent booleans operations, modifications history and parametric behaviour.








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.

-  **Box**: Draws a box by specifying its dimensions
-  **Cone**: Draws a cone by specifying its dimensions
-  **Cylinder**: Draws a cylinder by specifying its dimensions
-  **Sphere**: Draws a sphere by specifying its dimensions
-  **Torus**: Draws a torus (ring) by specifying its dimensions

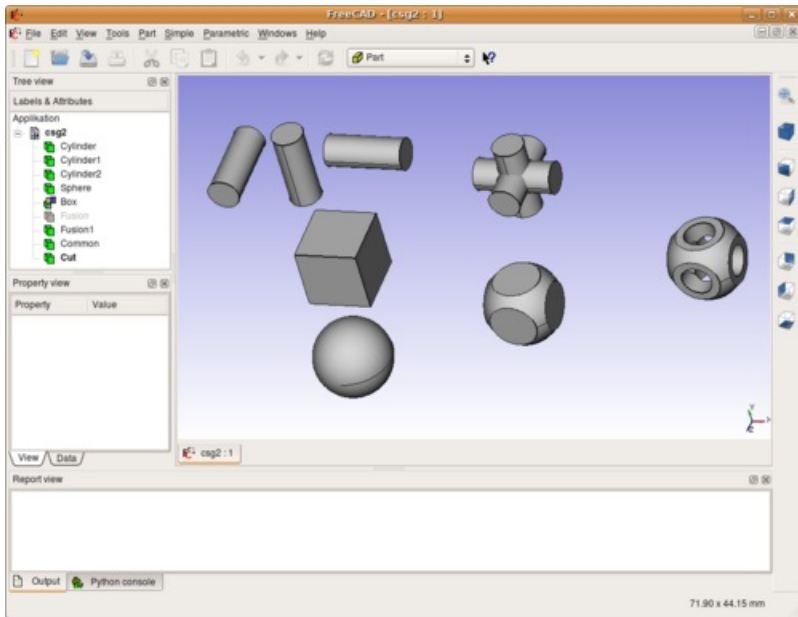
Modifying objects

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

-  **Booleans**: Performs boolean operations on objects
-  **Fuse**: Fuses (unions) two objects
-  **Common**: Extracts the common (intersection) part of two objects
-  **Cut**: Cuts (subtracts) one object from another
-  **Extrude**: Extrudes planar faces of an object
-  **Fillet**: Fillets (rounds) edges of an object
-  **Revolve**: Creates an object by revolving another object around an axis
-  **Section**: Creates a section by intersecting an object with a section plane
-  **Chamfer**: Chamfers edges of an object

-  **Mirror**: Mirrors the selected object around a given axis

Boolean Operations



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 primitive 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](#) and search for `Geom_Geometry` and `TopoDS_Shape`. There you can also read more about the differences between geometric objects and shapes. Please note that unfortunately the 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-shape a more complex shape is made of.

Scripting

The main data structure used in the Part module is the **BRep** data type from OpenCascade. About 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 creations methods exist, and for some of them, especially the TopoShapes, advanced operations like booleans union/difference/intersection are also available. Explore the contents of the Part module, as described in the [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 prepare 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 ex. Part.Line(point1,point2) or we can create a default line and set its properties afterwards, like 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](#) page if you would like to know more.





Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Part_Module"

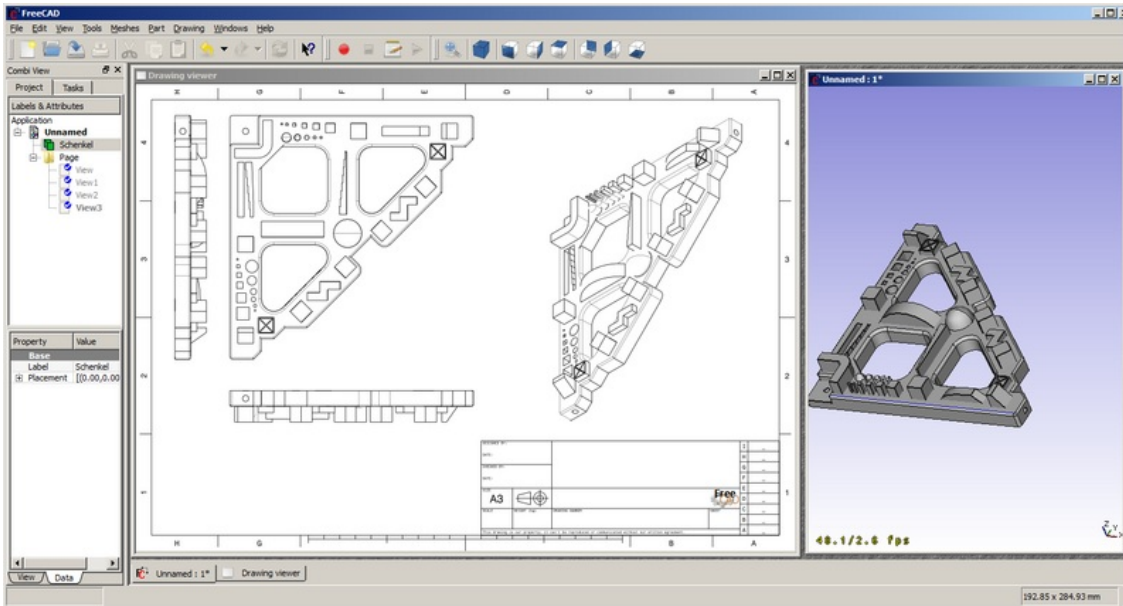
Drawing Module

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

-  **New drawing sheet:** Creates a new drawing sheet from an existing SVG file
-  **New A3 landscape drawing:** Creates a new drawing sheet from FreeCAD's default A3 template
-  **Insert a view:** Inserts a view of the selected object in the active drawing sheet
-  **Save sheet:** Saves the current sheet as a SVG file



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 are very limited, so the scripting API is more interesting. Here follows examples on how to use the scripting API of the drawing module.

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

```
# 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"
```

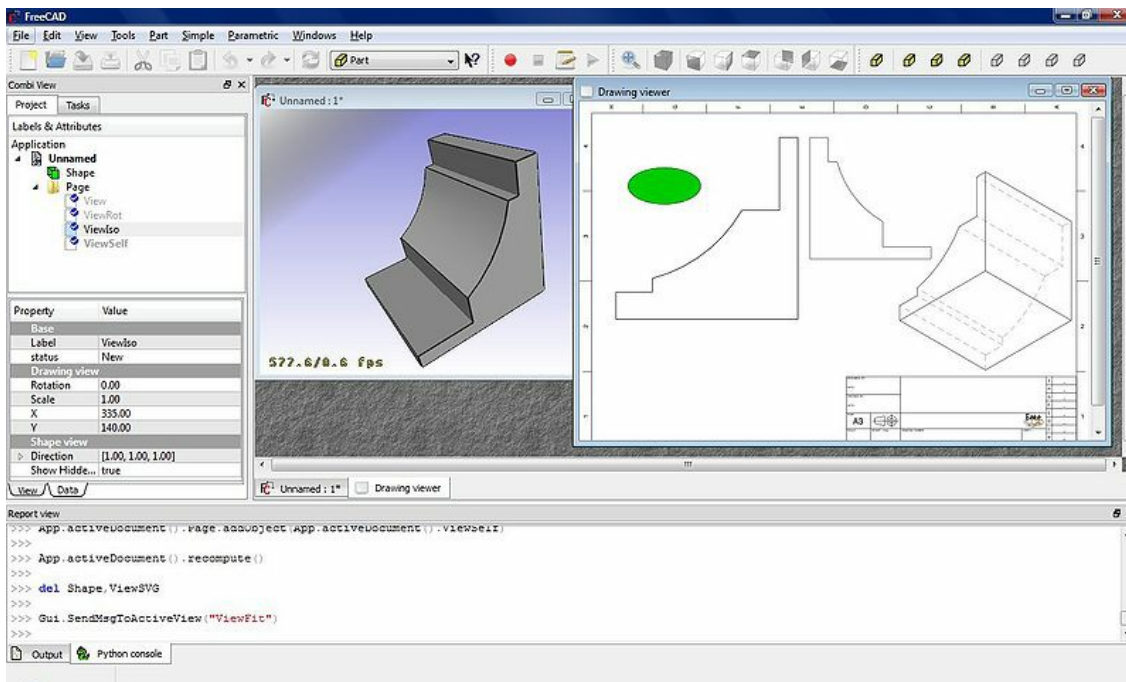
Important: free the file!

```
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 Shape, ViewSVG, resultSVG
```

That leads to the following result:



Templates

FreeCAD comes bundled with a set of default templates, but you can find more on the [Drawing templates](#) page.





Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Drawing_Module"

Raytracing Module

This module is aimed at sending the contents of your scene to an external [renderer](#), for generating photorealistic images of your work. The Raytracing module is still in very early stage of completion, so you have not many options available at the moment. Currently, only a basic set of tools to export Part objects as [POV-ray](#) files is implemented. Those files can then be loaded into POV-Ray and rendered.

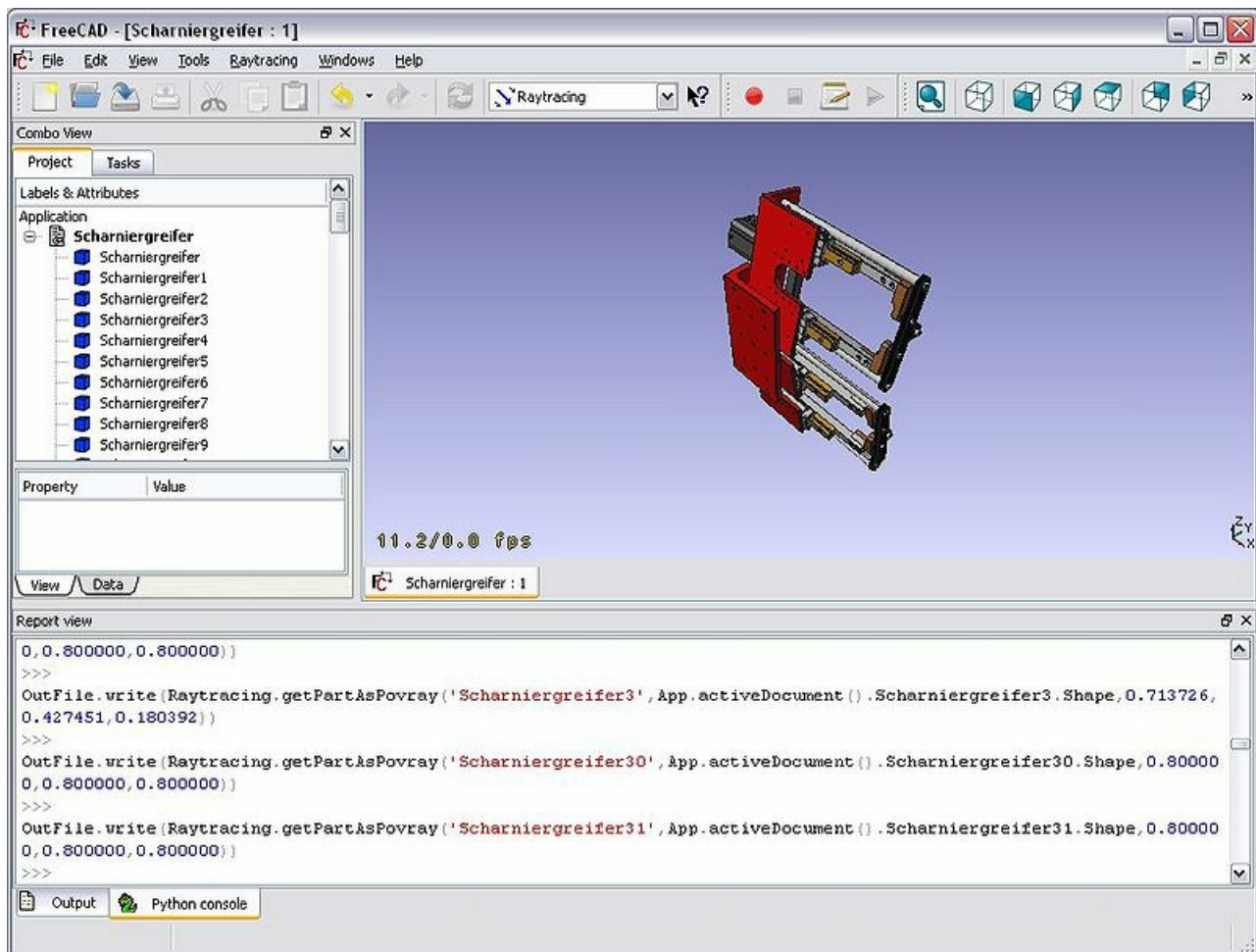
GUI Tools

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

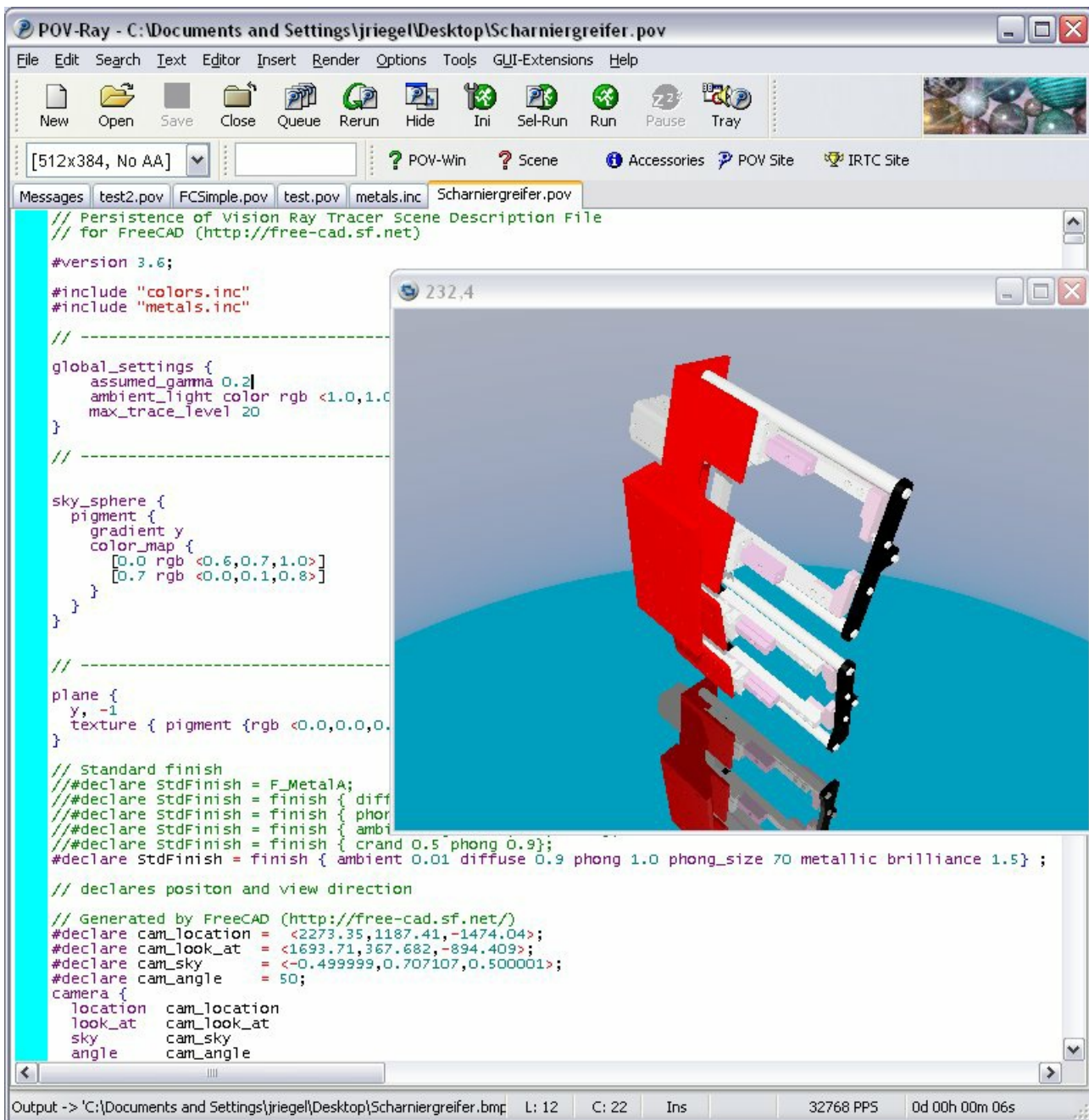
-  **New PovRay project:** Insert new PovRay project in the document
-  **Export view to povray:** Write the active 3D view with camera and all its content to a povray file
-  **Export camera to povray:** Export the camera position of the active 3D view in POV-Ray format to a file
-  **Export part to povray:** Write the selected Part (object) as a povray file

Export a View

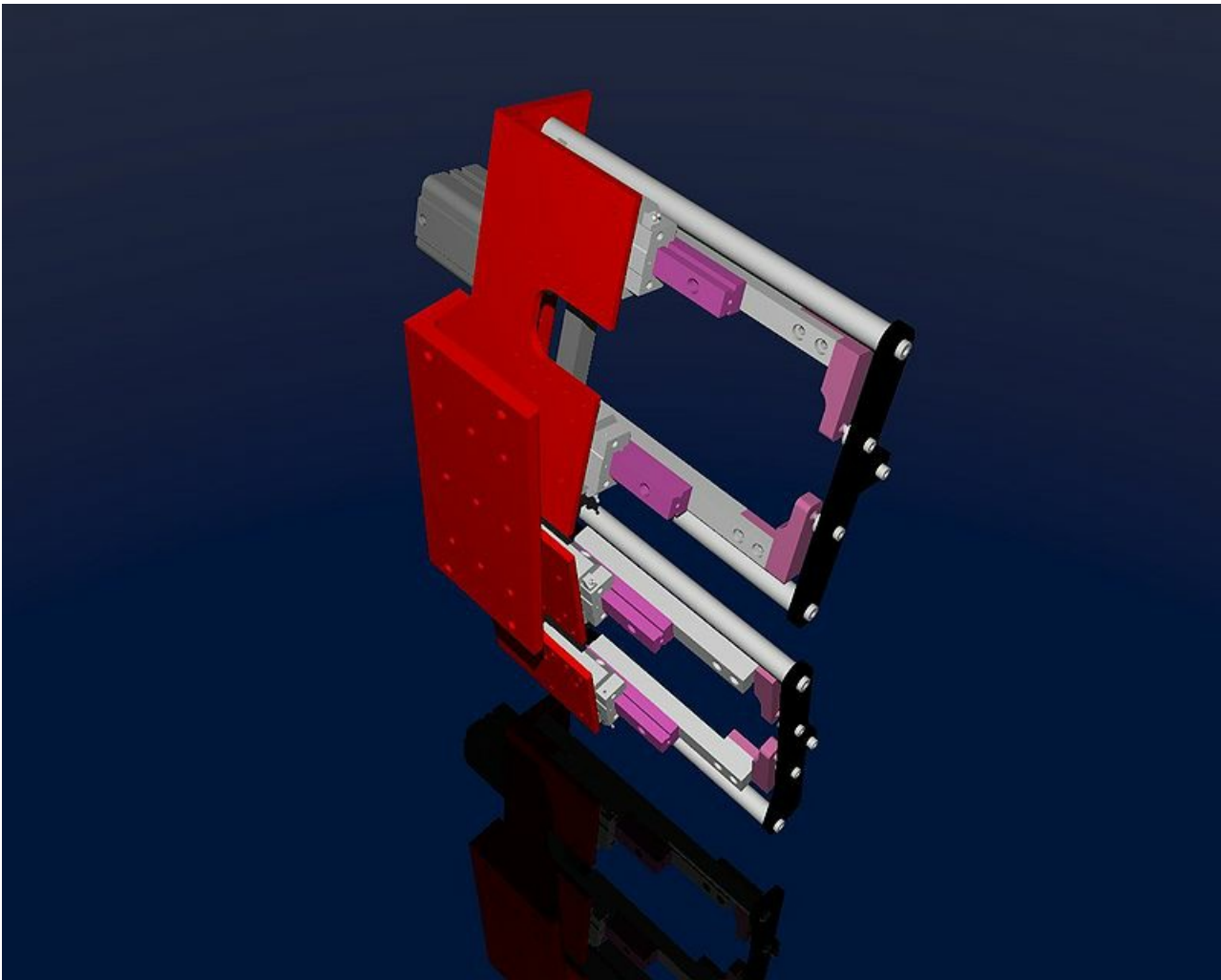
The easiest way is to export the current 3D view and all of its content to a [Povray](#) file. First, you must load or create your CAD data and position the 3D View orientation as you wish. Then choose "Export View..." from the raytracing menu.



You get asked for a location to save the resulting *.pov file. After that you can open it in [Povray](#) and render:



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



Scripting

Here is how to use these features from python:

```
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,0.800000,0.800000,0.800000))
OutFile.close()
del OutFile
```

Links

About POV-Ray:

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

About other open-source renderers (for future implementation):

- <http://www.yafaray.org/>
- <http://www.luxrender.net/>

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Raytracing_Module"

Image Module

The image module manages different types of **bitmap images**, and lets you open them in FreeCAD. Currently, the module lets you open .bmp, .jpg, .png and .xpm file formats in a separate viewer window. There is also a tool that allows you to capture an image from a webcam.

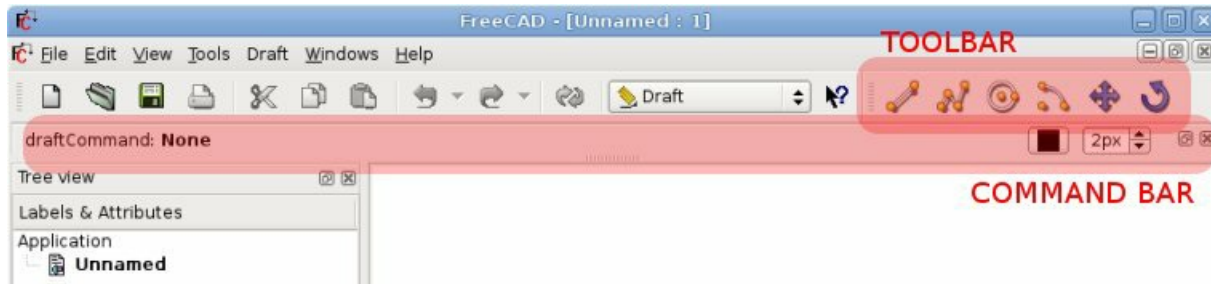
Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Image_Module"

Draft Module

(Redirected from [2d Drafting Module](#))

The draft Module is a work-in-progress and quite experimental module made to add basic 2d drawing functionality to FreeCAD. It is written entirely in python, and is also intended to showcase how far you can extend FreeCAD entirely in python, without even touching the source code.

Currently it is not really usable for production work, but already contains a couple of working basic functions. Feel free to test, and give us a feedback on the [discussion page](#).



The draft workbench is available in your workbenches list. When you activate it, two toolbars will appear; a classical **toolbar** containing the standard draft commands listed below and a special **command bar** that has no tool icon on it, but that is used by the different functions to display their controls. On that command bar, you can also change general things like current line color and width. As a sidenote, the command bar (and, IMHO, the whole QT interface) looks much better if you choose the "cleanlooks" style in FreeCAD general preferences...

Tutorial

For an in-depth explanation, read the [Draft tutorial](#) (work in progress), or read the quickstart section below to get you quickly on rails.

Quickstart

Not all Draft commands work well in 3D at the moment. So, the best thing to do is to put yourself in orthographic 2D view before starting to draw. To do that, open or create a new document, then press the **O** key (or menu view -> orthographic view) to switch to orthographic mode. Then press the **2** key (or menu view -> standard views -> top) to put you in top view. Now, you are ready to draw. You can also configure freecad (menu edit -> preferences) to always start in orthographic mode.

All draft commands follow more or less the same rules: Drawing tools will ask you to pick points on the screen or enter numeric coordinates, while modification tools will ask you to choose an object to work on first, in case no object is selected. In almost all commands, pressing the **CTRL** key will allow you to snap to existing points, **SHIFT** will constrain your movement horizontally, vertically or in relation to an existing segment, and in some tools **ALT** will give you extra options such as creating a new object instead of transforming an existing one. The **ESC** key will always cancel the active command.

Note
On some desktop systems (ex. Gnome, Kde), the **ALT** key is bound by default to moving windows on the desktop. You might need to change that shortcut key in your desktop preferences.

Some commands work in non-horizontal planes too, just make sure the Z coordinate is unlocked when drawing, and place yourself in the appropriate view. Below you will find a more complete description of all available tools.








Importing & exporting



These are functions for opening, importing or exporting other file formats. Opening will open a new document with the contents of the file, while importing will append the file content to the current document. Exporting will save the selected objects to a file. If nothing is selected, then all objects will be exported. Be aware that since the purpose of the Draft module is to work with 2d objects, those importers focus only on 2d objects, and, although DXF and OCA formats do support objects definitions in 3D space (objects are not necessarily flat), they won't import volumetric objects like meshes, 3D faces, etc, but rather lines, circles, texts or flat shapes. Currently supported file formats are:

- **Autodesk .DXF**: Imports and exports DXF files created with other CAD applications
- **SVG (as geometry)**: Imports and exports SVG files created with vector drawing applications
- **Open Cad format .OCA**: Imports and exports OCA/GCAD files, a potentially new **open CAD file format**
- **Airfoil Data Format .DAT**: Imports DAT files describing **Airfoil profiles**

Drawing objects











These are tools for creating objects.

-  **2-point Line**: Draws a line segment from 2 points
-  **Wire (multiple-point line)**: Draws a line made of multiple line segments
-  **Circle**: Draws a circle from center and radius
-  **Arc**: Draws an arc segment from center, radius, start angle and end angle
-  **Rectangle**: Draws a rectangle from 2 opposite points
-  **Polygon**: Draws a regular polygon from a center and a radius
-  **BSpline**: Draws a B-Spline from a serie of points

-  **Text**: Draws a multi-line text annotation
-  **Dimension**: Draws a dimension annotation

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.

-  **Move**: Moves object(s) from one location to another
-  **Rotate**: Rotates object(s) from a start angle to an end angle
-  **Offset**: Moves segments of an object about a certain distance
-  **Upgrade**: Joins objects into a higher-level object
-  **Downgrade**: Explodes objects into lower-level objects
-  **Trim/Extend (Trimex)**: Trims or extends an object
-  **Scale**: Scales selected object(s) around a base point
-  **Edit**: Edits a selected object
-  **Drawing**: Writes selected objects to a **Drawing sheet**
-  **Shape 2D View**: Creates a 2D object which is a flattened 2D view of another 3D object

Common behaviours

- **Snapping**: Allows to place new points on special places on existing objects
- **Constraining**: Allows to place new points horizontally or vertically in relation to previous points
- **Working with manual coordinates**: Allows to enter manual coordinates instead of clicking on screen
- **Copying**: All modification tools can either modify the selected objects or create a modified copy of them. Pressing **ALT** while using the tool will make a copy
- **Construction Mode**: Allows you to put geometry apart from the rest, for easy switch on/switch off
- **Work Plane**: All Draft commands can be used on any plane in the 3D space. The current working plane can be easily configured
- All newly created objects adopt current Draft **color and width**
- The Draft module also has its **preferences** screen

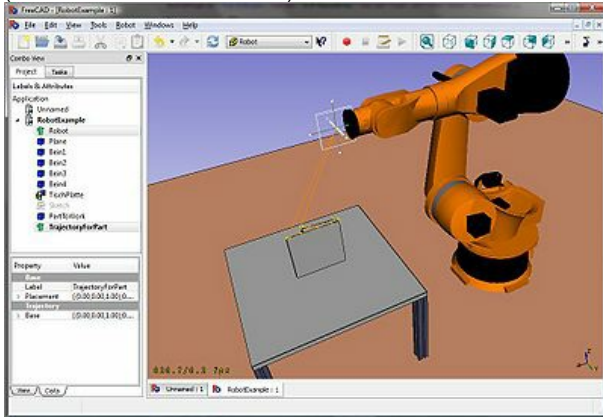
API

See the [Draft API](#) page for a complete description of the Draft functions that you can use in scripts and macros

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Draft_Module"

Robot Workbench

(Redirected from [Robot Module](#))



The robot workbench is a tool to simulate industrial grade **6-Axis Robots**, like e.g. **Kuka**. You can do following tasks:

- set up a simulation environment with a robot and work pieces
- create and fill up trajectories
- decompose features of an CAD part to a trajectory
- simulate the robot movement and reachability
- export the trajectory to a robot program file






An examples you can find here: [Example files](#) or try the [Robot tutorial](#).

Tools

Here the principal commands you can use to create a robot set-up.

Robots






The tools to create and manage the 6-Axis robots

-  **Create a robot**: Insert a new robot into the scene
-  **Simulate a trajectory**: Opens the simulation dialog and let you simulate
-  **Export a trajectory**: Export a robot program file
-  **Set home position**: Set the home position of an robot
-  **Restore home position**: move the robot to its home position




Trajectories

Tools to creat and manipulate trajectories. There are two kinds, the parametric and non parametric ones.

non parametric

-  **Create a trajectory**: Insert a new robot into the scene
-  **Set the default orientation**: Set the orientation way-points gets created by default
-  **Set the default speed parameter**: set the defaults for way-point creation
-  **Insert a waypoint**: Insert a way-point from the current robot position into a trajectory
-  **Insert a waypoint**: Insert a way-point from the current mouse position into a trajectory

parametric

-  **Create a trajectory out of edges**: Insert a new object which decompose edges to a trajectory
-  **Dress-up a trajectory**: Let you override one or more properties of a trajectory
-  **Trajectory compound**: create a compound out of some single trajectories

Scripting

This section is generated out of: <http://free-cad.svn.sourceforge.net/viewvc/free-cad/trunk/src/Mod/Robot/RobotExample.py?view=markup> You can use this file directly if you want.

Example how to use the basic robot class `Robot6Axis` which represents a 6-axis industrial robot. The Robot module is dependent on Part but not on other modules. It works mostly with the basic types `Placement`, `Vector` and `Matrix`. So we need only:

```
from Robot import *
from Part import *
from FreeCAD import *
```

Basic robot stuff

create the robot. If you do not specify another kinematic it becomes a Puma 560

```
rob = Robot6Axis()
print rob
```

accessing the axis and the Tcp. Axes go from 1-6 and are in degree:

```
Start = rob.Tcp
print Start
print rob.Axis1
```

move the first axis of the robot:

```
rob.Axis1 = 5.0
```

the Tcp has changed (forward kinematic)

```
print rob.Tcp
```

move the robot back to start position (reverse kinematic):

```
rob.Tcp = Start
print rob.Axis1
```

the same with axis 2:

```
rob.Axis2 = 5.0
print rob.Tcp
rob.Tcp = Start
print rob.Axis2
```

Waypoints:

```
w = Waypoint(Placement(), name="Pt", type="LIN")
print w.Name, w.Type, w.Pos, w.Cont, w.Velocity, w.Base, w.Tool
```

generate more. The trajectory always finds automatically a unique name for the waypoints

```
l = [w]
for i in range(5):
    l.append(Waypoint(Placement(Vector(0,0,i*100),Vector(1,0,0),0),"LIN","Pt"))
```

create a trajectory

```
t = Trajectory(l)
print t
for i in range(7):
    t.insertWaypoints(Waypoint(Placement(Vector(0,0,i*100+500),Vector(1,0,0),0),"LIN","Pt"))
```

see a list of all waypoints:

```
print t.Waypoints
del rob, Start, t, l, w
```

working with the document

Working with the robot document objects: first create a robot in the active document

```
if(App.activeDocument() == None):App.newDocument()
App.activeDocument().addObject("Robot::RobotObject", "Robot")
```

Define the visual representation and the kinematic definition (see [6-Axis Robot](#) and [VRML Preparation for Robot Simulation](#) for details about that)

```
App.activeDocument().Robot.RobotVrmlFile = App.getResourceDir()+"Mod/Robot/Lib/Kuka/kr500_1.wrl"
App.activeDocument().Robot.RobotKinematicFile = App.getResourceDir()+"Mod/Robot/Lib/Kuka/kr500_1.csv"
```

start position of the Axis (only that which differ from 0)

```
App.activeDocument().Robot.Axis2 = -90
App.activeDocument().Robot.Axis3 = 90
```

retrieve the Tcp position

```
pos = FreeCAD.getDocument("Unnamed").getObject("Robot").Tcp
```

move the robot

```
pos.move(App.Vector(-10,0,0))
FreeCAD.getDocument("Unnamed").getObject("Robot").Tcp = pos
```

create an empty Trajectory object in the active document

```
App.activeDocument().addObject("Robot::TrajectoryObject", "Trajectory")
```

get the Trajectory

```
t = App.activeDocument().Trajectory.Trajectory
```

add the actual TCP position of the robot to the trajectory

```
StartTcp = App.activeDocument().Robot.Tcp
t.insertWaypoints(StartTcp)
App.activeDocument().Trajectory.Trajectory = t
print App.activeDocument().Trajectory.Trajectory
```

insert some more Waypoints and the start point at the end again:

```
for i in range(7):
    t.insertWaypoints(Waypoint(Placement(Vector(0,1000,i*100+500),Vector(1,0,0),i),"LIN","Pt"))

t.insertWaypoints(StartTcp) # end point of the trajectory
App.activeDocument().Trajectory.Trajectory = t
print App.activeDocument().Trajectory.Trajectory
```

Simulation

To be done..... ;-)

Exporting the trajectory

The trajectory is exported by Python. That means for every control cabinet type there is a post-processor Python module. Here is in detail the Kuka post-processor described

```
from KukaExporter import ExportCompactSub

ExportCompactSub(App.activeDocument().Robot,App.activeDocument().Trajectory,'D:/Temp/TestOut.src')
```

and that's kind of how it's done:

```
for w in App.activeDocument().Trajectory.Trajectory.Waypoints:
    (A,B,C) = (w.Pos.Rotation.toEuler())
    print ("LIN {X %.3f,Y %.3f,Z %.3f,A %.3f,B %.3f,C %.3f} ; %s"%(w.Pos.Base.x,w.Pos.Base.y,w.Pos.Base.z,A,B,C,w.Name))
```

Online version: http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Robot_Workbench

Category:Command Reference

(Redirected from [List of Commands](#))

This category contains the help pages of each of FreeCAD command.

Pages in category "Command Reference"

The following 173 pages are in this category, out of 173 total.

A

- [Arch Add](#)
- [Arch Building](#)
- [Arch Cell](#)
- [Arch Floor](#)
- [Arch Remove](#)
- [Arch SectionPlane](#)
- [Arch Site](#)
- [Arch Structure](#)
- [Arch Wall](#)
- [Arch Window](#)

C

- [Constraint EqualLength](#)
- [Constraint Horizontal](#)
- [Constraint HorizontalDistance](#)
- [Constraint InternalAngle](#)
- [Constraint Length](#)
- [Constraint Lock](#)
- [Constraint Parallel](#)
- [Constraint Perpendicular](#)
- [Constraint PointOnObject](#)
- [Constraint PointOnPoint](#)
- [Constraint Radius](#)
- [Constraint Symmetric](#)
- [Constraint Tangent](#)
- [Constraint Vertical](#)
- [Constraint VerticalDistance](#)

D

- [Draft Apply](#)
- [Draft Arc](#)
- [Draft BSpline](#)
- [Draft Circle](#)
- [Draft Dimension](#)
- [Draft Downgrade](#)
- [Draft Drawing](#)
- [Draft Edit](#)
- [Draft Line](#)
- [Draft Move](#)
- [Draft Offset](#)
- [Draft Polygon](#)
- [Draft Rectangle](#)
- [Draft Rotate](#)
- [Draft Scale](#)
- [Draft Shape2DView](#)
- [Draft Text](#)
- [Draft ToggleDisplayMode](#)
- [Draft Trimex](#)
- [Draft Upgrade](#)
- [Draft Wire](#)

G

- [GuiCommand model](#)

M

M cont.

- [Mesh ExMakeUnion](#)
- [Mesh FillInteractiveHole](#)
- [Mesh FillupHoles](#)
- [Mesh FixDegenerations](#)
- [Mesh FixDuplicateFaces](#)
- [Mesh FixDuplicatePoints](#)
- [Mesh FixIndices](#)
- [Mesh FlipNormals](#)
- [Mesh FromGeometry](#)
- [Mesh HarmonizeNormals](#)
- [Mesh Import](#)
- [Mesh Intersection](#)
- [Mesh PolyCut](#)
- [Mesh PolySegm](#)
- [Mesh PolySplit](#)
- [Mesh RemoveCompByHand](#)
- [Mesh RemoveComponents](#)
- [Mesh ToolMesh](#)
- [Mesh Transform](#)
- [Mesh Union](#)
- [Mesh VertexCurvature](#)

P

- [Part Booleans](#)
- [Part Box](#)
- [Part Chamfer](#)
- [Part Common](#)
- [Part Cone](#)
- [Part Cut](#)
- [Part Cylinder](#)
- [Part Extrude](#)
- [Part Fillet](#)
- [Part Fuse](#)
- [Part Mirror](#)
- [Part Revolve](#)
- [Part Section](#)
- [Part Sphere](#)
- [Part Torus](#)
- [PartDesign Pad](#)
- [PartDesign Pocket](#)
- [PartDesign Revolve](#)

R

- [Robot CreateRobot](#)
- [Robot CreateTrajectory](#)
- [Robot Edge2Trac](#)
- [Robot Export](#)
- [Robot InsertWaypoint](#)
- [Robot InsertWaypointPre](#)
- [Robot RestoreHomePos](#)
- [Robot SetDefaultOrientation](#)
- [Robot SetDefaultValues](#)
- [Robot SetHomePos](#)
- [Robot Simulate](#)
- [Robot TrajectoryCompound](#)
- [Robot TrajectoryDressUp](#)

S cont.

- [Sketcher NewSketch](#)
- [Sketcher Rectangle](#)
- [Sketcher Wire](#)
- [Std About](#)
- [Std AboutQt](#)
- [Std CommandLine](#)
- [Std Copy](#)
- [Std Cut](#)
- [Std Delete](#)
- [Std DlgCustomize](#)
- [Std DlgMacroExecute](#)
- [Std DlgMacroExecuteDirect](#)
- [Std DlgMacroRecord](#)
- [Std DlgMacroStop](#)
- [Std DlgParameter](#)
- [Std DlgPreferences](#)
- [Std Export](#)
- [Std FreeCADWebsite](#)
- [Std FreezeViews](#)
- [Std Import](#)
- [Std MeasureDistance](#)
- [Std New](#)
- [Std OnlineHelp](#)
- [Std OnlineHelpPython](#)
- [Std OnlineHelpWebsite](#)
- [Std Open](#)
- [Std OrthographicCamera](#)
- [Std Paste](#)
- [Std PerspectiveCamera](#)
- [Std Print](#)
- [Std PrintPdf](#)
- [Std ProjectInfo](#)
- [Std PythonWebsite](#)
- [Std Quit](#)
- [Std RecentFiles](#)
- [Std Redo](#)
- [Std SaveAs](#)
- [Std SceneInspector](#)
- [Std SelectAll](#)
- [Std SetAppearance](#)
- [Std TipOfTheDay](#)
- [Std ToggleClipPlane](#)
- [Std ToggleVisibility](#)
- [Std TreeSelection](#)
- [Std ViewBoxZoom](#)
- [Std ViewCreate](#)
- [Std ViewDockUndockFullscreen](#)
- [Std ViewExamples](#)
- [Std ViewFitAll](#)
- [Std ViewFitSelection](#)
- [Std ViewIvIssueCamPos](#)
- [Std ViewIvStereo](#)
- [Std ViewScreenShot](#)
- [Std ViewXX](#)
- [Std ViewZoom](#)
- [Std WhatsThis](#)
- [Std Workbench](#)

T

- [Mesh BoundingBox](#)
- [Mesh BuildRegularSolid](#)
- [Mesh CurvatureInfo](#)
- [Mesh Demolding](#)
- [Mesh Difference](#)
- [Mesh EvaluateFacet](#)
- [Mesh EvaluateSolid](#)
- [Mesh Evaluation](#)
- [Mesh ExMakeMesh](#)
- [Mesh ExMakeTool](#)

S

- [Sketcher Arc](#)
- [Sketcher Circle](#)
- [Sketcher ConstructionMode](#)
- [Sketcher Fillet](#)
- [Sketcher LeaveSketch](#)
- [Sketcher Line](#)

- [Template:GuiCommand](#)

Category:Tutorials

(Redirected from [Tutorials](#))

FreeCAD tutorials. Please help us adding more!

Pages in category "Tutorials"

The following 7 pages are in this category, out of 7 total.

B

- [Basic modeling tutorial](#)

D

- [Draft tutorial](#)

E

- [Engine Block Tutorial](#)
- [Engine tutorial](#)

P

- [Python scripting tutorial](#)

V

- [VRML Preparation for Robot Simulation](#)
- [Video tutorials](#)

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

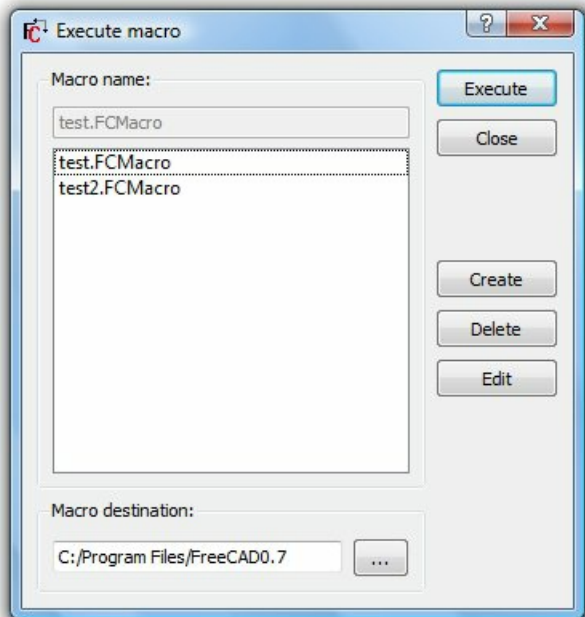
If you enable console output (Menu Edit -> Preferences -> General -> Macros -> Show scripts commands in python console), you will see that in FreeCAD, every action you do, such as pressing a button, outputs a python command. Those commands are what can be recorded in a macro. The main

tool for making macros is the macros toolbar:



. 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:



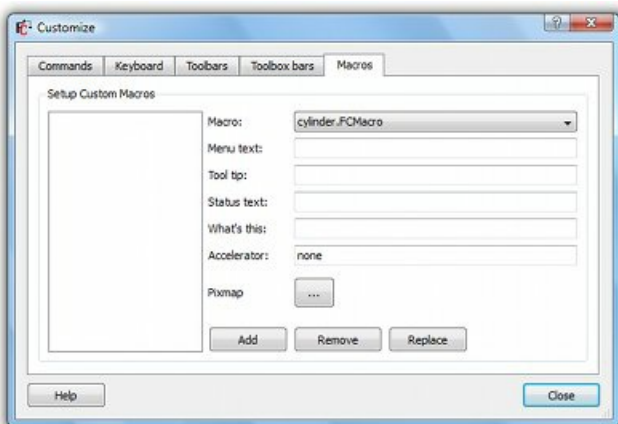
There you can manage your macros, delete, edit or create new ones from scratch. If you edit a macro, it will be opened in an editor window where you can make changes to its code.

Example

Press the record button, give a name, let's say "cylinder 10x10", then, in the [Part Workbench](#), 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:



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](#) page if you want to know more about python scripting...

Creating macros without recording

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](#) page to pick some useful macros to add to your FreeCAD installation.

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Macros>"

Introduction to Python

(Redirected from [Introduction to python](#))

This is a short tutorial made for who is totally new to Python. [Python](#) is an open-source, multiplatform [programming language](#). Python has several features that make it very different than other common programming languages, and very accessible to new users like yourself.

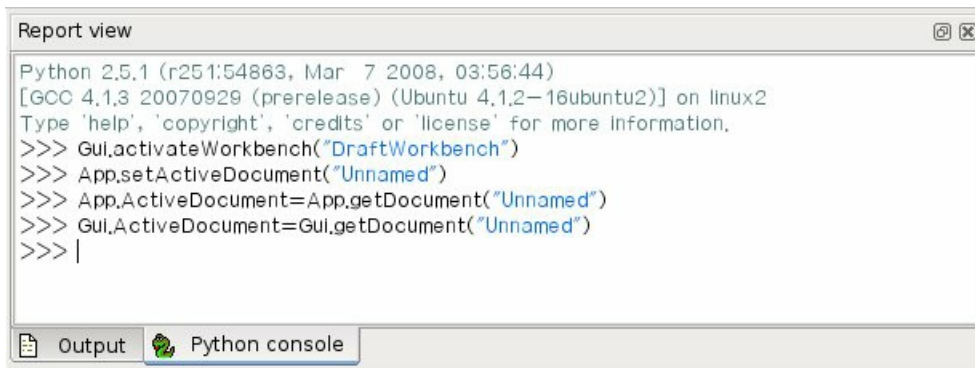
- It has been designed specially to be easy to read by human beings, and so it is very easy to learn and understand.
- It is interpreted, that is, unlike compiled languages like C, your program doesn't need to be compiled before it is executed. The code you write can be immediately executed, line by 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](#) 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:



(If you don't have it, click on View â†’ Views â†’ 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:

```
print "hello"
```

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:

```
print hello
```

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:

```
help
```

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:

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

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:

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

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:

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

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

```
print = "hello"
```

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:

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

We changed the value of myVariable. We can also copy variables:

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

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:

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

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:

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

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:

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

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 float numbers can have a decimal part:

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

Int and Floats can be mixed together without problem:

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

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:

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

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:

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

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:

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

We can also use int() and float() to convert to int and float if we want:

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

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:

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

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:

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

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 []:

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

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:

```
len(myOtherList)
```

or retrieving one item of a list:

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

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](#), 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:

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

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](#), or you can even create your own data types with [classes](#).

Indentation

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

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

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:

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

Or more complex things like this:

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

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:

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

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:

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

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

```
if "Lucky" in alldaltons:
```

Then nothing is printed. We can also specify an else: statement:

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

Functions

The **standard Python commands** 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 of 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:

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

printsqm(45)
```

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:

```
printsqm(45, 34)
```

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:

```
def sum(val1, val2):
```

```
total = val1 + val2
return total
```

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

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:

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

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:

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

to have something printed. Read more about functions [here](#).

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:

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

print "test.py succesfully loaded"
```

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

```
import test
```

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:

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

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

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

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:

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

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:

```
help()
modules
```

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

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

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?

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

And finally one last little goodie: When we work on programming a new module, we often want to test it. So once we wrote a little piece of module, in a python interpreter, we do something like this, to test our new code:

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

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:

```
reload(myModule)
```

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

```
dir()
```

and read on to [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 two very important Python reference documents on the net:

- the [official Python reference](#)
- the [Dive into Python](#) wikibook

Be sure to bookmark them!

FreeCAD Scripting Basics

Python scripting in FreeCAD

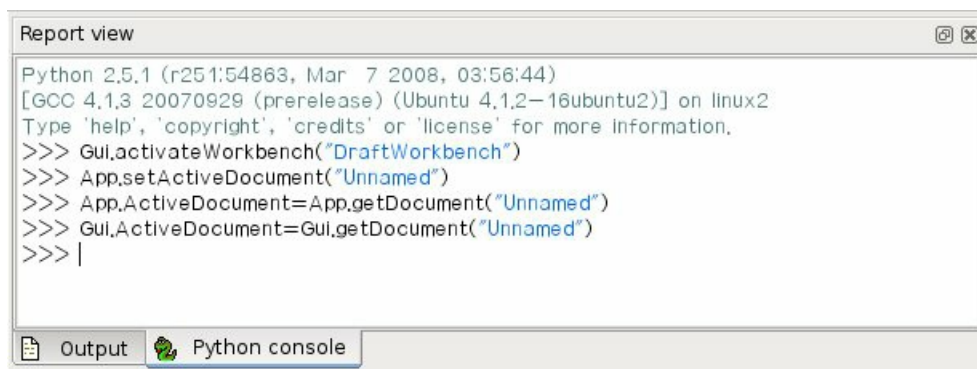
FreeCAD is built from scratch to be totally controlled by Python scripts. Almost all parts of FreeCAD such as the interface, the scene contents, and even the representation of this content in the 3d views are accessible from the built-in Python interpreter or from your own scripts. As a result, FreeCAD 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 on your 3D objects, mainly because it is still in early stage 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 to 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 from simple commands 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 "Report view" as shown below, you can activate it under the View -> Views -> Report view to bring up the interpreter.

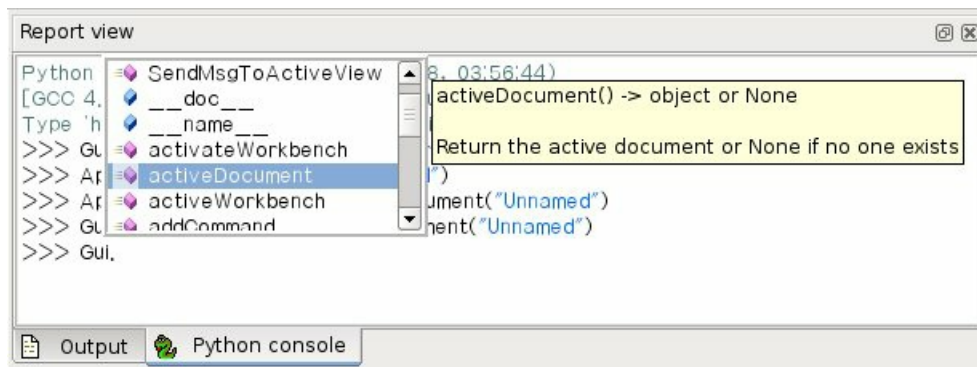
The interpreter

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:



```
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")
>>> |
```

From the interpreter, you can execute Python code and browse through the available classes and function. 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 existing) is displayed:



```
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")
>>> Gui.
activeDocument() -> object or None
Return the active document or None if no one exists
```

So, start here by typing **App.** or **Gui.** and see what happens. Another more generic Python way of exploring contents 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 in command history and retrieve a line of code you already typed earlier. To navigate in 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 to experiment something here, then make a full script of it), or insert 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 in documenting his code, but usually Python module have the reputation to be 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 Graphic User Interface, almost all its functionality is separated in 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 respective names of FreeCAD and FreeCADGui.

- 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 interesting, 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 along FreeCAD development. But the two browsing tools provided (the class browser and the Python help) should give you, at any moment, a 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, such as faces color, drawing mode, that are stored in a corresponding Gui object.

This way of doing allows a very wide range of uses, like having algorithms work only on the defining 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, exists a corresponding Gui object. The document itself, actually, also has 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 generated again everytime an App object is marked as "to be recomputed" (for example when one of its parameters changed), so changes you might have done directly to the Gui object might get lost.

To access the App part of something, you type:

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

where "ObjectName" is the name of your object. You can also type:

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

To access the Gui part of the same object, you type:

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

where "ObjectName" is the name of your object. You can also type:

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

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:

```
myDocument = App.ActiveDocument
```

To create a new document, type:

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

To access the Gui part the currently open (active) document, you type:

```
myGuiDocument = Gui.ActiveDocument
```

To access the current view, you type:

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

Using additional modules

The FreeCAD and FreeCADGui modules are solely responsables 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** uses the OpenCascade kernel, and therefore is able to create and manipulate **B-rep** type geometry, which is what OpenCascade is built for. The **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:

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

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):

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

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:

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

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:

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

Modifying objects

Modifying an object is done the same way:

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

Now let's change the shape by a bigger one:

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

Querying objects

You can always look at the type of an object like this:

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

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

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

Now you can really start playing with FreeCAD! To look at what you can do with the [Part Module](#), read the [Part scripting](#) page, or the [Mesh Scripting](#) page for working with the [Mesh Module](#). Note that, although the Part and Mesh modules are the most complete and widely used, other modules such as the [Draft Module](#) also have [scripting](#) APIs that can be useful to you. For a complete list of each modules and their available tools, visit the [Category:API](#) section.

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=FreeCAD_Scripting_Basics"

Mesh Scripting

Introduction

First of all you have to import the Mesh module:

```
import Mesh
```

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:

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

You can also create an object from a file

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

(A list of compatible filetypes can be found under 'Meshes' [here](#).)

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

```
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)
```

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.

```
import BuildRegularGeoms
```

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:

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

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.

```
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
```

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

```
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.difference(cylinder)  
d = FreeCAD.newDocument()  
d.addObject("Mesh::Feature", "Diff_Sphere_Cylinder").Mesh=diff  
d.recompute()
```

Examining and Testing

Write your own Algorithms

Exporting

You can even write the mesh to a python module:

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

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](#).

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Mesh_Scripting"

Topological data scripting

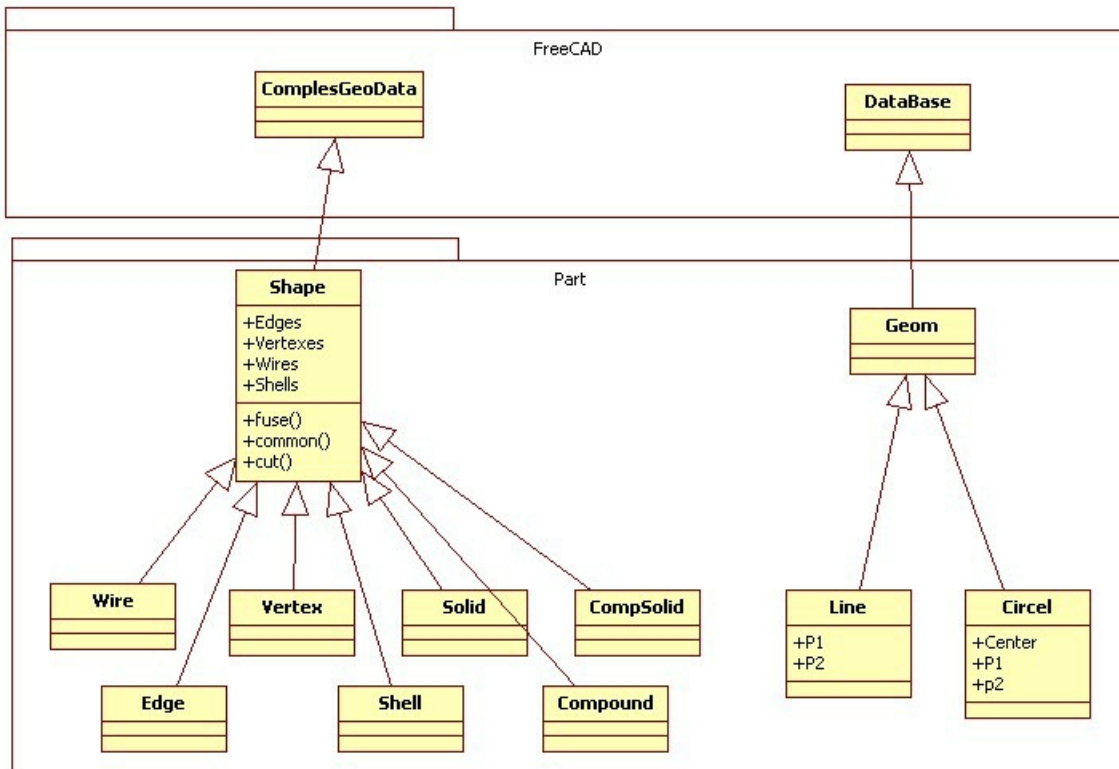
This page describes several methods for creating and modifying **Part shapes** from python. Before reading this page, if you are new to python, it is a good idea to read about [python scripting](#) and [how python scripting works in FreeCAD](#).

Introduction

We will here explain you how to control the **Part Module** directly from the FreeCAD python interpreter, or from any external script. Be sure to browse the [Scripting](#) section and the [FreeCAD Scripting Basics](#) pages if you need more information about how python scripting works in FreeCAD.

Class Diagram

This is a [Unified Modeling Language \(UML\)](#) overview of the most important classes of the Part module:



Geometry

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

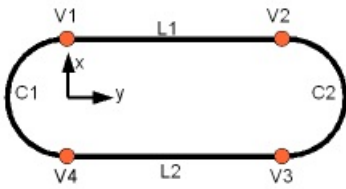
- **Geom** Base class of the geometric objects
- **Line** A straight line in 3D, defined by starting point and end point
- **Circle** Circle or circle segment defined by a center point and start and end point
- And soon some more ;-)

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



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.

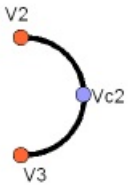
Creating Geometry

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:

```
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)
```

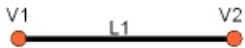
Arc



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

```
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)
```

Line



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

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

Putting all together

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

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

Make a prism

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

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

Show it all

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

Creating basic shapes

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

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

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](#) 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:

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

Creating a Vector

Vectors 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:

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

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:

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

Creating an Edge

An edge is nothing but a line with two vertexes:

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

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

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

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

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

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:

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

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:

```
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 0197AED8>, <Edge object at 01828B20>, <Edge object at 0190A788>]
Part.show(wire3)
```

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

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

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)

```
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
```

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

Creating a Circle

A circle can be created as simply as this:

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

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

```
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))
```

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:

```
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)
```

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):

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

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.

```
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()
```

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:

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

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:

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

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:

```
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)
```

`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:

```
Part.Ellipse()
```

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

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

Create a copy of the given ellipse

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

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.

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

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)

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

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

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

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:

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

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

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

The above code will create a slice of the torus

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

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)`

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

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.

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

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

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

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

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

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:

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

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:

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

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:

```
myMat = Base.Matrix()
myMat.move(Base.Vector(2,0,0))
myMat.rotateZ(math.pi/2)
```

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:

```
myMat.move(2,0,0)
myMat.move(Base.Vector(2,0,0))
```

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:

```
myShape.transformShape(myMat)
```

or

```
myShape.transformGeometry(myMat)
```

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():

```
myMat = Base.Matrix()
myMat.scale(2,1,1)
myShape.transformGeometry(myMat)
```

Boolean Operations

Subtraction

Subtracting a shape from another one is called "cut" in OCC/FreeCAD jargon and is done like this:

```
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)
```

Intersection

The same way, the intersection between 2 shapes is called "common" and is done this way:

```
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)
```

Union

Union is called "fuse" and works the same way:

```
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)
```

Section

A Section is the intersection between a solid shape and a plane shape. It will return an intersection curve, a compound with edges

```
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>]
```

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":

```
circle = Part.makeCircle(10)
tube = circle.extrude(Base.Vector(0,0,2))
```

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:

```
wire = Part.Wire(circle)
disc = Part.makeFace(wire)
cylinder = disc.extrude(Base.Vector(0,0,2))
```

Exploring shapes

You can easily explore the topological data structure:

```
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
```

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:

```
import Part
box = Part.makeBox(100,100,100)
anEdge = box.Edges[0]
print anEdge.Length
```

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.

```
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)
```

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

```
import Part
Part.show(Part.makeBox(100,100,100))
Gui.SendMsgToActiveView("ViewFit")
```

Select now some faces or edges. With this script you can iterate all selected objects and their sub elements:

```
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
```

Select some edges and this script will calculate the length:

```
length = 0.0
for o in Gui.Selection.getSelectionEx():
    for s in o.SubObjects:
        length += s.Length
print "Length of the selected edges:" ,length
```

Complete example: The OCC bottle

A typical example found on the [OpenCasCade Getting Started Page](#) 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:

```
import Part
import MakeBottle
bottle = MakeBottle.makeBottle()
Part.show(bottle)
```

The complete script

Here is the complete MakeBottle script:

```
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

```

Detailed explanation

```

import Part, FreeCAD, math
from FreeCAD import Base

```

We will need, of course, the Part module, but also the FreeCAD.Base module, which contains basic FreeCAD structures like vectors and matrixes.

```

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)

```

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.

```

aArcOfCircle = Part.Arc(aPnt2,aPnt3,aPnt4)
aSegment1=Part.Line(aPnt1,aPnt2)
aSegment2=Part.Line(aPnt4,aPnt5)

```

Here we actually define the geometry: an arc, made of 3 points, and two line segments, made of 2 points.

```

aEdge1=aSegment1.toShape()
aEdge2=aArcOfCircle.toShape()
aEdge3=aSegment2.toShape()
aWire=Part.Wire([aEdge1,aEdge2,aEdge3])

```

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.

```

aTrsf=Base.Matrix()
aTrsf.rotateZ(math.pi) # rotate around the z-axis
aMirroredWire=aWire.transformGeometry(aTrsf)
myWireProfile=Part.Wire([aWire,aMirroredWire])

```

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.

```

myFaceProfile=Part.Face(myWireProfile)
aPrismVec=Base.Vector(0,0,myHeight)
myBody=myFaceProfile.extrude(aPrismVec)
myBody=myBody.makeFillet(myThickness/12.0,myBody.Edges)

```

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?

```
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)
```

Then, the body of our bottle is made, we still need to create a neck. So we make a new solid, with a cylinder.

```
myBody = myBody.fuse(myNeck)
```

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.

```
return myBody
```

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:

```
myObject = FreeCAD.ActiveDocument.addObject("Part::Feature", "myObject")
myObject.Shape = bottle
```

or, more simple:

```
Part.show(bottle)
```

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, IGES, 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:

```
import Part
s = Part.makeBox(0,0,0,10,10,10)
s.exportStep("test.stp")
```

this will save our box into a STEP file. To load a BREP, IGES or STEP file, simply do the contrary:

```
import Part
s = Part.Shape()
s.read("test.stp")
```

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

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Topological_data_scripting"

Mesh to Part

Converting Part objects to Meshes

Converting higher-level objects such as [Part shapes](#) into simpler objects such as [meshes](#) 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
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)
```

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:

```
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
```

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](#) 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:

```
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)
```

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
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)
```

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Mesh_to_Part"

Scenegraph

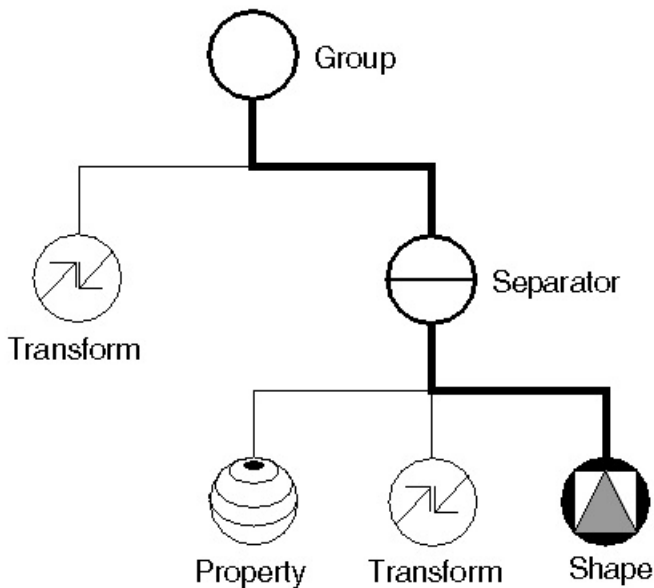
FreeCAD is basically a collage of different powerful libraries, the most important being [openCascade](#), for managing and constructing geometry, [Coin3d](#) to display that geometry, and [Qt](#) to put all this in a nice Graphical User Interface.

The geometry that appears in the 3D views of FreeCAD are rendered by the Coin3D library. Coin3D is an implementation of the [OpenInventor](#) 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.

[OpenInventor](#) 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](#), like the one below:



— Path

image from [Inventor mentor](#)

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:

```
#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 {
    }
  }
}
```

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](#).

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:

```
obj = FreeCAD.ActiveDocument.ActiveObject
viewprovider = obj.ViewObject
print viewprovider.toString()
```

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](#).

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Scenegraph>"

Pivy is a python binding library for **Coin3d**, the 3D-rendering library used FreeCAD. When imported in a running python interpreter, it allows to dialog directly with any running Coin3d **scenegraphs**, 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:

```
from pivy import coin
```

Accessing and modifying the scenegraph

We saw in the **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:

```
sg = FreeCADGui.ActiveDocument.ActiveView.getSceneGraph()
print sg
```

This will return the root node:

```
<pivy.coin.SoSelection; proxy of <Swig Object of type 'SoSelection *' at 0x360cb60> >
```

We can inspect the immediate children of our scene:

```
for node in sg.getChildren():
    print node
```

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**.

Let's try to add something to our scenegraph now. We'll add a nice red cube:

```
col = coin.SoBaseColor()
col.rgb=(1,0,0)
cub = coin.SoCube()
myCustomNode = coin.SoSeparator()
myCustomNode.addChild(col)
myCustomNode.addChild(cub)
sg.addChild(myCustomNode)
```

and here is our (nice) red cube. Now, let's try this:

```
col.rgb=(1,1,0)
```

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:

```
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)
```

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:

```
trans.translation.setValue([2,0,0])
```

And our cube would jump 2 units to the right. Finally, removing something is done with:

```
sg.removeChild(myCustomNode)
```

Using callback mechanisms

A **callback mechanism** 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:

```
class ButtonTest:
    def __init__(self):
        self.view = FreeCADGui.ActiveDocument.ActiveView
        self.callback = self.view.addEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(), self.getMouseClick)
    def getMouseClick(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()
```

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](#) of possible events and their parameters, or the [official coin documentation](#).

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()`)

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Pivy>"

PyQt

PyQt is a python module that allows python applications to create, access and modify **Qt** applications. You can use it for example to create your own Qt programs in python, or to access and modify the interface of a running qt application, like FreeCAD.

By using the PyQt module from inside FreeCAD, you have therefore full control over its interface. You can for example:

- Add your own panels, widgets and toolbars
- Add or hide elements to existing panels
- Change, redirect or add connections between all those elements

PyQt has an extensive [API documentation](#), and there are many tutorials on the net to teach you how it works.

If you want to work on the FreeCAD interface, the very first thing to do is create a reference to the FreeCAD main window:

```
import sys
from PyQt4 import QtGui
app = QtGui.QApp
mw = app.activeWindow()
```

Then, you can for example browse through all the widgets of the interface:

```
for child in mw.children():
    print 'widget name = ', child.objectName(), ', widget type = ', child
```

The widgets in a Qt interface are usually nested into "containers" widgets, so the children of our main window can themselves contain other children. Depending on the widget type, there are a lot of things you can do. Check the API documentation to see what is possible.

Adding a new widget, for example a `QDockWidget` (which can be placed in one of FreeCAD's side panels) is easy:

```
myWidget = QtGui.QDockWidget()
mw.addDockWidget(QtCore.Qt.RightDockWidgetArea, myWidget)
```

You could then add stuff directly to your widget:

```
myWidget.setObjectName("my Nice New Widget")
myWidget.resize(QtCore.QSize(300,100)) # sets size of the widget
label = QtGui.QLabel("Hello World", myWidget) # creates a label
label.setGeometry(QtCore.QRect(50,50,200,24)) # sets its size
label.setObjectName("myLabel") # sets its name, so it can be found by name
```

But a preferred method is to create a UI object which will do all of the setup of your widget at once. The big advantage is that such a UI object can be **created graphically** with the Qt Designer program. A typical object generated by Qt Designer is like this:

```
class myWidget_Ui(object):
    def setupUi(self, myWidget):
        myWidget.setObjectName("my Nice New Widget")
        myWidget.resize(QtCore.QSize(300,100).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))
```

To use it, you just need to apply it to your freshly created widget like this:

```
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
```

Online version: <http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=PyQt>

Scripted objects

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, can be saved in a document and opened on any other installation of FreeCAD, since the python code that defines the object is also saved in the document.

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.

Basic example

The following sample can be found in the [src/Mod/TemplatePyMod/FeaturePython.py](#) 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(l,w,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 optional"
        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])
```


This example makes use of the [Part Module](#) to create an octahedron, then creates its coin representation with pivy.

First is the Document object itself:

```
import FreeCAD, FreeCADGui, Part

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.Vertexes:
                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)
```



```

selectionNode = coin.SoType.fromName("SoFCSelection").createInstance()
selectionNode.documentName.setValue(FreeCAD.ActiveDocument.Name)
selectionNode.objectName.setValue(obj.Object.Name) # here obj is the ViewObject, we need its associated App Object
selectionNode.subElementName.setValue("Face")
selectNode.addChild(selectionNode)
...
self.shaded.addChild(selectionNode)
self.wireframe.addChild(selectionNode)

```

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:

```

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()

```

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Scripted_objects"

Embedding FreeCAD

FreeCAD has the amazing ability to be importable as a python module in other programs or in a standalone python console, together with all its modules and components. It's even possible to import the FreeCAD GUI as python module -- with some restrictions, however.

Using FreeCAD without GUI

One first, direct, easy and useful application you can make of this is to import FreeCAD documents into your program. In the following example, we'll import the Part geometry of a FreeCAD document into **blender**. Here is the complete script. I hope you'll be impressed by its simplicity:

```
FREECADPATH = '/opt/FreeCAD/lib' # path to your FreeCAD.so or FreeCAD.dll file
import Blender, sys
sys.path.append(FREECADPATH)

def import_fcstd(filename):
    try:
        import FreeCAD
    except ValueError:
        Blender.Draw.PupMenu('Error! FreeCAD library not found. Please check the FREECADPATH variable in the import script is correct')
    else:
        scene = Blender.Scene.GetCurrent()
        import Part
        doc = FreeCAD.open(filename)
        objects = doc.Objects
        for ob in objects:
            if ob.Type[:4] == 'Part':
                shape = ob.Shape
                if shape.Faces:
                    mesh = Blender.Mesh.New()
                    rawdata = shape.tessellate(1)
                    for v in rawdata[0]:
                        mesh.verts.append((v.x, v.y, v.z))
                    for f in rawdata[1]:
                        mesh.faces.append.append(f)
                    scene.objects.new(mesh, ob.Name)
        Blender.Redraw()

def main():
    Blender.Window.FileSelector(import_fcstd, 'IMPORT FCSTD',
                               Blender.sys.makeName(ext='.fcstd'))

# This lets you import the script without running it
if __name__ == '__main__':
    main()
```

The first, important part is to make sure python will find our FreeCAD library. Once it finds it, all FreeCAD modules such as Part, that we'll use too, will be available automatically. So we simply take the sys.path variable, which is where python searches for modules, and we append the FreeCAD lib path. This modification is only temporary, and will be lost when we'll close our python interpreter. Another way could be making a link to your FreeCAD library in one of the python search paths. I kept the path in a constant (FREECADPATH) so it'll be easier for another user of the script to configure it to his own system.

Once we are sure the library is loaded (the try/except sequence), we can now work with FreeCAD, the same way as we would inside FreeCAD's own python interpreter. We open the FreeCAD document that is passed to us by 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](#)), 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

```
import FreeCADGui
FreeCADGui.showMainWindow()
```

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.

Online version: http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Embedding_FreeCAD

Category:API

(Redirected from [API documentation](#))

This category gathers article that list objects and methods available to the python programmer.

Pages in category "API"

The following 15 pages are in this category, out of 15 total.

B

- [Base API](#)
- [Builtin modules](#)

C

- [Console API](#)

D

- [Draft API](#)

F

- [FreeCAD API](#)

F cont.

- [FreeCADGui API](#)

M

- [Matrix API](#)
- [Mesh API](#)

O

- [Object API](#)

P

- [Part API](#)
- [Placement API](#)

S

- [Selection API](#)

T

- [TopoShape API](#)

V

- [Vector API](#)
- [ViewObject API](#)

Code snippets

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

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 Scripts.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

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.forName("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
```

Online version: http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Code_snippets

Line drawing function

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:

```
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)
```

Detailed explanation

```
import Part, FreeCADGui
from pivy.coin import *
```

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](#), 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...

```
class line:
```

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.

```
"this class will create a line after the user clicked 2 points on the screen"
```

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.

```
def __init__(self):
```

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.

```
self.view = FreeCADGui.ActiveDocument.ActiveView
```

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.

```
self.stack = []
```

Here we create an empty list that will contain the 3D points sent by the `getpoint` function.

```
self.callback = self.view.addEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.getpoint)
```

This is the important part: Since it is actually a [coin3D](#) 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](#) 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!

```
def getpoint(self,event_cb):
```

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](#)).

```
if event.getState() == SoMouseButtonEvent.DOWN:
```

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.

```
pos = event.getPosition()
```

Here we get the screen coordinates of the mouse cursor

```
point = self.view.getPoint(pos[0],pos[1])
```

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.

```
self.stack.append(point)
```

We add our new point to the stack

```
if len(self.stack) == 2:
```

Do we have enough points already? if yes, then let's draw the line!

```
l = Part.Line(self.stack[0],self.stack[1])
```

Here we use the function Line() from the [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.

```
shape = l.toShape()
```

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.

```
Part.show(shape)
```

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.

```
self.view.removeEventCallbackPivy(SoMouseButtonEvent.getClassTypeId(),self.callback)
```

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](#), 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:

```
import exercise
```

If no error message appear, that means our exercise script has been loaded. We can now check its contents with:

```
dir(exercise)
```

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:

```
exercise.line()
```

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:

```
class MyWorkbench (Workbench):
    MenuText = "MyScripts"
    def Initialize(self):
        import exercise
        commandslist = ["line"]
        self.appendToolbar("My Scripts",commandslist)
Gui.addWorkbench(MyWorkbench())
```

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:

```
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': 'Creates a line by clicking 2 points on the screen'}
FreeCADGui.addCommand('line', line())
```

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. Some 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 [talk page](#)!

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Line_drawing_function"

Dialog creation

In this page we will show how to build a simple Qt Dialog with [Qt Designer](#), 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.

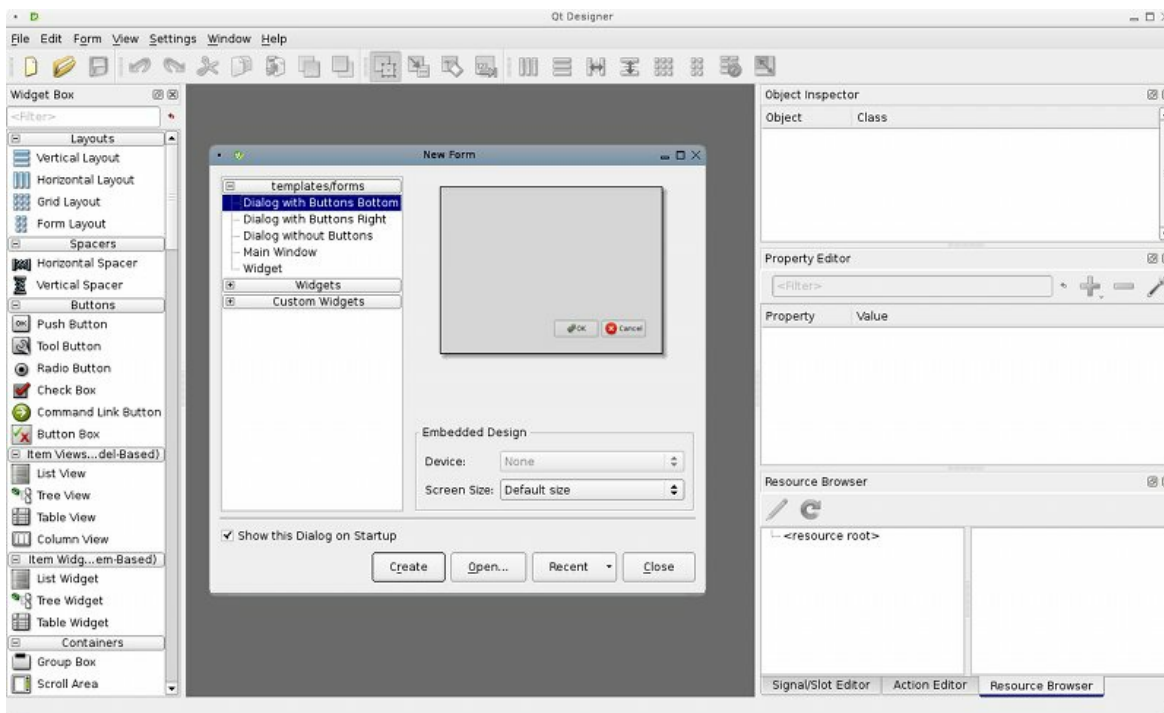
There are a couple of concepts to know when designing interface:

- **Modal/non-modal dialogs:** 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:



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:

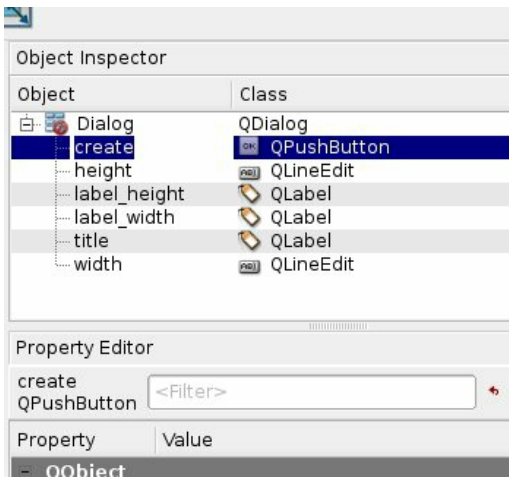


It is very simple to use. On the left bar you have elements that can be dragged on your widget. On the right side you have properties panels displaying all kinds of editable properties of selected elements. So, begin with creating a new widget. Select "Dialog without buttons", since we don't want the 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:



Converting our dialog to python

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:

```
pyuic mywidget.ui > mywidget.py
```

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:

```
from PyQt4 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-O-Matic", None, QtGui.QApplication.UnicodeUTF8))
        ...
```

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:

```
from PyQt4 import QtGui
import mywidget
d = QtGui.QWidget()
d.ui = mywidget.Ui_Dialog()
d.ui.setupUi(d)
d.show()
```

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:

```
d.hide()
```

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](#) lists all the qt widgets, what they can do, what signals they can send, etc...

What we will do here, is create a new function that will create a plane based on height and width, and 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:

```
import FreeCAD, Part
```

Then, let's add a new function to our Ui_Dialog class:


```

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()

```

Then, we need to inform Qt to connect the button to the function, by placing the following line just before `QtCore.QMetaObject.connectSlotsByName(Dialog)`:

```

QtCore.QObject.connect(self.create,QtCore.SIGNAL("pressed()"),self.createPlane)

```

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:

```

class plane():
    d = QtGui.QWidget()
    d.ui = Ui_Dialog()
    d.ui.setupUi(d)
    d.show()

```

Then, from FreeCAD, we only need to do:

```

import mywidget
mywidget.plane()

```

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](#) 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:

```

# -*- 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
#
# WARNING! All changes made in this file will be lost!

from PyQt4 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-O-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():
    d = QtGui.QWidget()
    d.ui = Ui_Dialog()
    d.ui.setupUi(d)
    d.show()
```

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Dialog_creation"

Licence

Statement of the maintainer

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** and the **GPL** 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.

Used Licences

Here the three licences under which FreeCAD is published:

General Public Licence (GPL2+)

For the Python scripts to build the binaries as stated in the .py files in src/Tools

Lesser General Public Licence (LGPL2+)

For the core libs as stated in the .h and .cpp files in src/App src/Gui src/Base and most **modules** 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

For the documentation on <http://free-cad.sourceforge.net/> as not marked differently by the author

We try to use only LGPL type licences for the core system linked libraries (see **Third Party Libraries**) with one exception:

- the Coin3D licence (www.coin3d.org).

See FreeCAD's **debian copyright file** for more details about the licenses used 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...

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.

Professional developers

Professional 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. If you want to sell your module you need a Coin3D licence, otherwise you are forced by this library to make it open source.

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Licence>"

Tracker

Reporting bugs

Recently the sourceforge platform made the [mantis bug tracker](#) application available to projects, and FreeCAD is now using it actively, instead of the old built-in bug tracker. The adress of our bug tracker is:

http://sourceforge.net/apps/mantisbt/free-cad/main_page.php

If you think you might have found a bug, you are welcome to report it there. 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, dont hesitate to explain your problem on the [forum](#) and ask what to do.
- Before submitting anything, read the [frequently asked questions](#), do a search on the [forum](#), 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.
- Join the following information: Your operating system, if it is 32 or 64 bits, and the version of FreeCAD you are running.
- 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 Subversion diff tool and submit it on the same tracker (file it as patch).

The "old" sourceforge tracker (obsolete)

Note: Please use the new mantis bug tracker to submit bugs, this one is now a bit deprecated...

Where to find?

The FreeCAD project has its own [tracker summary page](#). There you find the overview on the individual sections of the tracker.

When to use?



The FreeCAD Bug Tracker

Bugs

If you think you might have found a bug, go to the [Bugs Section](#) of the tracker and choose "any" for *status* to see all bug request ever filed. The keyword search allows you to find bug tracker entries for a similiar issue. If you can not find an older entry about your problem, you should file a new entry on the same page.

Feature Requests

If you are missing a feature in FreeCAD that you think of as beeing absolutely necessary to become the *worlds best* CAD-Software, you might find the [Feature Request](#) section helpfull.

Support Requests

If you don't get around compiling FreeCAD and the [Compile On Windows](#) or [Compile On Unix](#) section does not give you a hint, or you try to port it to a new environment or are programming new modules or extensions for FreeCAD and need some assistance then the [Support Requests](#) section is the place you might want to go to.

New 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 Subversion and file it in the [patches section](#).

CompileOnWindows

This article explains step by step **how to compile FreeCAD on Windows**.

Prerequisites

What you need is mainly the compiler. On Windows we use the MS VisualStudio 8 Compiler with the highest service pack. Although it's probably possible to use Cygwin or MingW gcc it's not tested or ported so far. We have also ported to use VC8 Express Edition. You need to download the Windows Platform SDK to get e.g. the Windows.h. Also you need all the [Third Party Libraries](#) to successfully compile FreeCAD.

If you use the MS compilers you want most likely download the FreeCAD LibPack which provides you with all needed libs to build FreeCAD on Windows.

Building with cMake

First of all, you have to [download cMake](#) and install it on your build machine.

The switch to cMake

Since version 0.9 we use the cMake build system to generate the build/make files for various compilers. We do not longer deliver .vcproj files. If you want build former versions of FreeCAD (0.8 and older) see "Building older versions" later in this article.

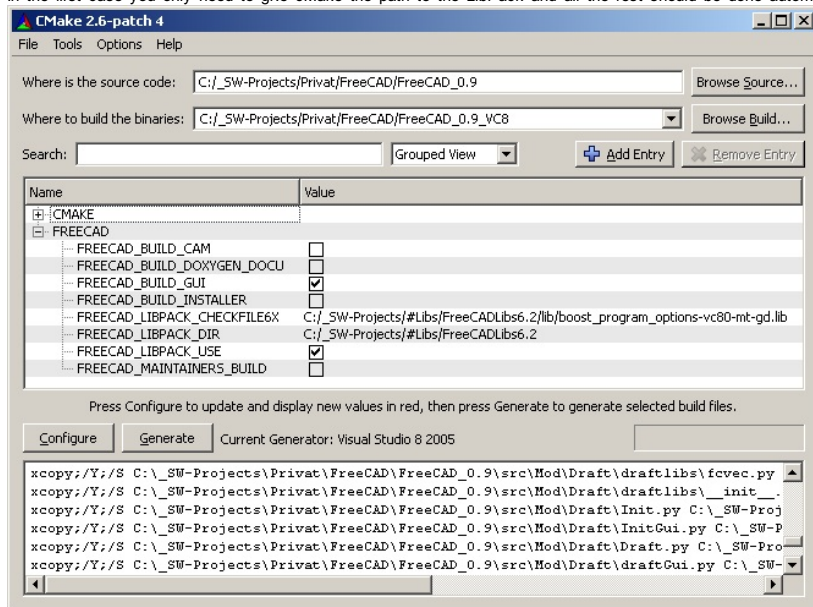
We switched because it became more and more painful to maintain project files for 30+ build targets and x compilers. cMake gives us the possibility to support alternative IDEs, like Code::Blocks, Qt Creator and Eclipse CDT the main compiler we use is still MS VC9 Express, though. But we plan for the future a build process on Windows without proprietary compiler software.

Configure the build process

The first step to build FreeCAD with cMake is to configure the environment. There are basically two ways to go:

- Using the LibPack
- Installing all needed libs and let cMake find them

In the first case you only need to give cMake the path to the LibPack and all the rest should be done automatically and you see such a screen:



You see the LibPack path inserted into the **FREECAD_LIBPACK_DIR** variable. Starting from that all includes and paths are set. You just need to press the **Generate** button and the project files get generated.

If you switch the **FREECAD_LIBPACK_USE** options off, the configuration tries to find each and every library needed on your system. Depending on the libs that works well more or less. So you have to do often define some paths by hand. cMake will show you what is not found and need to be specified.

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 switches to determine the build process.

Here is the description 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

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 programm
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 Paths
set LIB=%LIB%;C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib
set LIB=%LIB%;%PROGRAMFILES%\Microsoft Visual Studio\VC98\Lib

rem Start the Visual 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](#). You can find it on the [download page](#) 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](#) 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](#) 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.

Additional stuff

If you want to build the source code documentation you need [DoxyGen](#).

To create an installer package you need [WIX](#).

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//  
.....
```

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=CompileOnWindows>"

CompileOnUnix

On recent linux distributions, FreeCAD is generally easy to build, since all dependencies are usually provided by the package manager. Basically, you'll just need to get the FreeCAD source, then install the dependencies listed below, then issue:

```
./autogen.sh && ./configure && make
```

or

```
./cmake . && make
```

to get FreeCAD built, depending on what build system you prefer to use ([autotools](#) or [cmake](#)). Below, you'll find detailed explanations of the whole process and particularities you might encounter. There are automatic scripts located at the end of this section for [Popular Distributions](#). There is also a general [shellscript](#) that you can use to follow along for a general Debian-based Distribution. If you find anything wrong in it or here below, please help us correcting it.

Getting the source

Before you can compile FreeCAD, you must get the source code. First install [subversion](#). Then, from the directory of your choice (for example your user directory), do:

```
svn co https://free-cad.svn.sourceforge.net/svnroot/free-cad/trunk freecad
```

This will perform an anonymous checkout of the current development version in a new directory called "freecad". Alternatively you can download a [source tarball](#) but they could be already quite old so it's probably better to always get the latest sources via subversion. Note, though, that the subversion version is the FreeCAD version currently being worked on, and it might contain bugs or even fail to compile.

Prerequisites

To compile FreeCAD under Linux you have to install all libraries mentioned in [Third Party Libraries](#) first. You also need the [GNU gcc compiler](#) version equal or above 3.0.0. g++ is also needed because FreeCAD is completely written in C++. Both gcc and g++ are included in the build-essential package listed below. 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 would be 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.

Debian/Ubuntu and most recent distributions

The Eigen3 library is now required by the Sketcher module. This library is only available starting from Ubuntu 11.10 repositories. For prior Ubuntu releases, you can either download it [from here](#) and install it manually, or add the [FreeCAD Daily Builds PPA](#) to your software sources before installing it through one of the means listed below.

On Debian based systems it is quite easy to get all needed dependencies installed. Most of the libraries are available via apt-get or synaptic package manager. Below are listed all packages you need to install. On other distributions, the package names can vary, but usually you'll be able to find them all too:

```
build-essential
python
libtool
libcoin60-dev
libsoqt4-dev
libxerces-c2-dev (or libxerces28-dev depending on your system)
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
libboost-thread-dev
libqt4-dev
qt4-dev-tools
python2.5-dev (or higher version if available)
libopencascade-dev
libf2c2-dev
gfortran
libeigen3-dev
libqtwebkit-dev
```

To simply install all these libraries in one step, just copy/paste the following text in a terminal (only for debian/ubuntu based systems) as root:

On Ubuntu-based systems, substitute 'aptitude' for 'sudo apt-get install' as it's the preferred way.

```
aptitude install build-essential python libtool 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 libboost-thread-dev
libqt4-dev qt4-dev-tools python2.5-dev libopencascade-dev libf2c2-dev gfortran libeigen3-dev libqtwebkit-dev
```

Another simple way, if your debian-based distribution already has a freecad package, is to do:


```
apt-get build-dep freecad
```

which will fetch all packages needed to build freecad. Beware that some new packages might be needed since last version, that you still need to install manually.

optionally you can also install

```
libsimage-dev (to make Coin to support additional image file formats)
checkinstall (to register your installed files into your system's package manager, so yo 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

To build & install FreeCAD on Fedora 13, a few tips and tricks are needed:

- Install a bunch of required packages, most are available from the Fedora 13 repositories
- Download and build xerces
- Download and build OpenCASCADE. Need to point it to xmu: `./configure --with-xmu-include=/usr/include/X11/Xmu --with-xmu-library=/usr/lib`
- Download and build Pivy. You have to remove 2 references to non existent "SoQtSpaceball.h" from `pivy/interfaces/soqt.i` Commenting out those two lines allow the build & install to work.
- Configure Freecad. You will need to point it to a few things: `./configure --with-qt4-include=/usr/include --with-qt4-bin=/usr/lib/qt4/bin --with-occ-lib=/usr/local/lib --with-occ-include=/usr/local/inc --with-xercesc-lib=/usr/local/lib`
- make - hits a problem where the build is breaking because the `ldflags` for `soqt` are set to `"-LNONE"` which made `libtool` barf. My hackish workaround was to modify `/usr/lib/Coin2/conf/soqt-default.cfg` so that the `ldflags` are `""` instead of `"-LNONE"`. After this -> success !
- make install

Older and non-conventional distributions

On older distributions, however you might not find the following libraries:

OpenCASCADE

Not all Linux distributions have an official OpenCASCADE package in their repositories. You have to check yourself for your distribution if one is available. 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](#). 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 tcl8.4-dev tk8.4-dev libglib-mesa-dev
libglul-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](#):

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 = -l/usr/include and CSF_TclTkLibs_INCLUDES = -l/usr/include and add /tcl8.4 and /tk8.4 to it so they read: CSF_TclLibs_INCLUDES = -l/usr/include/tcl8.4 and CSF_TclTkLibs_INCLUDES = -l/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. At the time of writing, Pivy is very new and might not have made its way into your distribution repository. If you cannot find Pivy in your distribution's packages repository, you can grab debian/ubuntu packages on the FreeCAD download page:

<http://sourceforge.net/projects/free-cad/files/FreeCAD%20Linux/>

or compile pivy yourself:

[Pivy compilation instructions](#)

Compile FreeCAD



SourceForge bug tracker:
Finding compile bugs

The autotools way

You must have automake and libtool installed on your system; on Debian/Ubuntu:

```
aptitude install automake libtool
```

If you got the sources with subversion then the very first step must be

```
./autogen.sh
```

that creates the configure script and more. For the build process itself we provide a configure script. Just type

```
./configure
```

To get everything configured. If you want an overview of all options you can specify, you can type

```
./configure --help.
```

Normally you need none of them - unless you have one of your libraries installed in a really uncommon directory. After configuration has finished, compiling FreeCAD is as simple as

```
make
```

If any error occurs while building from sources, please double-check this page and README.Linux file, then you could jump to the [Bug Tracker](#) on SourceForge, choose *Any* for status and click the *Browse* button to see previous reports on compile problems. After having built FreeCAD successfully, do

```
make install
```

to install it onto your machine. The default install directory is

```
~/FreeCAD
```

It will be installed in a FreeCAD folder in your home folder, so you don't need root privileges. Instead of `make install`, you can also do

```
checkinstall
```

In this way FreeCAD will be installed by your package management system, so you can uninstall it easily later. But since all of FreeCAD installation resides into one single directory, just removing the FreeCAD directory is a valid way to uninstall too.

The cMake way

cMake is a newer build system which has the big advantage of being common for different target systems (Linux, Windows, MacOSX, etc). FreeCAD is progressively switching to the cMake system, and you can already build FreeCAD in that way. Like with autotools, the process goes in two steps: first you configure the source, and then you build the program proper.

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 and, assuming cMake is installed on your system, for a first compilation you can just switch to your FreeCAD source folder and issue:

```
cmake .
```

to have the source configured (Don't forget the dot! It's a parameter to the `cmake` command). Then issue:

```
make
```

to have FreeCAD built. A proper system-wide installation of FreeCAD still cannot be made with `cmake`, but you can run FreeCAD simply by issuing

```
./bin/FreeCAD
```

- Out of source build:

If you intend to follow the fast evolving SVN versions and maybe contribute to the code, cMake allows for out of source builds. Just create a build directory distinct from your `freecad` root folder, switch to this directory and there issue:

```
cmake path-to-freecad-root  
make
```

The effect is that every file generated at configure time goes in your build directory, and does not get mixed with the sources checked-out from SVN. The management of your local copy of the sources is greatly eased.

- 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`).

As an example, to configure on the command line with the Assembly module built, 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
```

- If you feel bold enough to dive in the code, you could take advantage to build and consult Doxygen generated FreeCAD's [Source documentation](#)

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.

Automake macros

The configure script of FreeCAD makes use of several automake macros that are sometimes not installed with their packages: `bnv_have_qt.m4`, `coin.m4`, and `gts.m4`. If needed (error while configuring), google for them and you will find them easily. They are just simple scripts that you need to put in your `/usr/share/aclocal` folder.

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
lintian (optional, used for checking if packages are standard-compliant)
```

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

```
lintian your-fresh-new-freecad-package.deb (replace by the name of the package you just created)
```

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. This scripts should run on 32 and 64 bit versions. They are written for distinct version, but are also likely to run on a later version with or without minor changes.

If you have such a script for your preferred distro, please send it! We will incorporate it into this article.

Note that this script starts by adding the [FreeCAD Daily Builds PPA](#) repository so it can proceed with the Eigen3 library (`libeigen3-dev`) installation. If you already have this library installed on your system, you can remove the first line.

Ubuntu 10.04 LTS - Lucid Lynx / Ubuntu 10.10 Maverick Meerkat / Ubuntu 11.04 Natty Narwhal

```
sudo add-apt-repository ppa:freecad-maintainers/freecad-daily && sudo apt-get update
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 libboost-thread-dev \
libqt4-dev qt4-dev-tools python2.6-dev libopencascade-dev libsoqt4-dev \
libode-dev subversion cmake libeigen2-dev libsimage-dev python-qt4 \
libtool autotools-dev automake bison flex libf2c2-dev gfortran libeigen3-dev libqtwebkit-dev

# checkout the latest source
svn co https://free-cad.svn.sourceforge.net/svnroot/free-cad/trunk freecad

# go to source dir
cd freecad

# build configuration
cmake .

# build FreeCAD
make

# test FreeCAD
cd bin
./FreeCAD -t 0

# use FreeCAD
./FreeCAD
```

OpenSuse 11.2

This script is **not working** at the moment because:

- `libXerces-c-devel` seems to be disappeared

```
# install needed packages for development
sudo zypper install gcc cmake subversion OpenCASCADE-devel \
libXerces-c-devel python-devel libqt4-devel python-qt4 \
Coin-devel SoQt-devel boost-devel libode-devel libQtWebKit-devel \
libeigen2-devel gcc-fortran f2c
```

```
# get the source
svn co https://free-cad.svn.sourceforge.net/svnroot/free-cad/trunk freecad

# go to source dir
cd freecad

# build configuration
cmake .

# build FreeCAD
nice make

# test FreeCAD
cd bin
./FreeCAD -t 0
```

OpenSuse 11.1

```
# additional repository (for OpenCascade)
sudo zypper -p http://packman.unixheads.com/suse/11.1/

# install needed packages for development
sudo zypper install gcc cmake subversion OpenCASCADE-devel \
libXerces-c-devel python-devel libqt4-devel python-qt4 \
Coin-devel SoQt-devel boost-devel libode-devel libQtWebKit-devel \
libeigen2-devel

# get the source
svn co https://free-cad.svn.sourceforge.net/svnroot/free-cad/trunk freecad

# go to source dir
cd freecad

# build configuration
cmake .

# build FreeCAD
nice 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 \
libsimage-dev libopencascade-dev \
libsoqt4-dev libode-dev subversion cmake libeigen2-dev python-pivy \
libtool autotools-dev automake libf2c2-dev gfortran

# checkout the latest source
svn co https://free-cad.svn.sourceforge.net/svnroot/free-cad/trunk freecad

# go to source dir
cd freecad

# build configuration
cmake .

# build FreeCAD
make

# test FreeCAD
cd bin
./FreeCAD -t 0
```

CompileOnMac

Compiling FreeCAD on a Mac isn't much different from the steps on Linux or other UNIX variants. The biggest challenge is really getting all of the dependencies installed. In the following sections, I have detailed the steps to get this application to compile on Leopard and Snow Leopard using an Intel Mac (PowerPC should be feasible, but requires recompiling some binary libraries that I haven't got to yet). I have been the only one to successfully build FreeCAD on a Mac recently, so please post on the [discussion forum](#) if these steps work for you, on the [help forum](#) if they don't, or edit this page if you find errors (currently being followed & updated by Shaneyfelt).

Download the FreeCAD sources

First you need to get a copy of the FreeCAD source tree. Just check out the latest revision from the Sourceforge subversion repository using this command in the terminal:

```
svn co http://free-cad.svn.sourceforge.net/svnroot/free-cad/trunk freecad
```

This will put the FreeCAD source and related files in your home folder (~/.freecad/). Location is not important if you'd rather put it somewhere else, you just need full access to the files.

Install MacPorts and Library Dependencies

Next, if you don't already have it, install MacPorts. MacPorts is a system that allows you to download, compile, and install many common open-source applications with a single command. Similar applications from the UNIX/Linux world are PKGSRC and APT. To install, just download the disk image from the MacPorts site and follow the directions:

<http://www.macports.org/install.php>

Whether or not you just installed MacPorts, you'll probably want to make sure it's up to date. Run:

```
sudo port selfupdate
```

Now that MacPorts is installed and up to date, you can start installing some of FreeCAD's required packages:

- ode
- xercesc
- boost
- gts
- opencv

The following command will compile/install all required libraries. If MacPorts produces errors, you may want to try installing them one at a time.

```
sudo port install ode xercesc boost gts opencv
```

(gts and opencv may no longer be necessary)

You also need a FORTRAN compiler. Apple's fork of gcc on OSX does not come with FORTRAN. One method I tried is to `sudo fink install gcc46` but this assumes you have fink (attempted by Shaneyfelt 2100.Nov.14)

```
sudo fink selfupdate
sudo fink install gcc46
```

This installs another gcc compiler collection with the name gcc-4 to avoid a name conflict with the apple one. Of course this assumes you also have fink installed already. Perhaps there's a simpler way?

Install Frameworks and OpenCASCADE

FreeCAD has other dependencies (see [CompileOnUnix](#)), but the rest are either included by default in OS X Leopard or can be installed using Installer packages. Download and install the following:

- Qt <http://qt.nokia.com/downloads>

Get the "Qt Libraries" version unless you plan to develop using Qt (it's much smaller). FreeCAD compiles on OS X Leopard with Qt 4.5. Installs in /Library/Frameworks and /usr/bin.

- Coin <http://www.coin3d.org/lib/coin/releases/>

Install Coin.pkg AND CoinTools.pkg. FreeCAD compiles on OS X Leopard with Coin 3.1.0. Installs in /Library/Frameworks.

This doesn't seem to do the trick. Later on there's errors with Coin in the .configure step of FreeCAD. (attempted by Shaneyfelt 2011.Nov.14)

- SoQt <http://dl.getdropbox.com/u/103808/FreeCAD/SoQt-1.4.1.dmg>

Install SoQt.pkg AND SoQtTools.pkg. For some reason, the SoQt framework is not provided as an official binary. For convenience, I'm providing the above compiled version. If you'd like to compile your own, download the latest source from <http://www.coin3d.org/lib/soqt/releases> and follow the directions in README.MACOSX. FreeCAD compiles on OS X Leopard with SoQt 1.4.1. Installs in /Library/Frameworks.

- OpenCASCADE http://dl.getdropbox.com/u/103808/FreeCAD/OpenCASCADE_i386_6.3.0_20091128.dmg

The above OCC 6.3.0 binary distribution is a modified version of the one provided by the maintainers of PythonOCC <http://www.pythonocc.org/>. You can use the version from PythonOCC, however changes to the .la files are needed in order for the FreeCAD build process to properly link to it and you'll need to download the OCC source separately. I have not yet successfully built OpenCASCADE myself, but would like to eventually -- this would be key to providing a PowerPC distribution (if that's even possible). If you get OpenCASCADE to build on OS X, let me know how. Installs in /usr/local/lib/OCC and /usr/local/include/OCC.

UPDATED 2009-11-28 with fixes for Snow Leopard. If you installed OCC prior to this date, it is recommended that you manually delete the old files and install the new package.

```
sudo rm -r /usr/local/lib/OCC
sudo rm -r /usr/local/include/OCC
```

Download and 'install' the FreeCAD.app template

The following archive contains an application bundle template for FreeCAD. This is not strictly necessary, but it makes working with FreeCAD more

convenient than the default installation configuration. Mine is in the /Applications folder, but you should be able to put it anywhere you want -- just remember that the bundle can't be moved after FreeCAD is compiled and installed (without some further modifications). Running make install using the configuration below will install into this bundle.

http://dl.getdropbox.com/u/103808/FreeCAD/FreeCAD_bundle_template_20091128.tar.gz
UPDATED 2009-11-28 with the new FreeCAD application icon

Compile

Now configure, compile, and install FreeCAD using the following commands from within the root FreeCAD folder. If you put your FreeCAD.app bundle somewhere other than /Applications (or aren't using the bundle), change the 'PREFIX' line accordingly.

```
./autogen.sh

PREFIX=/Applications/FreeCAD.app/Contents

./configure --with-xercesc-lib=/opt/local/lib --with-xercesc-include=/opt/local/include --with-boost-lib=/opt/local/lib \
--with-boost-include=/opt/local/include --with-qt4-bin=/usr/bin --with-qt4-framework=/Library/Frameworks \
--with-occ-lib=/usr/local/lib/OCC --with-occ-include=/usr/local/include/OCC --with-coin=/Library/Frameworks \
--with-soqt=/Library/Frameworks --prefix=$PREFIX --bindir=$PREFIX/MacOS --libdir=$PREFIX/Frameworks/FreeCAD \
--includedir=$PREFIX/Resources/include --datarootdir=$PREFIX/Resources/share

make LDFLAGS=-Wl,-headerpad_max_install_names

make install
```

Depending on your machine, the make step can take quite a while.

Run

If everything went properly, double-clicking the .app bundle should start FreeCAD. If you have any issues, post the details on the [help forum](#).

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=CompileOnMac>"

Third Party Libraries

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 have to be moved to the FreeCAD base package. The used libraries are:

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/
OpenCasCade	>= 5.2	http://www.opencascade.org
Qt	>= 4.1.x	http://www.qtsoftware.com
Coin3D	>= 2.x	http://www.coin3d.org
ODE	>= 0.10.x	http://www.ode.org
SoQt	>= 1.2	http://www.coin3d.org
Xerces-C++	>= 2.7.x < 3.0	http://xml.apache.org/xerces-c/
GTS	>= 0.7.x	http://gts.sourceforge.net/
Zlib	>= 1.x.x	http://www.zlib.net/
Boost	>= 1.33.x	http://www.boost.org/
Eigen3	>= 3.0.1	http://eigen.tuxfamily.org/index.php?title=Main_Page

Details

Python

Version: 2.5 or higher

License: Python 2.5 license

You can use the source or binary from <http://www.python.org/> or use alternativly ActiveState Python from <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>.

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>.

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!

ODE (Open dynamic engine)

Version: 0.10.0 or higher

License: LGPL v2.1 or later or BSD

ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.

Credits

Russell Smith is the primary author of ODE.

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.

GTS

Version: 0.7.x

License: LGPL v2.0 or later

GTS stands for the GNU Triangulated Surface Library. It is an Open Source Free Software Library intended to provide a set of useful functions to deal with 3D surfaces meshed with interconnected triangles. The source code is available free of charge under the Free Software LGPL license.

Actually not needed to compile FreeCAD. You can switch on the usage with a preprocessor switch in FCConfig.h.

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/> for details.

LibPack

LibPack is a convenient package with all the above libraries packed together. It is currently available for the Windows platform on the [Download](#) page! If you're working under Linux you don't need a LibPack, instead 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

Third Party Tools

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](#) of FreeCAD, but can also be downloaded at www.trolltech.com.

InkScape

Great vector drawing programm. Adheres to the SVG standard and is used to draw Icons and Pictures. Get it at 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

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](#)

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 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 Conflicts, 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 tortoisesvn.tigris.org.

StarUML

A full featured Open Source UML programm. It has a lot of features of the big ones, including reverse engeniering C++ source code....

Download here: staruml.sourceforge.net

Tools on Linux

TODO

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Third_Party_Tools"

Start up and Configuration

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 some 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:

```
FreeCAD --help
```

From the response you can read the possible parameters:

```
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

Configuration:
-l [ --write-log ] arg write a log file
-t [ --run-test ] arg test level
-M [ --module-path ] arg additional module paths
-P [ --python-path ] arg additional python paths
--response-file arg   can be specified with '@name', too
```

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:

```
FreeCAD @ResponseFile.txt
```

or:

```
FreeCAD --response-file=ResponseFile.txt
```

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:

```
FreeCAD -c
```

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:

```
import sys sys.path.append("path/to/FreeCAD/lib") # change this by your own FreeCAD lib path import FreeCAD
```

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 Posix (Linux)
UserAppData	Path where FreeCAD stores User Related application data.	C:\Documents and Settings\username\Application Data\FreeCAD	/home/username/.FreeCAD
UserParameter	File where FreeCAD stores User Related application data.	C:\Documents and Settings\username\Application Data\FreeCAD\user.cfg	/home/username/.FreeCAD/user.cfg
SystemParameter	File where FreeCAD stores Application Related data.	C:\Documents and Settings\username\Application Data\FreeCAD\system.cfg	/home/username/.FreeCAD/system.cfg
UserHomePath	Home path of the current user	C:\Documents and Settings\username\My Documents	/home/username

Command line arguments

User config entries

Config var name	Synopsis	Example
LoggingFile	1 if the logging is switched on	1
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:

- PYTHONPATH
- PYTHONHOME
- TCL_LIBRARY
- TCLLIBPATH

OpenCascade related environment variables:

- CSF_MDTVFontDirectory
- CSF_MDTVTexturesDirectory
- CSF_UnitsDefinition
- CSF_UnitsLexicon
- CSF_StandardDefaults
- CSF_PluginDefaults
- CSF_LANGUAGE
- CSF_SHMessage
- CSF_XCAFDefaults
- CSF_GraphicShr
- CSF_IGESDefaults
- CSF_STEPDefaults

System related environment variables:

- PATH

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/svnroot/free-cad/trunk/src
BuildRevisionDate	Date of the above Revision	2007/02/03 22:21:18
BuildScrClean	Indicates if the source was changed ager checkout	Src modified
BuildScrMixed		Src not mixed

Branding related

These Config entries are related to the branding mechanism of FreeCAD. See [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 Workbech which get started automaticly after Startup	"Part design"
HiddenDockWindow	List of dockwindows (separated by a semicolon) which will be disabled	"Property editor"

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Start_up_and_Configuration"

FreeCAD Build Tool

The **FreeCAD build tool** or **fbct** is a python script located at

```
trunc/src/Tools/fbct.py
```

It can be used to simplify some frequent tasks in building, distributing and extending FreeCAD.

Usage

With **Python** correctly installed, *fbct* can be invoked by the command

```
python fbct.py
```

It displays a menu, where you can select the task you want to use it for:

```
FreeCAD Build Tool
Usage:
  fbct <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:
fbct <command name> ?
```

At the input prompt enter the abbreviated command you want to call. For example type "CM" for [creating a module](#).

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](#).

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=FreeCAD_Build_Tool"

Module Creation

Creating a new application module in FreeCAD is rather simple. In the FreeCAD development tree exists the *FreeCAD Build Tool* (*fcbt*) that does the most important things for you. It is a *Python* script located under

```
trunk/src/Tools/fcft.py
```

When your python interpreter is correctly installed you can execute the script from a command line with

```
python fcft.py
```

It will display the following menu:

```
FreeCAD Build Tool
Usage:
  fcft <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:
  fcft <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* *fcft* 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!

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Module_Creation"

Debugging

Test first

Before you go through the pain of debugging use the test framework to check if the standard tests work properly. If not there is maybe a broken installation.

command line

The *debugging* of FreeCAD is supported by a few internal mechanisms. The command line version of FreeCAD provides to options for debugging support:

- v With the "v" option FreeCAD gives a more verbose output.
- l With the "l" option FreeCAD writes additional information to a logfile.

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Debugging>"

Testing

Introduction

FreeCAD comes with an extensive testing framework. The testing bases on a set of Python scripts which are located in the test module.

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Testing>"

Branding

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 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:

```
int main( int argc, char ** argv )
{
    // Name and Version of the Application
    App::Application::Config() ["ExeName"] = "FooApp.exe";
    App::Application::Config() ["ExeVersion"] = "0.7";

    // set the banner (for logging and console)
    App::Application::Config() ["ConsoleBanner"] = sBanner;
    App::Application::Config() ["AppIcon"] = "FCIcon";
    App::Application::Config() ["SplashPicture"] = "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;
}
```

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](#).

Images

All image resources are compiled into FreeCAD. This reduces delayed loading and keeps the installation compact. The images are included in XPM-Format which is basically a text format that uses C-syntax. You can basically draw this images with a text editor, but it is more comfortable to create the images with your favorite graphics program and convert it later to XPM format.

The GNU image program [Gimp](#) can save XPM file.

For conversion you can use the [ImageConv](#) tool which is included with freecad. You can find it under

```
/trunk/src/Tools/ImageTools/ImageConv
```

It can not only convert images but also automatically update the *BmpFactoryIcons.cpp* file, where the images are registered. The typical usage is as simple like the following example:

```
ImageConv -i InputImage.png -o OutputImage.xpm
```

This converts the file *InputImage.png* in XPM-format and writes the result to file *OutputImage.xpm*.

The line:

```
Gui::BitmapFactory().addXPM("FooAppSplasher", ( const char** ) splash_screen);
```

in the main() then include the image in the BitmapFactory of FreeCAD.

Icons

The main application icon *FCIcon* that appears in window titles and other places is defined in

```
/trunk/src/Gui/Icons/images.cpp
```

and starts with the line

```
static const char *FCIcon[]={
```

Replace it with your favourite icon, recompile freecad and the next step to create your own brand is done. There are many other icons in this file that you

might change to your gusto.

If you need to add new icons, you have to register it in

```
/trunk/src/Gui/Icons/BmpFactoryIcons.cpp
```

so that you can access from FreeCAD.

Background Image

The background image appears, when no document window is open. Like the splash screen it is defined in *developers.h* in the section starting with:

```
static const char* const background[]={
```

You should choose a low contrast image for the background. Otherwise it might irritate the user.

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Branding>"

Localisation

Localisation is in general the process of providing a Software with a multiple language user interface. In FreeCAD you can set the language of the user interface under *Editâ†Preferencesâ†Application*. FreeCAD uses **Qt** to enable multiple language support. On Unix/Linux systems, FreeCAD uses the current locale settings of your system by default.

Helping to translate FreeCAD

One of the very important things you can do for FreeCAD if you are not a programmer, is to help to translate the program in your language. To do so is now easier than ever, with the use of the **Crowdin** collaborative on-line translation system.

How to Translate

- Go to the [FreeCAD translation project page on Crowdin](#);
- Login by creating a new profile, or using a third-party account like your GMail address;
- Click on the language you wish to work on;
- Start translating by clicking on the Translate button next to one of the files. For example, *FreeCAD.ts* contains the text strings for the FreeCAD main GUI.
- You can vote for existing translations, or you can create your own.

Note: If you are actively taking part in translating FreeCAD and want to be informed before next release is ready to be launched, so there is time to review your translation, please subscribe to this issue: <http://sourceforge.net/apps/mantisbt/free-cad/view.php?id=137>

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](#) so we can include them in the freecad source code so they benefit other users too.

Available translation files

- [FreeCAD main GUI](#)
- [Complete Workbench](#)
- [Drawing Workbench](#)
- [Draft Workbench](#)
- [Reverse Engineering Workbench](#)
- [FEM Workbench](#)
- [Robot Workbench](#)
- [Image Workbench](#)
- [Sketcher Workbench](#)
- [Mesh Workbench](#)
- [Test Workbench](#)
- [Points Workbench](#)
- [Raytracing Workbench](#)
- [Part Workbench](#)
- [PartDesign Workbench](#)
- [Assembly Workbench](#)
- [MeshPart Workbench](#)

Preparing your own modules/applications for translation

Prerequisites

To localise your application module you need to have helpers that come with *Qt*. You can download them from the [Trolltech-Website](#), but they are also contained in the [LibPack](#):

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 your module and type on the command line:

```
qmake -project
```

This scans your project directory for files containing text strings and creates a project file like the following example:

```
#####  
# 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
```

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:

```
python ..\..\..\Tools\qembed.py "$(InputDir)\$(InputName).ts"  
    "$(InputDir)\$(InputName).h" "$(InputName)"
```

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

```
new Gui::LanguageProducer("Deutsch", <Modul>_de_h_data, <Modul>_de_h_len);
```

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:

```
pylupdate4 *.py -ts YourModule_fr.ts
```

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

```
lrelease YourModule_fr.ts
```

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):

```
translator = QtCore.QTranslator()  
translator.load("YourModule_"+languages[ln])  
QtGui.QApplication.installTranslator(translator)
```

Optionally, you can also create the file `XML Draft.qrc` with this content:

```
<RCC>  
<qresource prefix="/translations" >  
<file>Draft_fr.qm</file>  
</qresource>  
</RCC>
```

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 most up-to-date and interesting material is gathered in the [manual](#). 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](#) 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](#).

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 `Docnavfr`). Look at other already translated pages to see how they did it.

Then, once you translated a page, you must add to the original English page a link to your translation, so readers know that there is a translated version

available, using the `{{languages}}` template. Look at how other translators did it to do the same.

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.

Three templates are commonly used in manual pages. These 3 templates have localized versions (Template:Docnav/fr, Template:fr, etc...)

- **Template:Docnav** : it is the navigation bar at the bottom of the pages, showing previous and next pages
- **Template:Languages** : this template must be placed on original pages, to indicate the reader that a translation exists. The localised version must in return be placed on the translated page, to link to the original English page.
- **Template:en** : there is one of these for each language. They must be placed inside the language template.

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|Fonctionnalit s}}
```

The page "Bienvenue sur l'aide en ligne" redirects to Online Help Startpage/fr, and the page "Fonctionnalit s" redirects to Feature list/fr.

If you are unsure how to proceed, don't hesitate to ask for help in the [forum](#).

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Localisation>"

Extra python modules

The python interpreter inside FreeCAD can easily be extended by adding new modules to your system's python installation. Those modules will be automatically detected and used by FreeCAD.

All python modules can be used from within FreeCAD, but several of them, listed below, have a special importance because they allow python programs complete access to core functionality of FreeCAD. Examples of use of those modules can also be found on the [Code snippets](#) page.

Note: of the following modules, Pivy is now fully integrated into any FreeCAD installation package, and PyQt4 is also integrated in the Windows installation package.

PyQt4

homepage: <http://www.riverbankcomputing.co.uk/pyqt>

PyQt (version 4) is a python bindings library which allow programs to access, create or modify **Qt** interfaces. Since the FreeCAD interface is built with Qt, installing PyQt4 on your system allow python programs inside FreeCAD to access all the interface, modify its parts, create new widgets or gather info from interface parts.

PyQt is released under a multiple licensing system, same system as **used by Qt**. To resume, there is a commercial version and a free GPL version. If you want to use it to make commercial (closed source) programs, you need to purchase the commercial license, otherwise you can simply install and use freely the GPL version.

Installation

Before installing PyQt4, you obviously need a python environment installed and working.

Linux

The simplest way to install PyQt4 is through your distribution's package manager. On Debian/Ubuntu systems, the package name is generally *python-qt4*, while on RPM-based systems it is named *pyqt4*. The necessary dependencies (Qt and SIP) will be taken care of automatically.

Windows

The program can be downloaded from [here](#). You'll need to install the Qt and SIP libraries before installing pyqt4 (to be documented).

Usage

Once it is installed, you can check that everything is working by typing in FreeCAD python console:

```
import PyQt4
```

To access the FreeCAD interface, type:

```
from PyQt4 import QtCore, QtGui
app = QtGui.QApp
FreeCADWindow = app.activeWindow()
```

Now you can start to explore the interface with the `dir()` command. You can add new elements, like a custom widget, with commands like:

```
FreeCADWindow.addDockWidget(QtCore.Qt.RightDockWidgetArea, my_custom_widget)
```

Documentation

More PyQt4 tutorials (including how to build interfaces with Qt Designer to use with python):

<http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/classes.html> - the official PyQt4 API Reference

<http://www.rkblog.rk.edu.pl/w/p/introduction-pyqt4/> - a simple introduction

<http://www.zetcode.com/tutorials/pyqt4/> - very complete in-depth tutorial

Pivy

homepage: <http://pivy.coin3d.org/>

Pivy is a **coin** bindings library for python, officially supported by coin. Coin itself is a toolkit for building 3D applications in OpenGL. It is the toolkit that FreeCAD uses to draw its 3d Scene on the screen. Installing Pivy on your system will allow python programs to access the FreeCAD scenegraph, draw new objects on the scene and use the wide range of available Coin tools for drawing operations. Coin is based on the open Inventor scene description language. Pivy is used by the 2D drafting module of FreeCAD (and also by the complete module), so it is needed if you want to use any tool of those modules.

It is important to know that FreeCAD only uses coin for representation of objects on the screen, which is separated from the definition of objects. This means that pivy won't be able to modify existing objects, neither to create valid FreeCAD objects. But it can be used to draw all kind of temporary things on screen, such as axis, grids, manipulators, construction geometry, etc...

Pivy, as well as Coin, is released under a GPL license.

Installation

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](#) 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> (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:

```
patch -p1 < ../pivy_0.5.0~svn765-2.diff
```

then

```
debuild
```

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:

```
svn co https://svn.coin3d.org/repos/Pivy/trunk Pivy
```

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 a 1.3.25 source tarball from <http://www.swig.org>. Then unpack it and from a command line do (as root):

```
./configure
```



```
make
make install (or checkinstall if you use it)
```

It takes just a few seconds to build.

After that go to the pivy sources and call

```
python setup.py build
```

which creates the source files. 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):

```
python setup.py install (or checkinstall python setup.py install)
```

That's it, pivy is installed.

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

```
set PATH=path_to_python_2.5;%PATH%
```

To get pivy working you should get the latest sources from the project's repository:

```
svn co https://svn.coin3d.org/repos/Pivy/trunk Pivy
```

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>. Then unpack it and from the command line add it to the system path

```
set PATH=path_to_swig_1.3.25;%PATH%
```

and set COINDIR to the appropriate path

```
set COINDIR=path_to_coin
```

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.') (do not remove the closing parenthesis).

After that go to the pivy sources and call

```
python setup.py build
```

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

```
set INCLUDE=%INCLUDE%;path_to_coin_include_dir
```

and if the SoQt headers are not in the same place as the Coin headers also

```
set INCLUDE=%INCLUDE%;path_to_soqt_include_dir
```

and finally the Qt headers

```
set INCLUDE=%INCLUDE%;path_to_qt4\include\Qt
```

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

```
vsbase = r"Software\Microsoft\VisualStudio\%0.1f" % version
```

with

```
vsbase = r"Software\Microsoft\VCExpress\%0.1f" % version
```

Then retry again. If you get a second error like

```
error: Python was built with Visual Studio 2003;...
```

you must also replace line 128

```
self.set_macro("FrameworkSDKDir", net, "sdkinstallrootv1.1")
```

with

```
self.set_macro("FrameworkSDKDir", net, "sdkinstallrootv2.0")
```

Retry once again. If you get again an error like

```
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.
```

then you should check the environment variables DISTUTILS_USE_SDK and MSSDK with

```
echo %DISTUTILS_USE_SDK%
echo %MSSDK%
```

If not yet set then just set it e.g. to 1

```
set DISTUTILS_USE_SDK=1
set MSSDK=1
```

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 have Pivy access the FreeCAD scenegraph do the following:

```
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
```

You can now explore the FCSceneGraph with the dir() command.

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:

<http://doc.coin3d.org/Coin/classes.html> - Coin3D API Reference

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.

Online version: "http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Extra_python_modules"

Contributors

Here 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](#) page.

Development

Project managers

Lead developers of the FreeCAD project

- [Jürgen Riegel](#)
- [Werner Mayer](#)

Main developers

People who work regularly on the FreeCAD code

- [Yorik van Havre](#)
- [Logari81](#)
- [Mrlukeparry](#)

Other coders

People who contributed with code to the FreeCAD project

- Graeme van der Vlugt
- Berthold Grupp
- [Georg Wiora](#)
- Martin Burbaum
- Jacques-Antoine Gaudin
- Ken Cline
- Dmitry Chigrin

Companies

Companies which donated Code or Developer:

- Imetric 3D

Community

People from the community who put a lot of efforts in helping the FreeCAD project

- [Normandc](#)
- Kwahoo
- [Eduardo Magdalena](#)

Online version: "<http://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Contributors>"